Module 3:

Programming in assembly

Introduction to computers II

José Manuel Mendías Cuadros
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid

Outline

- ✓ Introduction.
- ✓ Assembly language.
- ✓ Pseudo-instructions.
- ✓ Variables and constants.
- ✓ Expressions.
- ✓ Code organization.
- ✓ Functions.
- ✓ Development workflow.

These slides are based on:

- S.L. Harris and D. Harris. *Digital Design and Computer Architecture. RISC-V Edition.*
- D.A. Patterson and J.L. Hennessy. *Computer Organization and Design. RISC-V Edition.*

Introduction

- Computers execute machine code, but programmers develop software using high level languages.

- The assembly code is the halfway point between HW and SW:
  - High level programs are compiled into assembly.
  - Assembly programs are assembled into machine code.

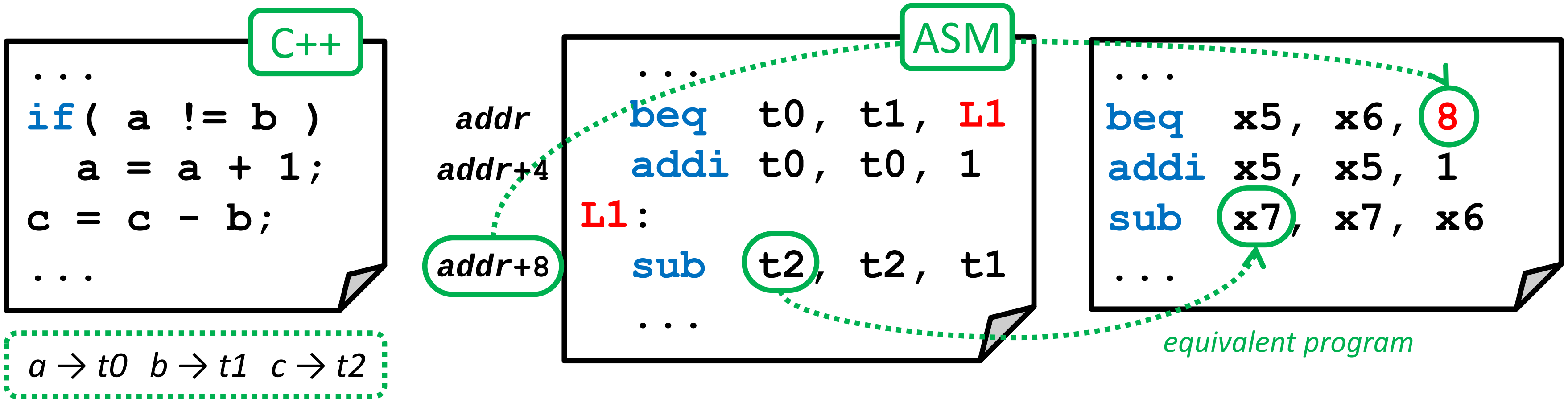
- An assembly program is mostly composed of assembly instructions
  - But it can also have other elements that facilitate programming: labels, directives, comments, pseudo-instructions, etc...

- Nowadays, practically nobody programs in assembly, but it is important to know how to do it because it helps:
  - Learn the abstraction layer of the processor architecture.
  - Understand how compilers translate high level programs into assembly.

Assembly language

Elements of a program (i)

- An assembly language program:
  - Is composed of a sequence of instructions that will be located in memory in the same order to be serially executed.
    - Most of the times, using alias to indicate used registers.
  - Instruction or data memory addresses are not explicitly indicated.
    - But consecutive instructions will have consecutive addresses.
  - If there is a branch to a certain instruction, a label can be defined to refer its address symbolically.
    - Labels exempt programmers from calculating the PC-relative offsets required by the branch instructions.



```
...
if( a != b )
   a = a + 1;
c = c - b;
...
```

a → t0 b → t1 c → t2

```
   ...
   beq   t0, t1, L1
   addi  t0, t0, 1
L1:
   sub   t2, t2, t1
   ...
```

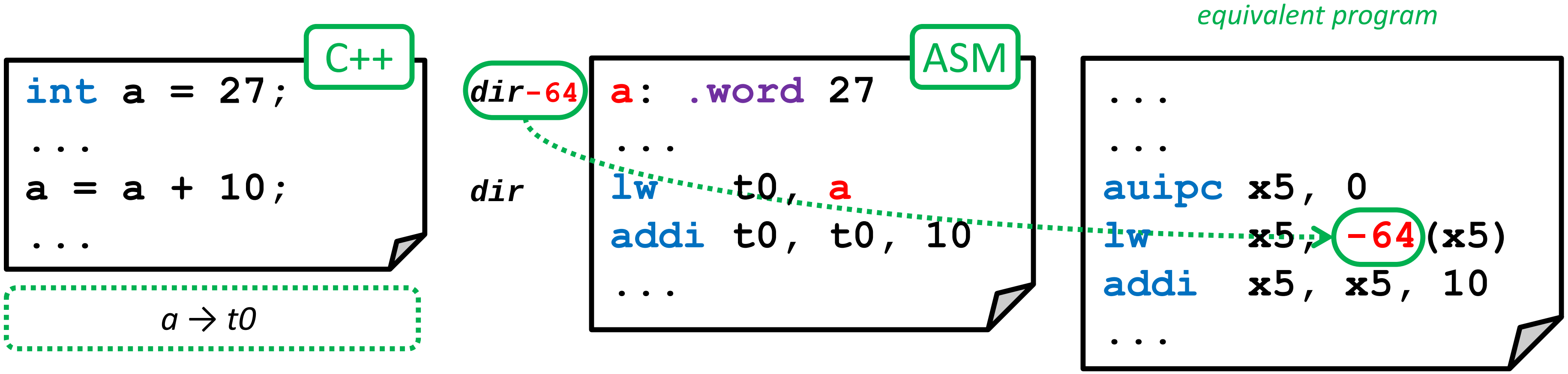
```
...
beq  x5, x6, 8
addi x5, x5, 1
sub  x7, x7, x6
...
```

equivalent program

Assembly language

Elements of a program (ii)

- An assembly program can also contain:
  - Labels to refer data addresses symbolically.
    - They exempt programmers from the 32b absolute address management



C++

```
int a = 27;
...
a = a + 10;
...
```

a → t0

ASM

```
a:  .word 27
...
lw     t0, a
addi  t0, t0, 10
...
```

equivalent program

```
...
...
auipc  x5, 0
lw     x5, -64(x5)
addi   x5, x5, 10
...
```

-
  - Symbols to refer immediate constants symbolically.
    - They facilitate code readability

C++

```
#define N 10
...
a = a + N;
...
```

ASM

```
.equ N, 10
...
addi  t0, t0, N
...
```

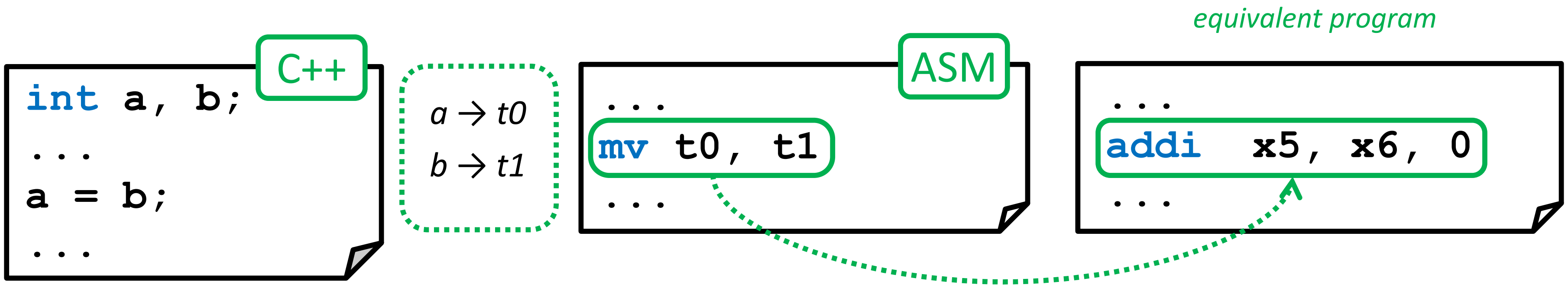
equivalent program

```
...
...
addi   x5, x5, 10
...
```

Assembly language

Elements of a program (iii)

- Apart from instructions, an assembly program can contain:
  - Pseudo-instructions: these are aliases of instructions.
    - They facilitate programmers the use of frequent instructions



C++

```
int a, b;
...
a = b;
...
```

a → t0

b → t1

ASM

```
...
mv t0, t1
...
```

equivalent program

```
...
addi   x5, x6, 0
...
```

  - Comments: explanatory text that is not translated into machine code
    - They facilitate code readability

ASM

```
...
addi t0, t0, 1   # Increment a
...
```

equivalent program

```
...
addi x5, x5, 1
...
```

  - Other directives: to control the assembly process.
    - Code and data location, alignment, etc...

Assembly language

Elements of a program (iv)

- An assembly program is a sequence of lines, in which each line contains at most one of each of the following elements:
  - Label: Symbolic reference to an instruction or data address.
    - Each label must start with a letter and end with a colon (:).
    - To use the label, the final colon is omitted.
    - If a label is alone in a line, it refers to the address of the following instruction or data.
  - Assembly instruction: composed of an instruction and its operands.
    - Operands may be explicit or symbolic.
    - There can be a pseudo-instruction, instead of an instruction.
  - Directive: Auxiliary indication used during the assembly process.
    - All directives start with a dot (.)
  - Comment: Free text added by the programmer
    - Comments may be at the end of a line or occupy the whole line.
    - They start with # but also // and /* */ may be used.

Assembly language

Sections

- An assembly program is divided into sections.

- A section represents an adjacent memory region, in which a set of data/instructions with the same purpose will be located.
  - Consecutive instructions/data within a section will have consecutive memory addresses.
  - During the linking process, each section can be placed in a different memory region.

- An assembly process consists of 3 sections:
  - **`text`**: contains the instructions of the program.
  - **`data`**: contains the constants and global variables with their initial value.
  - **`bss`**: contains global variables without an initial value.
  - Other sections can be declared using a special directive (**`.section`**)

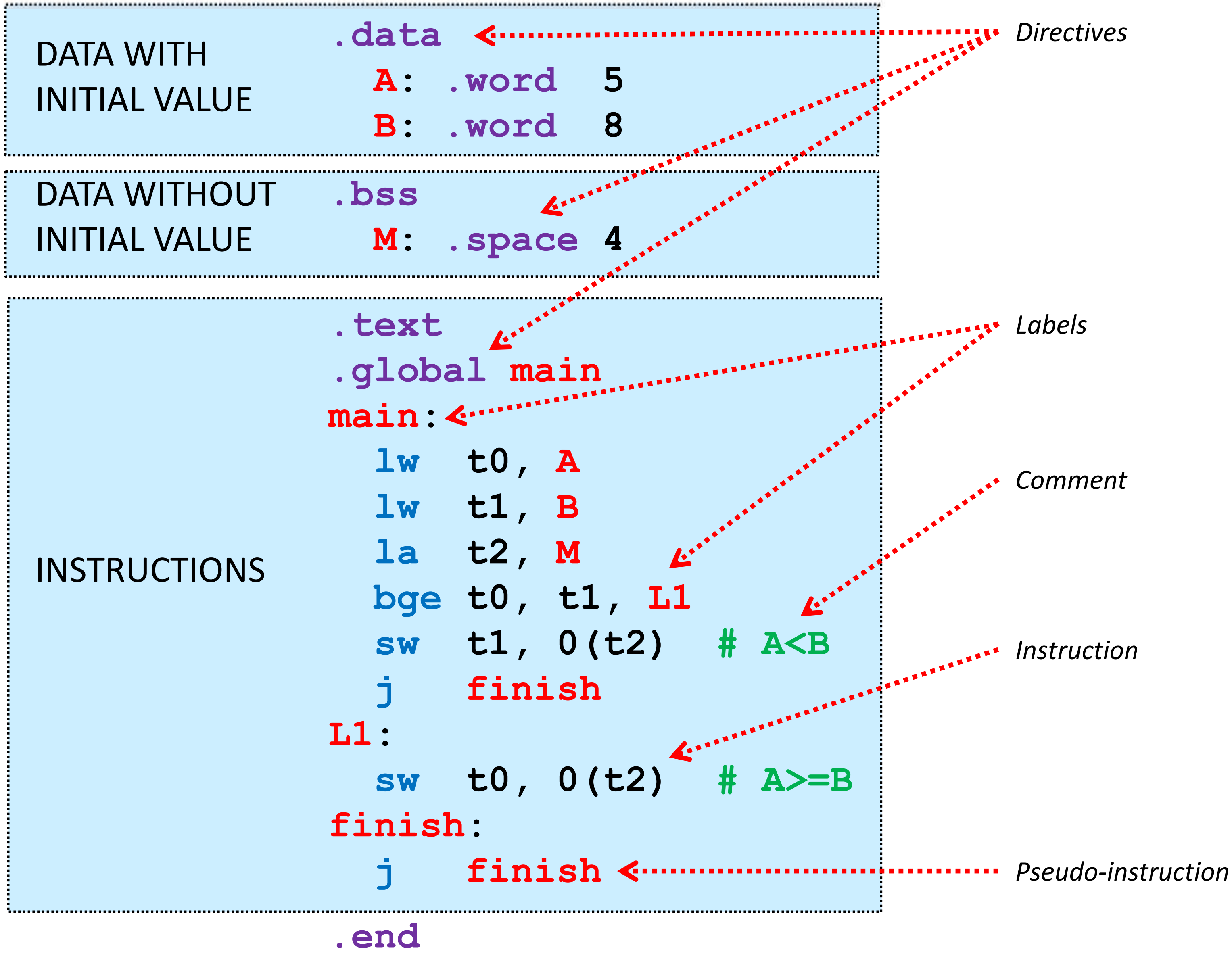
Assembly language

Directives

Directive	Description
`.text`	Declares the beginning of the instruction section
`.data`	Declares the beginning of the global variable section with initial value
`.bss`	Declares the beginning of the global variable section without initial value
`.word` $w_1, … w_n$	Reserves memory space for *n* words with initial values $w_1 … w_n$
`.half` $h_1, … h_n$	Reserves memory space for *n* half words with initial values $h_1 … h_n$
`.byte` $b_1, … b_n$	Reserves memory space for *n* bytes with initial values $b_1 … b_n$
`.zero` *n*	Reserves memory space for *n* bytes with initial values equal to 0
`.space` *n*	Reserves memory space for *n* bytes without an initial value
`.string` *"str"*	Reserves memory space whose initial value is the *"str“ string*
`.align` *n*	Aligns data/instructions to addresses that are multiple of 2^n
`.equ` *sym, val*	Define a symbolic constant with the name *sym* and value *val*
`.global` *sym*	Makes label *sym* visible out of the file that contains it (global)
`.extern` *sym*	Indicates that a symbol is defined in another file of the project
`.end`	Declares the end of the assembly program

Assembly language

Example (i)



DATA WITH INITIAL VALUE

```
.data
  A:  .word   5
  B:  .word   8
```

DATA WITHOUT INITIAL VALUE

```
.bss
  M:  .space  4
```

INSTRUCTIONS

```
.text
.global main
main:
  lw   t0, A
  lw   t1, B
  la   t2, M
  bge  t0, t1, L1
  sw   t1, 0(t2)    # A<B
  j    finish
L1:
  sw   t0, 0(t2)    # A>=B
finish:
  j    finish
.end
```

Assembly language

Example (ii)

DATA WITH INITIAL VALUE

```
.data
   A:  .word   5
   B:  .word   8
```

DATA WITHOUT INITIAL VALUE

```
.bss
   M:  .space  4
```

INSTRUCTIONS

```
.text
.global main
main:
   lw   t0, A
   lw   t1, B
   la   t2, M
   bge  t0, t1, L1
   sw   t1, 0(t2)    # A<B
   j    finish
L1:
   sw   t0, 0(t2)    # A>=B
finish:
   j    finish
.end
```

Indicates the beginning of the input data section

Input data defined as words

Indicates the beginning of the output data section

Output data defined as a word

Indicates the beginning of the code section

Makes this label visible out of this file, so that the simulator can know the initial address of the program

Indicates the end of the assembly program

Assembly language

Example (iii)

DATA WITH INITIAL VALUE

```
.data
   A:  .word   5
   B:  .word   8
```

DATA WITHOUT INITIAL VALUE

```
.bss
   M:  .space  4
```

INSTRUCTIONS

```
.text
.global main
main:
   lw   t0, A
   lw   t1, B
   la   t2, M
   bge  t0, t1, L1
   sw   t1, 0(t2)   # A<B
   j    finish
L1:
   sw   t0, 0(t2)   # A>=B
finish:
   j    finish
.end
```

Loads the value inside address `A` *into* `t0`

Loads the value inside address `B` *into* `t1`

Loads address `M` *into* `t2`

Compares the loaded values

Stores the value of `t1 (B)` *into address* `M`

Branches to the last instruction of the program

Stores the value of `t0 (A)` *into address* `M`

This instruction is executed indefinitely

Pseudo-instructions

- Pseudo-instructions are aliases of very usual cases of certain instructions.
    - They perform specific operations with their own mnemonic and operands.
    - During the assembly process, they are translated into actual instructions that have an equivalent behavior.
    - They enrich the assembly language without adding HW complexity.

- For example, it is quite common to copy the content of a register into another one:
    - To do this, RISC-V has the `mv` pseudo-instruction, with 2 operands:

`mv rd, rs1`	rd ← rs1

- But during the assembly process, it is translated into this actual instruction:

`addi rd, rs1, 0`	rd ← rs1 + 0

- Programmers may use one or the other, indistinctly.

Pseudo-instructions

Arithmetic-logic (i)

- In assembly, comparisons with 0 are very common.

Instruction	Operation	Translation	Description
`seqz rd, rs1`	rd ← *if* (rs1 = 0) *then* (1) *else* (0)	`sltiu rd, rs1, 1`	set if **eq**ual to **z**ero
`snez rd, rs1`	rd ← *if* (rs1 ≠ 0) *then* (1) *else* (0)	`sltu rd, x0, rs1`	set if **n**ot **e**qual to **z**ero
`sltz rd, rs1`	rd ← *if* (rs1 < 0) *then* (1) *else* (0)	`slt rd, rs1, x0`	set if **l**ess **t**han **z**ero
`sgtz rd, rs1`	rd ← *if* (rs1 > 0) *then* (1) *else* (0)	`slt rd, x0, rs1`	set if **g**reater **t**han **z**ero

- Changing the sign of an operand is also quite common.

Instruction	Operation	Translation	Description
`neg rd, rs1`	rd ← − rs1	`sub rd, x0, rs1`	**neg**ate opposite

Pseudo-instructions

Arithmetic-logic (ii)

- This performs the bitwise complement of an operand.

Instruction	Operation	Translation	Description
`not` *rd*, *rs1*	rd ← ~ rs1	`xori` **rd, rs1, -1**	**not** bitwise logical NOT

- It is also useful to rename instructions that operate with an immediate.
  - This allows using the same mnemonic for register and immediate operands.

Instruction	Operation	Translation
`add` *rd*, *rs1*, imm_{12b}	rd ← rs1 + sExt(imm)	`addi` *rd*, *rs1*, *imm*
`slt` *rd*, *rs1*, imm_{12b}	rd ← *if* (rs1 $<_S$ sExt(imm)) *then* (1) *else* (0)	`slti` *rd*, *rs1*, *imm*
`sltu` *rd*, *rs1*, imm_{12b}	rd ← *if* (rs1 $<_U$ sExt(imm)) *then* (1) *else* (0)	`sltiu` *rd*, *rs1*, *imm*
`and` *rd*, *rs1*, imm_{12b}	rd ← rs1 & sExt(imm)	`andi` *rd*, *rs1*, *imm*
`or` *rd*, *rs1*, imm_{12b}	rd ← rs1 | sExt(imm)	`ori` *rd*, *rs1*, *imm*
`xor` *rd*, *rs1*, imm_{12b}	rd ← rs1 ^ sExt(imm)	`xori` *rd*, *rs1*, *imm*

Pseudo-instructions

Shift

- Same for shift instructions with immediate operands

Instruction	Operation	Translation
`sll` ***rd, rs1, imm***$_{5b}$	rd ← rs1 << imm	`slli` ***rd, rs1, imm***
`srl` ***rd, rs1, imm***$_{5b}$	rd ← rs1 >> imm	`srli` ***rd, rs1, imm***
`sra` ***rd, rs1, imm***$_{5b}$	rd ← rs1 >>> imm	`srai` ***rd, rs1, imm***

Pseudo-instructions

Data transfer (i)

- It is common to load/store data with a given absolute address
    - Normally, PC-relative addressing is used in this case.
    - These pseudo-instructions avoid address calculations.

Instruction	**Operation**	**Translation**
`lb` $rd,\ imm_{32b}$	$rd \leftarrow sExt(\ Mem[\ PC + imm\ ]_{7:0}\ )$	`auipc` $rd,\ imm_{31:12}^{*}$ `lb` $rd,\ imm_{11:0}(rd)$
`lh` $rd,\ imm_{32b}$	$rd \leftarrow sExt(\ Mem[\ PC + imm\ ]_{15:0}\ )$	`auipc` $rd,\ imm_{31:12}^{*}$ `lh` $rd,\ imm_{11:0}(rd)$
`lw` $rd,\ imm_{32b}$	$rd \leftarrow sExt(\ Mem[\ PC + imm\ ]_{31:0}\ )$	`auipc` $rd,\ imm_{31:12}^{*}$ `lw` $rd,\ imm_{11:0}(rd)$
`sb` $rs2,\ imm_{32b},\ rs1$	$Mem[\ PC + imm\ ]_{7:0} \leftarrow rs2_{7:0}$	`auipc` $rs1,\ imm_{31:12}^{*}$ `sb` $rs2,\ imm_{11:0}(rs1)$
`sh` $rs2,\ imm_{32b},\ rs1$	$Mem[\ PC + imm\ ]_{15:0} \leftarrow rs2_{15:0}$	`auipc` $rs1,\ imm_{31:12}^{*}$ `sh` $rs2,\ imm_{11:0}(rs1)$
`sw` $rs2,\ imm_{32b},\ rs1$	$Mem[\ PC + imm\ ]_{31:0} \leftarrow rs2_{31:0}$	`auipc` $rs1,\ imm_{31:12}^{*}$ `sw` $rs2,\ imm_{11:0}(rs1)$

(*) If imm_{11} is 1, the value of $imm_{31:12}$ will be incremented by 1

Pseudo-instructions

Data transfer (ii)

- To copy data from a register to another one.

Instruction	Operation	Translation	Description
`mv rd, rs1`	rd ← rs1	`addi rd, rs1, 0`	**m**o**v**e register copy

- To load a constant into a register.

Instruction	Operation	Translation	Description
`li rd,` imm_{12b}	rd ← sExt(imm)	`addi rd, x0, imm`	**l**oad **i**mmediate 12 bits
`li rd,` imm_{32b}	rd ← imm	`lui rd,` $imm_{31:12}{}^{*}$ `addi rd, rd,` $imm_{11:0}$	**l**oad **i**mmediate 32 bits

- To load the address of a data located in memory.

Instruction	Operation	Translation	Description
`la rd,` imm_{32b}	rd ← PC + imm	`auipc rd,` $imm_{31:12}{}^{*}$ `addi rd, rd,` $imm_{11:0}$	**l**oad **a**ddress 32 bits

Pseudo-instructions

Conditional branch (i)

- It is common to perform any kind of comparison in conditional branches:
  - In the actual ISA, there are only these comparison types: =, ≠, < y ≥
  - For the others (≤ , >), programmers must change the operand order.
  - There are pseudo-instructions to facilitate these: ≤ , >

Instruction	Operation	Translation	Description
ble ***rs1, rs2,*** imm_{13b}	*if* (rs1 $\leq_S$ rs2) *then* (PC ← PC + sExt($imm_{12:1}$ << 1))	**bge** ***rs2, rs1, imm***	branch if less than or equal signed
bgt ***rs1, rs2,*** imm_{13b}	*if* (rs1 $>_S$ rs2) *then* (PC ← PC + sExt($imm_{12:1}$ << 1))	**blt** ***rs2, rs1, imm***	branch if greater than signed
bleu ***rs1, rs2,*** imm_{13b}	*if* (rs1 $\leq_U$ rs2) *then* (PC ← PC + sExt($imm_{12:1}$ << 1))	**bgeu** ***rs2, rs1, imm***	branch if less than or equal unsigned unsigned
bgtu ***rs1, rs2,*** imm_{13b}	*if* (rs1 $>_U$ rs2) *then* (PC ← PC + sExt($imm_{12:1}$ << 1))	**bltu** ***rs2, rs1, imm***	branch if greater than unsigned unsigned

Pseudo-instructions

Conditional branch (ii)

- In particular, comparisons with 0 in branches are the most common

Instruction	Operation	Translation	Description
`beqz` *rs1*, *imm*$_{13b}$	*if* (rs1 = 0) *then* (PC ← PC + sExt(imm$_{12:1}$ << 1))	`beq` *rs1*, `x0`, *imm*	branch if equal to zero
`bnez` *rs1*, *imm*$_{13b}$	*if* (rs1 ≠ 0) *then* (PC ← PC + sExt(imm$_{12:1}$ << 1))	`bne` *rs1*, `x0`, *imm*	branch if not equal to zero
`bltz` *rs1*, *imm*$_{13b}$	*if* (rs1 < 0) *then* (PC ← PC + sExt(imm$_{12:1}$ << 1))	`blt` *rs1*, `x0`, *imm*	branch if less than zero
`bgez` *rs1*, *imm*$_{13b}$	*if* (rs1 ≥ 0) *then* (PC ← PC + sExt(imm$_{12:1}$ << 1))	`bge` *rs1*, `x0`, *imm*	branch if greater than or equal to zero
`blez` *rs1*, *imm*$_{13b}$	*if* (rs1 ≤ 0) *then* (PC ← PC + sExt(imm$_{12:1}$ << 1))	`bge` `zero`, *rs1*, *imm*	branch if less than or equal to zero
`bgtz` *rs1*, *imm*$_{13b}$	*if* (rs1 > 0) *then* (PC ← PC + sExt(imm$_{12:1}$ << 1))	`blt` `zero`, *rs1*, *imm*	branch if greater than zero

Pseudo-instructions

Branch to function (ii)

- Register **`ra`** (alias of **`x1`**) is frequently used as the register to store the return address when branching to a function.
  - There are pseudo-instructions that use it explicitly.

Instruction	Operation	Translation	Description
`jalr` ***rs1***	PC ← rs1 ra ← PC+4	`jalr ra, rs1, 0`	jump and link register branch to a function with base addressing
`jal` imm_{21b}	PC ← PC + sExt($imm_{20:1}$ << 1) ra ← PC+4	`jal ra,` ***imm***	jump and link branch to a function with PC-relative addressing (nearby)
`call` imm_{21b}	PC ← PC + sExt($imm_{20:1}$ << 1) ra ← PC+4	`jal ra,` ***imm***	**call** branch to a function with PC-relative addressing (nearby)
`call` imm_{32b}	PC ← PC + imm ra ← PC+4	`auipc ra,` $imm_{31:12}{}^{*}$ `jalr ra, ra,` $imm_{11:0}$	**call** branch to a function with PC-relative addressing (faraway)
`ret`	PC ← ra	`jalr x0, ra, 0`	**return** Return from function

Pseudo-instructions

Others

- Unconditional branches are very useful, but these do not exist in the RISC-V ISA.

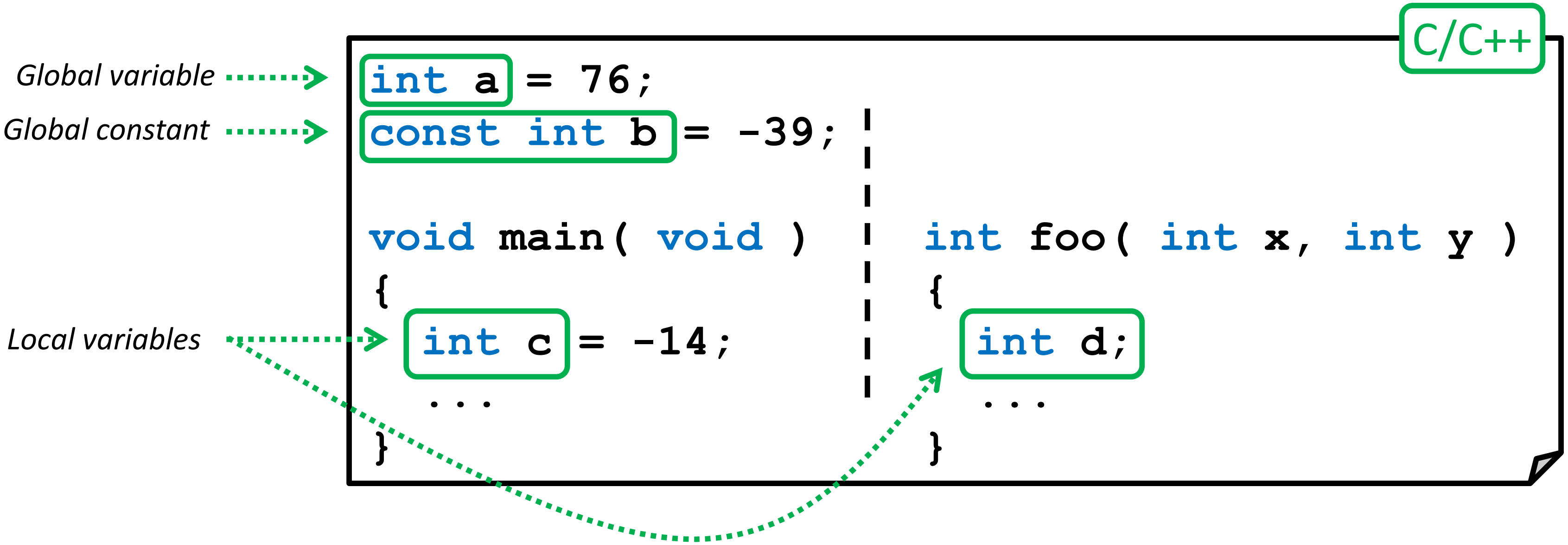
Instruction	Operation	Translation	Description
`j` *imm_{21b}*	PC ← PC + sExt($imm_{20:1}$ << 1)	`jal x0, imm`	**j**ump unconditional branch (immediate)
`jr rs1`	PC ← rs1	`jalr x0, rs1, 0`	**j**ump **r**egister unconditional branch (register)

- Sometimes, it is useful to do nothing.

Instruction	Operation	Translation	Description
`nop`	---	`addi x0, x0, 0`	**n**o **op**eration does nothing

Variables and constants

- Variables in C/C++ are typed and can be global or local.
  - A global variable is declared outside of functions
    - It is visible from any point of the program.
    - They persist during the whole execution of the program (static).
  - A local variable is declared inside a function.
    - It is only visible inside the function where it is declared.
    - By default, it only persists during the execution of the function (automatic).
    - The formal parameters of a function behave as local variables.



Global variable

Global constant

Local variables

C/C++

```
int a = 76;
const int b = -39;

void main( void )            int foo( int x, int y )
{                            {
    int c = -14;                 int d;
    ...                          ...
}                            }
```

Variables and constants

- There are no actual variables in assembly
  - Thera are data placed in memory, in registers or alternating both locations
    - To operate with them, they must be placed in registers because there are no instructions with operands in memory in RISC-V.
  - In the same way, the memory address or register where the data is located can change throughout the execution of the program.
    - Programmers must trace the location in which the data is at each time.

- There is no distinction between variables and constants in assembly.
  - If the data changes throughout the execution of the program, we consider it to be a variable.
  - If it does not change, we consider it a constant.

- Besides, constants can exist in memory or in the instruction itself (as immediate operands).

Variables and constants

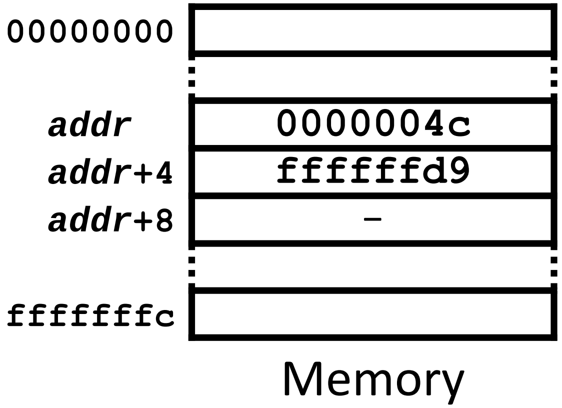
- In the case of global variables/constants, labels are used to avoid the use of explicit addresses in the assembly code.
  - These labels are equivalent to the name of the variables in assembly.

C/C++

```c
int a = 76;
const int b = -39;
int c;
...
```

ASM

```asm
a:  .word   76
b:  .word  -39
c:  .space  4
...
```



- In assembly, a constant value may be expressed as
  - Decimal: `109`
  - Hexadecimal, with the `0x` prefix: `0x6d`
  - Binary, with the `0b` prefix: `0b1101101`

Variables and constants

- In short, the input and output data of a program can be handled as global variables.
  - Initially, the input data of the program are located in memory.
    - Because they have been received from a peripheral.
    - Because they have an initial value (determined by the programmer or calculated by a previous program).
  - The output data of the program will also have to be stored in memory.
    - To transmit them to a peripheral or to be used by another program.
  - Data addresses will be fixed and known by the programmer.
    - Therefore, input and output data will be identified by labels.

- Since the number of registers is limited:
  - The rest of the program data are mostly located in memory.
- But, since accessing a register is much faster than accessing the memory:
  - Data must be kept in register as long as possible
  - Since it is not possible to keep them all, the most used data are kept.

Variables and constants

- Data placed in memory:
  - Must be loaded in registers to operate with them.
  - Once the result is calculated, it will be stored in memory.

C/C++

```
int a = 5;
...
a = a + 1;
...
```

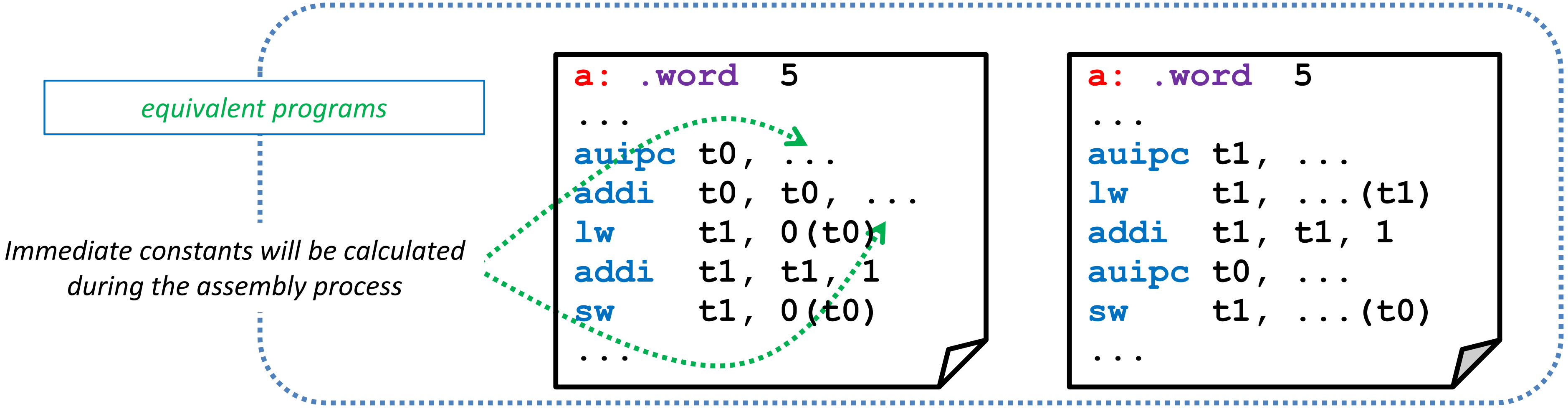
&a → t0   a → t1

ASM

```
a:  .word   5
...
la    t0, a
lw    t1, 0(t0)
addi  t1, t1, 1
sw    t1, 0(t0)
...
```

ASM

```
a:  .word   5
...
lw    t1, a
addi  t1, t1, 1
sw    t1, a, t0
...
```



equivalent programs

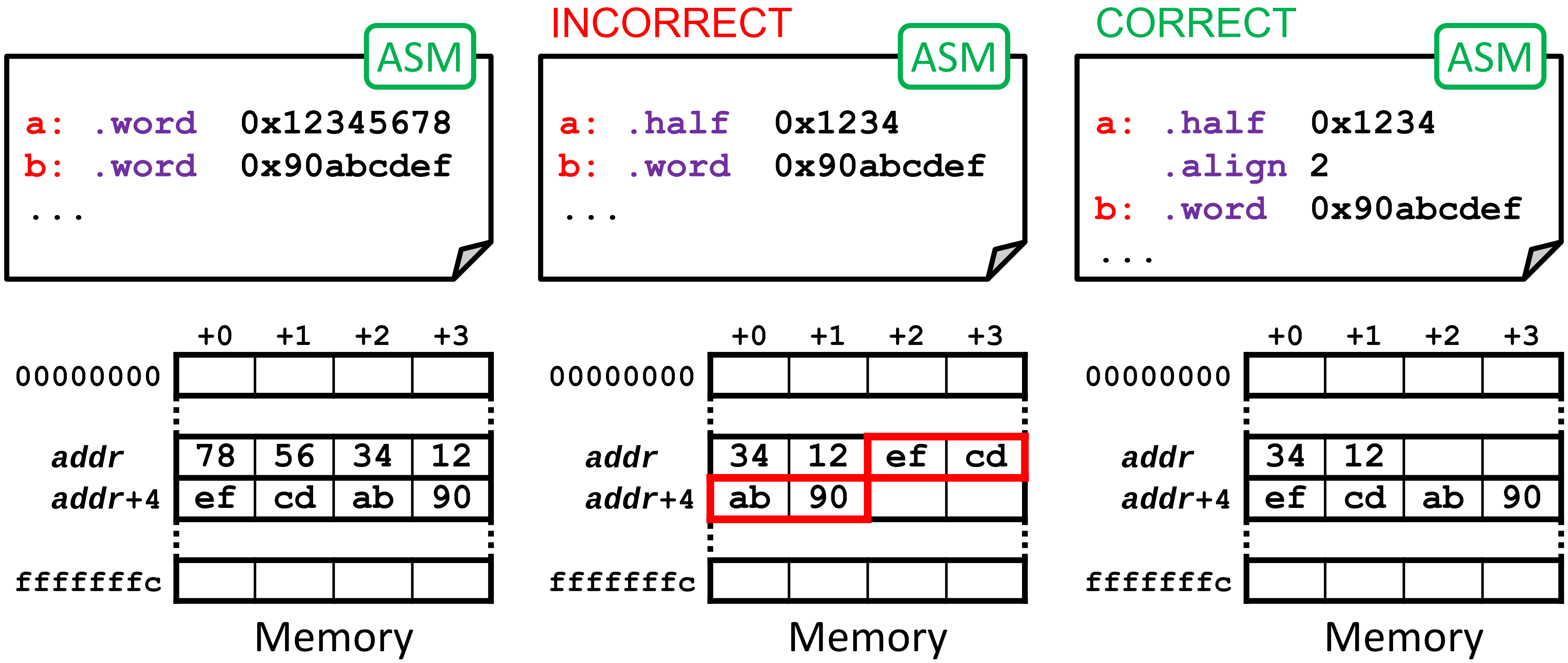
Immediate constants will be calculated during the assembly process

```
a:  .word   5
...
auipc  t0, ...
addi   t0, t0, ...
lw     t1, 0(t0)
addi   t1, t1, 1
sw     t1, 0(t0)
...
```

```
a:  .word   5
...
auipc  t1, ...
lw     t1, ...(t1)
addi   t1, t1, 1
auipc  t0, ...
sw     t1, ...(t0)
...
```

Variables and constants

- As with the instructions, data are placed in memory following the same order that they have in the assembly program
  - When loading (during execution) data with different sizes that are consecutively stored, alignment errors may happen.
  - To align them correctly, the `.align` directive is used



ASM

```
a:  .word   0x12345678
b:  .word   0x90abcdef
...
```

	+0	+1	+2	+3
00000000				
addr	78	56	34	12
addr+4	ef	cd	ab	90
fffffffc				

Memory

INCORRECT

ASM

```
a:  .half   0x1234
b:  .word   0x90abcdef
...
```

	+0	+1	+2	+3
00000000				
addr	34	12	ef	cd
addr+4	ab	90		
fffffffc				

Memory

CORRECT

ASM

```
a:  .half   0x1234
    .align  2
b:  .word   0x90abcdef
...
```

	+0	+1	+2	+3
00000000				
addr	34	12		
addr+4	ef	cd	ab	90
fffffffc				

Memory

Variables and constants

Types (i)

- In assembly, variables are not explicitly typed.
  - Data have a certain bit width, without a reference to how they are encoded.
  - Programmers should keep the coherence between the data encoding and the instructions that operate with it.
- The equivalence between C/C++ types and bit widths in assembler is:

C/C++ type	Width	Declaration	Load
`[signed] char`	8b = 1B	`.byte / .space 1`	`lb`
`unsigned char`	8b = 1B	`.byte / .space 1`	`lbu`
`[signed] short [int]`	16b = 2B	`.half / .space 2`	`lh`
`unsigned short [int]`	16b = 2B	`.half / .space 2`	`lhu`
`[signed] int`	32b = 4B	`.word / .space 4`	`lw`
`unsigned int`	32b = 4B	`.word / .space 4`	`lw`
pointer (address)	32b = 4B	`.word / .space 4`	`lw`

Variables and constants

Types (ii)

- The load instruction to use is different depending on the data width and whether they are signed or unsigned.

C/C++
```
unsigned char a = 5;
...
a = a + 1;
...
```

C/C++
```
short a = 5;
...
a = a + 1;
...
```

C/C++
```
int a = 5;
...
a = a + 1;
...
```

&a → t0
a → t1

ASM
```
a: .byte   5
...
la    t0, a
lbu   t1, 0(t0)
addi  t1, t1, 1
sb    t1, 0(t0)
...
```

ASM
```
a: .half   5
...
la    t0, a
lh    t1, 0(t0)
addi  t1, t1, 1
sh    t1, 0(t0)
...
```

ASM
```
a: .word   5
...
la    t0, a
lw    t1, 0(t0)
addi  t1, t1, 1
sw    t1, 0(t0)
...
```

Variables and constants

Arrays (i)

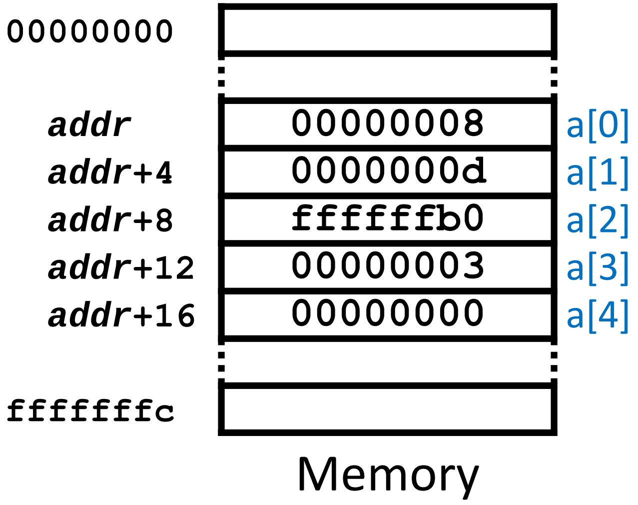
- An array is a collection of data with the same size located in consecutive memory addresses, in increasing index order.
  - The index indicates the relative position of the data respect to the first one.

C/C++

```
int a[5] = { 8, 13, -80, 3, 0 };
...
```

ASM

```
a: .word  8, 13, -80, 3, 0
...
```



00000000		
addr	00000008	a[0]
addr+4	0000000d	a[1]
addr+8	ffffffb0	a[2]
addr+12	00000003	a[3]
addr+16	00000000	a[4]
fffffffc		

Memory

- To access an array element, its address has to be calculated:
  - It is the sum of the array base address plus an offset
    - The array base address is the address of its first element.
  - The offset in bytes is calculated:

offset (bytes) = index × data size (bytes)

Variables and constants

Arrays (ii)

- The offset is calculated by the programmer if the index is a constant.
- If the index is variable, it has to be calculated by the program.

C/C++
```
int a[5];
...
a[0] = a[1] + a[2];
...
```

ASM
```
a:  .space 20
...
la   t0, a
lw   t1, 4(t0)
lw   t2, 8(t0)
add  t1, t1, t2
sw   t1, 0(t0)
...
```

Loads `a[1]` → `lw t1, 4(t0)`

Loads `a[2]` → `lw t2, 8(t0)`

Stores `a[0]` → `sw t1, 0(t0)`

C/C++
```
int a[5], i;
...
a[i] = a[i] + 1;
...
```

a ≡ &a[0] → t1
i → t1
a[i] → t2

ASM
```
a:  .space 20
...
la   t0, a
slli t1, t1, 2
add  t0, t0, t1
lw   t2, 0(t0)
addi t2, t2, 1
sw   t2, 0(t0)
...
```

Loads the array base address → `la t0, a`

*Calculates the offset i*4* → `slli t1, t1, 2`

Adds base and offset → `add t0, t0, t1`

Loads `a[i]` → `lw t2, 0(t0)`

Stores `a[i]` → `sw t2, 0(t0)`

Variables and constants

Arrays (iii)

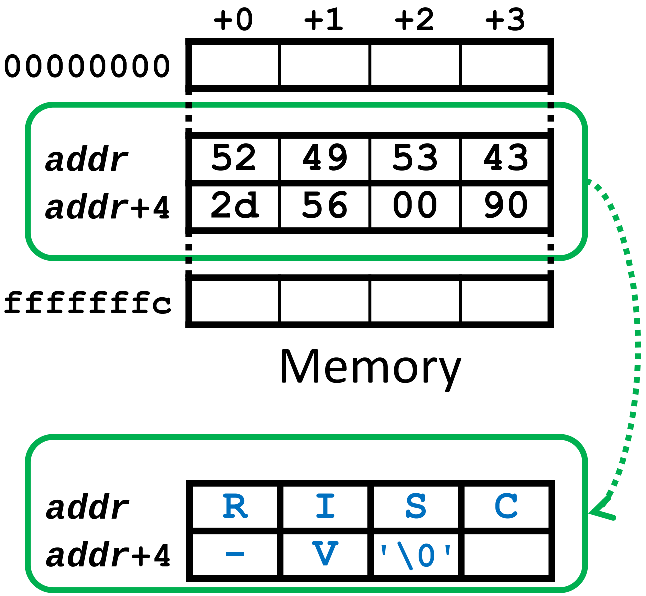
- Character strings are a special case of an array.
  - They store ASCII characters in order.
  - Each ASCII character takes one byte.
  - An array ends with the '\0’ character (0x0), which acts as the end-of-string watchdog (it detects the end of the string).

C/C++

```
const char a[] = "RISC-V";
...
```

ASM

```
a:  .string "RISC-V"
...
```



	+0	+1	+2	+3
00000000				
addr	52	49	53	43
addr+4	2d	56	00	90
fffffffc				

Memory

addr	R	I	S	C
addr+4	-	V	'\0'	

Variables and constants

Structures (i)

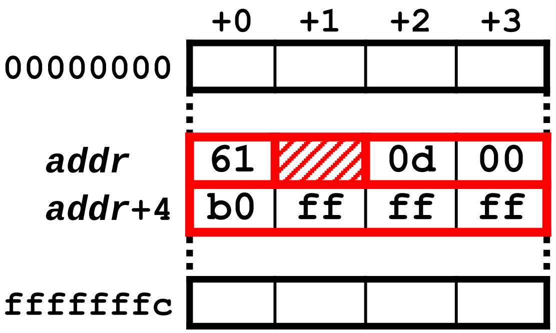
- A structure is a collection of data with different sizes located in consecutive memory addresses
  - In C/C++, each member of a structure is identified by a name.
  - The structure and its members must be aligned according to its size.

C/C++

```
struct baz {
  char c;
  short int si;
  int i;
} a = { 'a', 13, -80 };
...
```

ASM

```
a:
   .align 2
   .byte  'a’
   .align 1
   .half  13
   .word  -80
...
```



	+0	+1	+2	+3
00000000				
addr	61	▨	0d	00
addr+4	b0	ff	ff	ff
fffffffc				

Memory

- To access a structure member, its address has to be calculated:
  - It is the sum of the structure base address plus an offset.
    - The structure base address is the address of its first member.
    - The offset in bytes is calculated depending on the member relative position.

Variables and constants

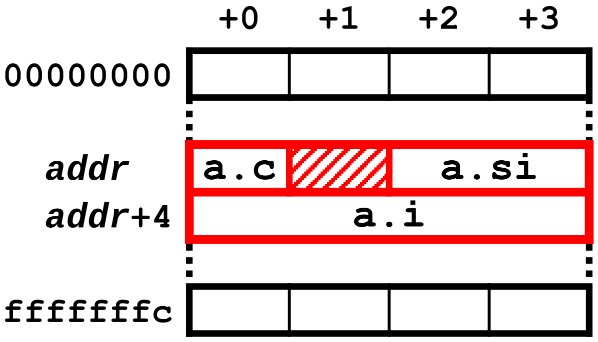
Structures (ii)

- The offset of each member is always a constant and it is calculated by the programmer.

C/C++

```
struct baz { char c; short int si; int i; } a;
...
a.i = a.c + a.si;
...
```

&a → t0
a.c → t1
a.si → t2
a.i → t3



Memory

ASM

```
a:  .space   8
...
la   t0, a
lb   t1, 0(t0)
lh   t2, 2(t0)
add  t3, t1, t2
sw   t0, 4(t0)
...
```

Loads the structure base address

loads `a.c`

loads `a.si`

stores `a.i`

Expressions

- Simple expressions in C/C++ require a single instruction
  - Using registers where operands have been previously loaded

C/C++

```
int a, b, c;
...
a = b + c;
...
```

a → t0
b → t1
c → t2

ASM

```
...
add t0, t1, t2
...
```

- Compound expressions require more than one instruction.
  - Using additional registers to store intermediate results

C/C++

```
int a, b, c, d;
...
a = (b + c) - d;
...
```

a → t0
b → t1
c → t2
d → t3

ASM

```
...
add t4, t1, t2
sub t0, t4, t3
...
```

Expressions

- Explicit constants can appear in a symbolic form:
  - If the constant is short (≤12b), i.e., in the [−2048, +2047] range, it can be used directly, as an immediate operand.

C/C++

```
int a, b;
...
a = b + 5;
...
```

a → t0
b → t1

ASM

```
...
addi t0, t1, 5
...
```

ASM

```
.equ  N, 5
...
addi t0, t1, N
...
```

-
  - Long constants (>12b) must be previously loaded in a register
    - To avoid having to divide the constant explicitly (and correct its upper part in the case that bit 11 is 1), the `li` pseudo-instruction can be used.

ASM

```
...
li   t2, 50000
add t0, t1, t2
...
```

ASM

```
.equ N, 50000
...
li    t2, N
add   t0, t1, t2
...
```

equivalent program

```
...
lui   t2, 0xc
addi t2, t2, 0x350
add   t0, t1, t2
...
```

50000_{10} = $00000c350_{16}$

Expressions

- Constants must also be loaded into registers when the instructions that use them do not accept immediate operands.
  - Programmers do not have to worry about the constant size, the `li` pseudo-instruction can handle this.

C/C++

```
int a, b;
...
a = b * 27;
...
```

a → t0
b → t1

ASM

```
...
li    t2, 27
mul   t0, t1, t2
...
```

equivalent program

```
...
addi  t2, x0, 27
mul   t0, t1, t2
...
```

C/C++

```
int a, b;
...
a = b * 50000;
...
```

a → t0
b → t1

ASM

```
...
li    t2, 50000
mul   t0, t1, t2
...
```

equivalent program

```
...
lui   t2, 0xc
addi  t2, t2, 0x350
add   t0, t1, t2
...
```

50000_{10} = $00000c350_{16}$

Expressions

- Constants may be defined with expressions formed by other constants (explicit or symbolic) or other expressions.
  - During the assembly process, expressions will be reduced to an explicit numeric constant that will be translated into machine code.

C/C++

```
#define N 5
...
int a[N];
...
a[0] = a[1] + a[2];
...
```

a ≡ &a[0] → t0
a[1] → t1
a[2] → t2
a[0] → t3

ASM

```
.equ N,    5
.equ LEN, 4
...
a:  .space N*LEN
...
la   t0, a
lw   t1, 1*LEN(t0)
lw   t2, 2*LEN(t0)
add  t3, t1, t2
sw   t3, 0*LEN(t0)
...
```

equivalent program

```
...
la   t0, ...
lw   t1, 4(t0)
lw   t2, 8(t0)
add  t1, t1, t2
sw   t1, 0(t0)
...
```

Code organization

- In assembly, there are no restrictions about how to organize code:
  - But it is advisable to perform structured and procedural programming.
  - Avoid "spaghetti" programs full of branches that are difficult to understand, debug and maintain.
- Structured programming implies programming in blocks:
  - Each block has a single input point and a single output point
  - A block can be:
    - Linear: code without branches.
    - Conditional (e.g. `if`, `switch`): blocks that are executed depending on the value of a condition.
    - Iterative (e.g. `for`, `while`): blocks that are executed several times depending on the value of a condition.
- Procedural programming implies dividing the code into smaller reusable functions:
  - They use local data and communicate using parameters.

Code organization

Linear blocks

- Formed by a sequence of blocks with no branches.

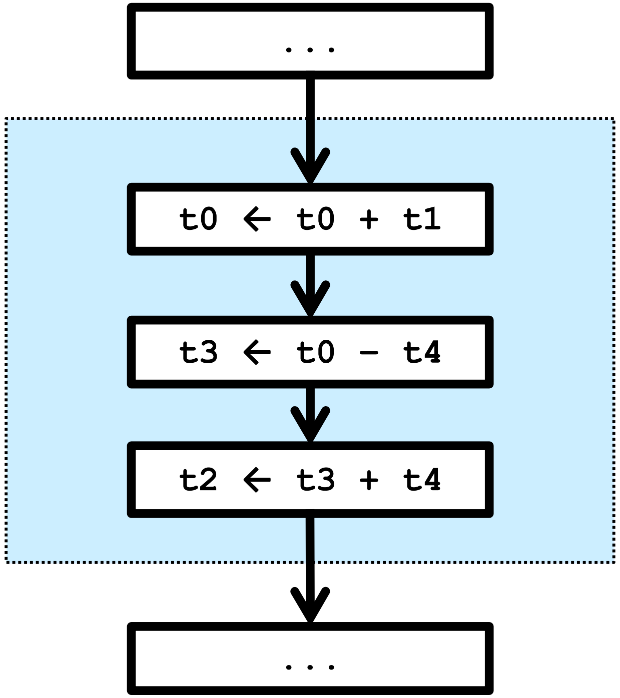
C/C++

```
...
a = a + b;
g = a – h;
f = g + h;
...
```

a → *t0*   *b* → *t1*
f → *t2*   *g* → *t3*   *h* → *t4*

ASM

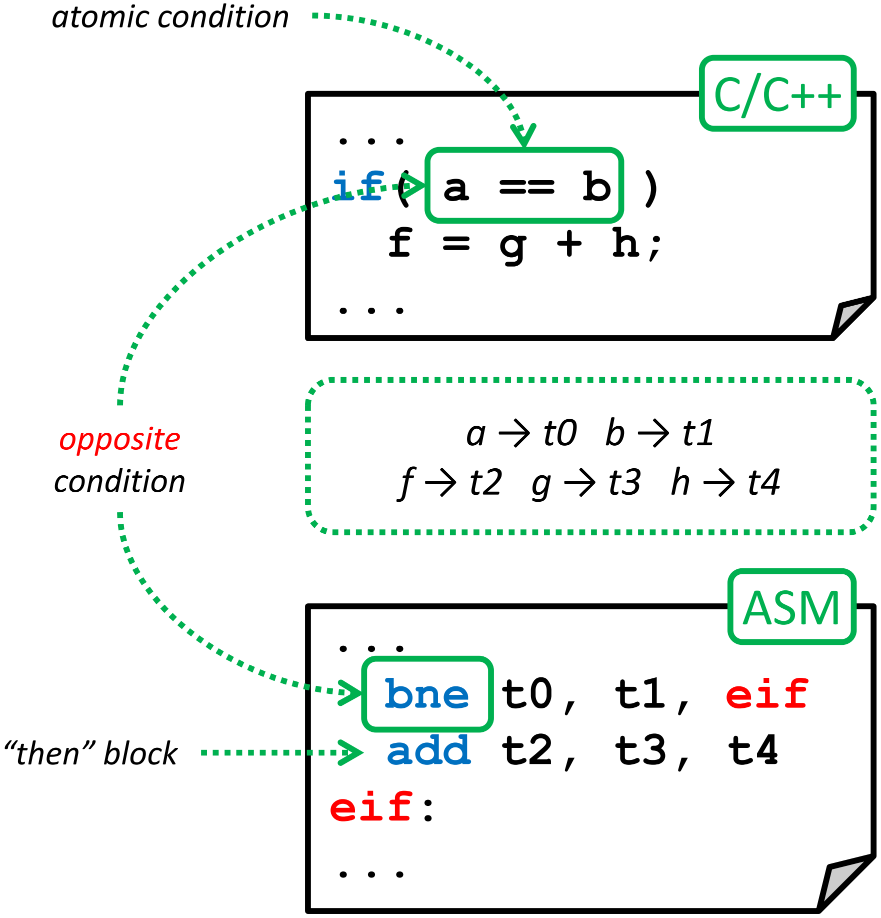
```
...
add t0, t0, t1
sub t3, t0, t4
add t2, t3, t4
...
```



Code organization

Simple conditional *if-then* block (i)

- The block is executed depending on the value of a condition.



atomic condition

C/C++

```
...
if ( a == b )
   f = g + h;
...
```

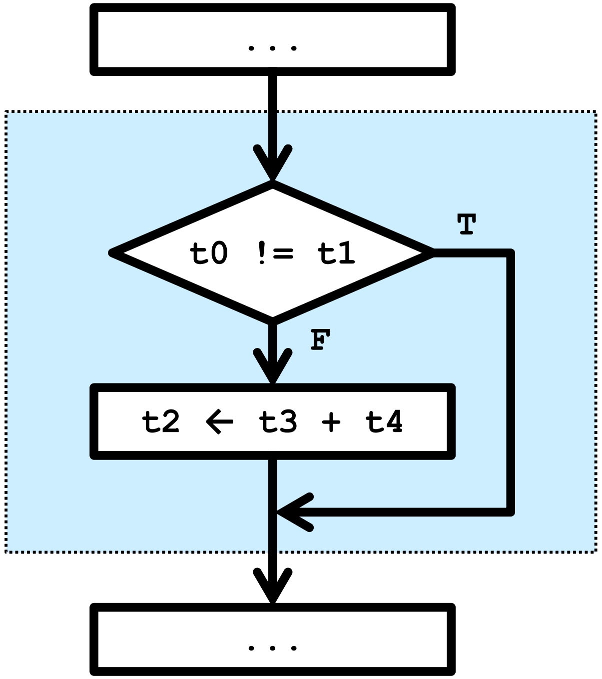
a → t0   b → t1
f → t2   g → t3   h → t4

opposite *condition*

ASM

```
...
 bne t0, t1, eif
 add t2, t3, t4
eif:
...
```

"then" block



Code organization

Simple conditional *if-then* block (ii)

- When there is a composed condition, all the atomic conditions that form the compound condition are checked individually.

conjunction compound condition

C/C++

```
...
if ( a == b && a > 0 )
   f = g + h;
...
```

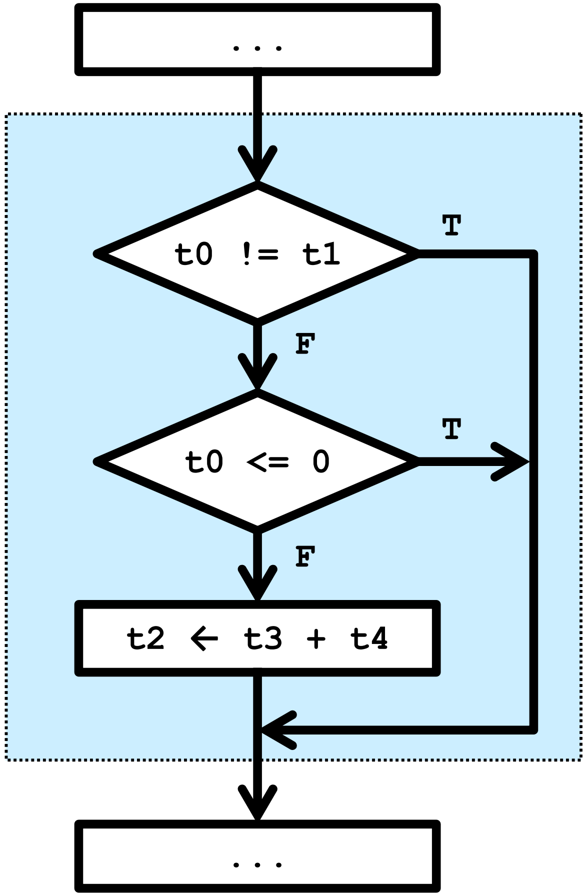
opposite conditions

a → t0   b → t1
f → t2   g → t3   h → t4

ASM

```
...
  bne   t0, t1, eif
  blez  t0, eif
  add   t2, t3, t4
eif:
...
```

"then" block



Code organization

Simple conditional *if-then* block (iii)

- When there is a composed condition, all the atomic conditions that form the compound condition are checked individually.

disjunction compound condition

C/C++

```
...
if ( a == b || a > 0 )
   f = g + h;
...
```

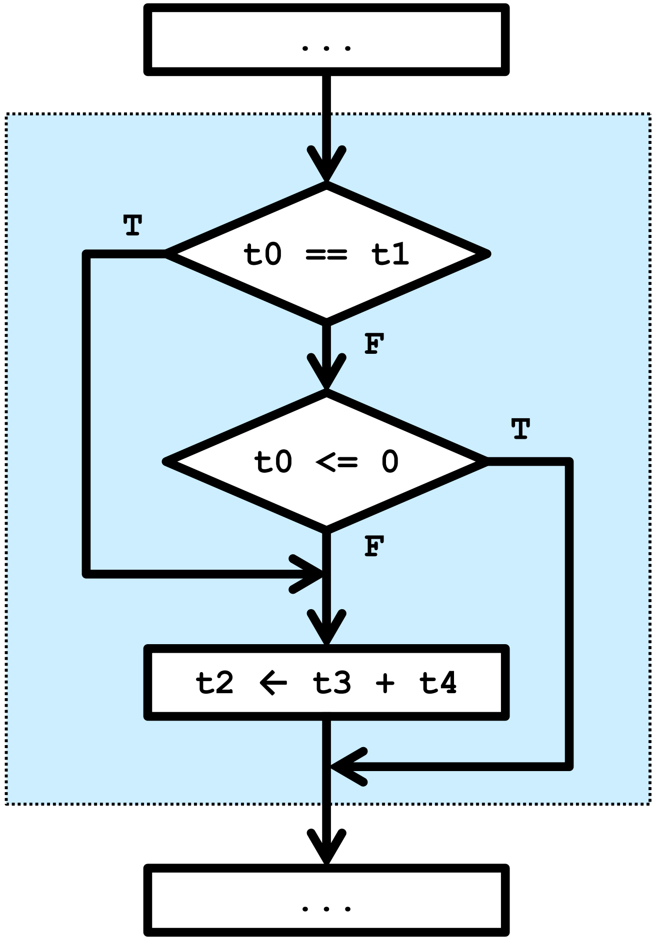
same 1st condition
opposite 2nd condition

a → t0   b → t1
f → t2   g → t3   h → t4

ASM

```
...
   beq  t0, t1, then
   blez t0, eif
then:
   add  t2, t3, t4
eif:
...
```

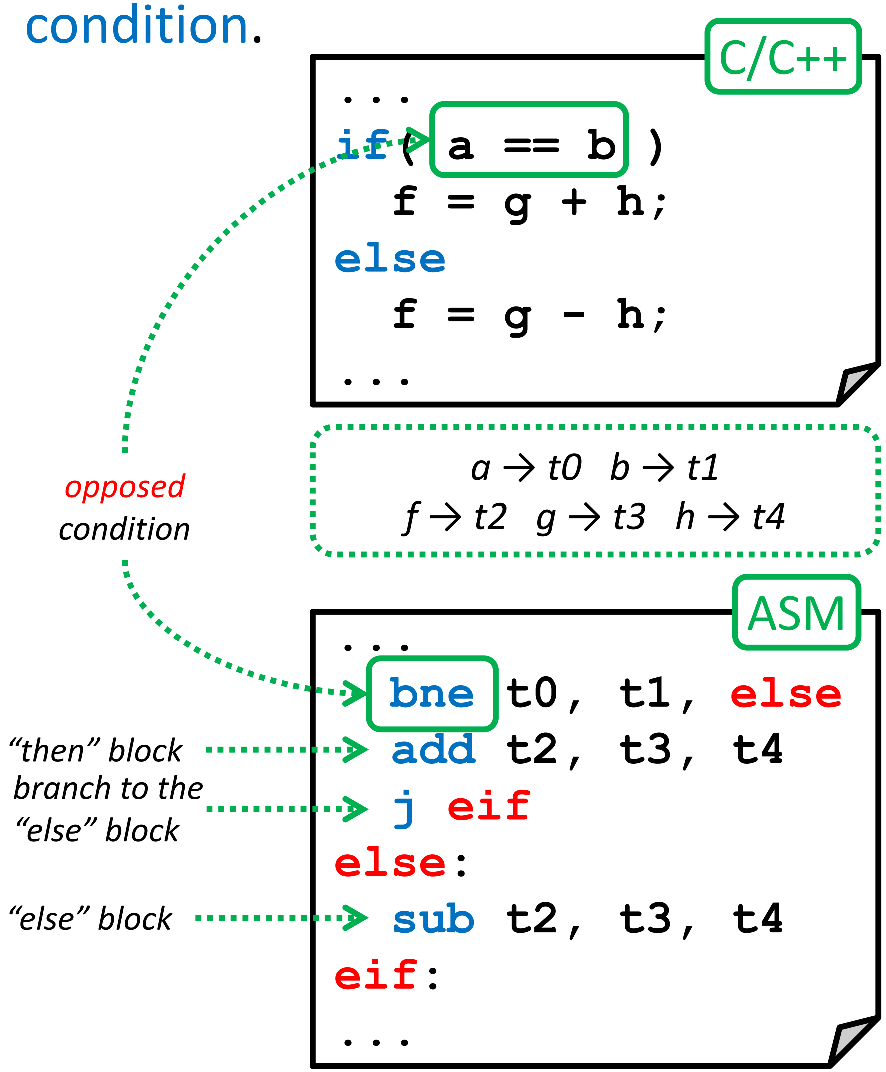
"then" block



Code organization

Double conditional *if-then-else* block

- One of either two blocks is executed, depending on the value of a condition.



C/C++

```
...
if( a == b )
   f = g + h;
else
   f = g - h;
...
```

a → t0   b → t1
f → t2   g → t3   h → t4

opposed condition

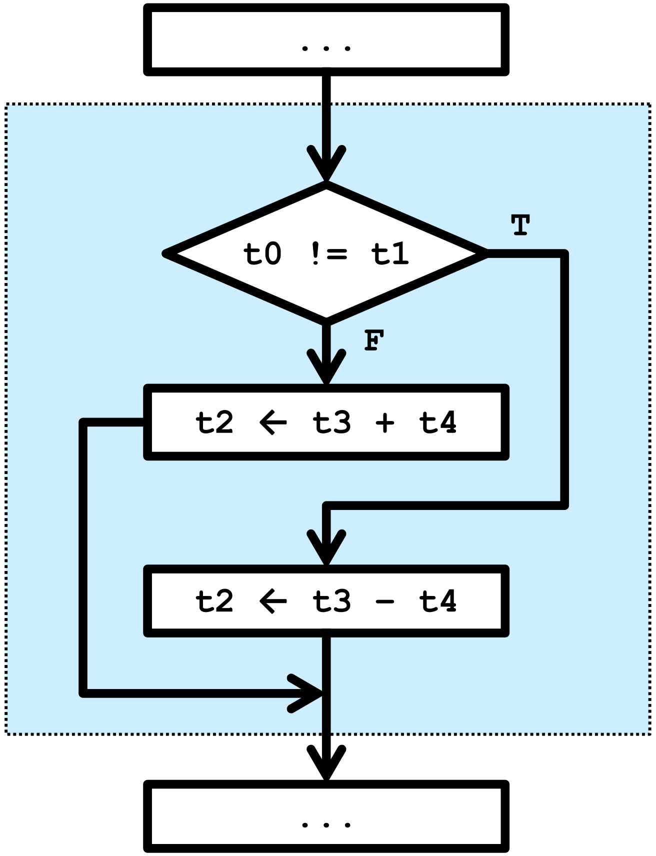
ASM

```
...
bne t0, t1, else
add t2, t3, t4
j eif
else:
sub t2, t3, t4
eif:
...
```

"then" block

branch to the "else" block

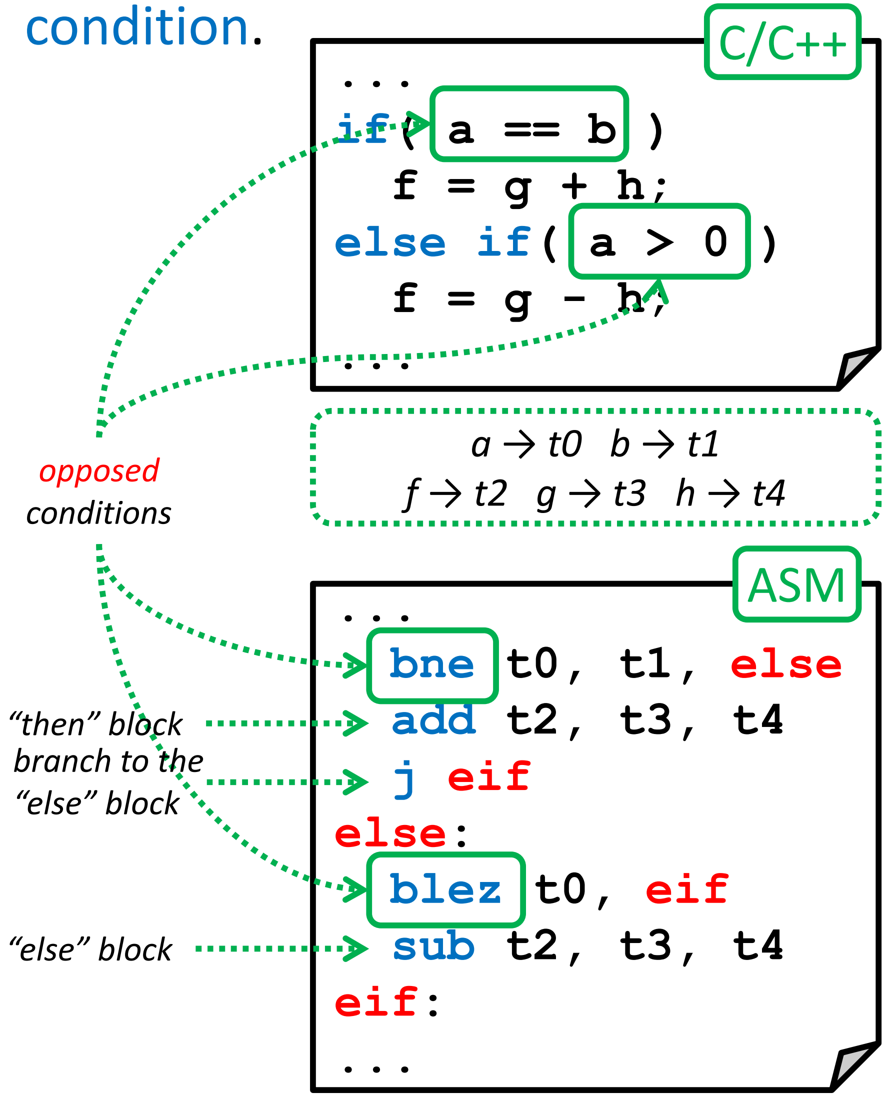
"else" block



Code organization

Double conditional *if-then-else* block

- One of either two blocks is executed, depending on the value of a condition.



C/C++

```
...
if( a == b )
  f = g + h;
else if( a > 0 )
  f = g - h;
...
```

a → t0   b → t1
f → t2   g → t3   h → t4

opposed conditions

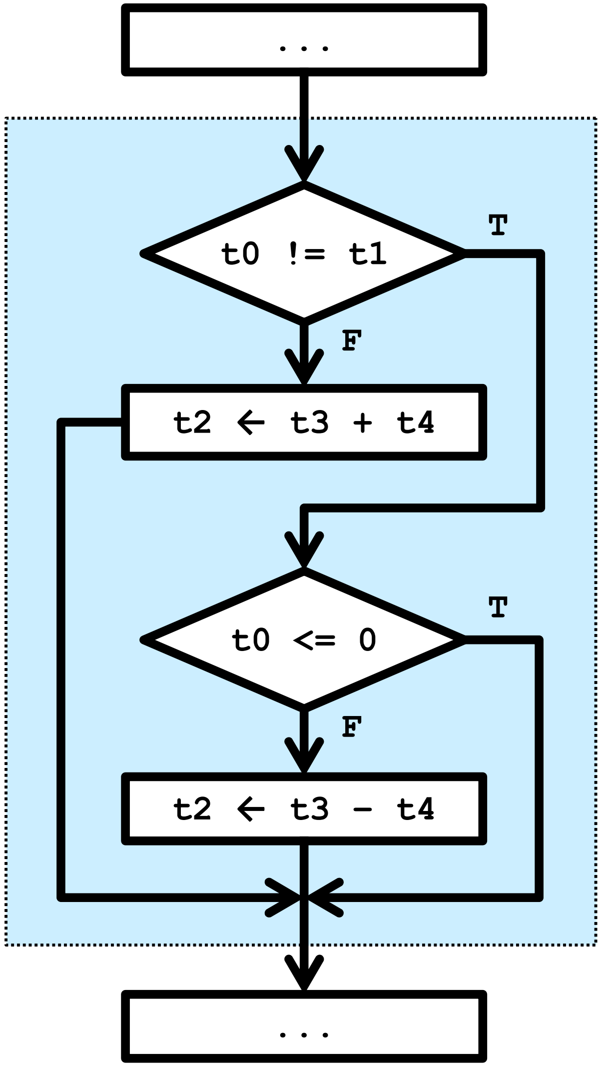
ASM

```
...
bne t0, t1, else
add t2, t3, t4
j eif
else:
blez t0, eif
sub t2, t3, t4
eif:
...
```

"then" block

branch to the "else" block

"else" block



Code organization

Selective *switch* block

- One among several blocks is executed, depending on the value of a variable.

C/C++

```
...
switch( a )
{
   case 0:
     f = g + h;
     break;
   case 1:
     f = g - h;
     break;
   default:
     f = g;
}
...
```

a → t0   f → t2
g → t3   h → t4

ASM

```
switch: .word case0, case1
   ...
   li    t5, 1
   bgt   t0, t5, default
   la    t5, switch
   slli  t6, t0, 2
   add   t5, t5, t6
   lw    t5, 0(t5)
   jr    t5
case0:
   add t2, t3, t4
   j   eswitch
case1:
   sub t2, t3, t4
   j   eswitch
default:
   mv  t2, t3
eswitch:
...
```

Address array with the beginning of each block

Branches to the default block

Loads the array base address

Calculates the offset

Adds base and offset

Loads branch address

Branches to the appropriate block

Code organization

Iterative *while-do* block

- The execution of a block is repeated depending on the value of a condition that is evaluated at the beginning of the block.

C/C++

```
...
while ( a != b )
{
   f = f + a;
   a = a - b;
}
...
```

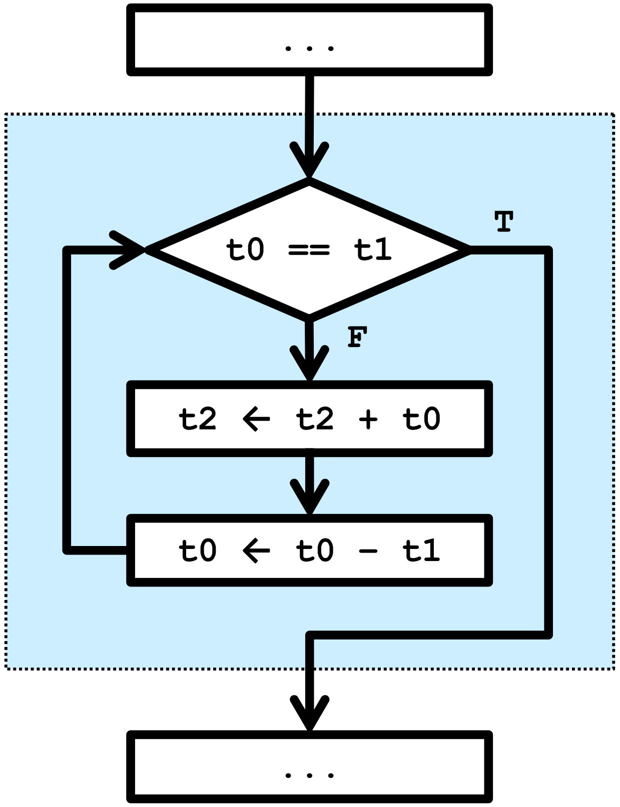
a → t0
b → t1
f → t2

opposed condition

ASM

```
...
while:
  beq t0, t1, ewhile
  add t2, t2, t0
  sub t0, t0, t1
  j     while
ewhile:
...
```

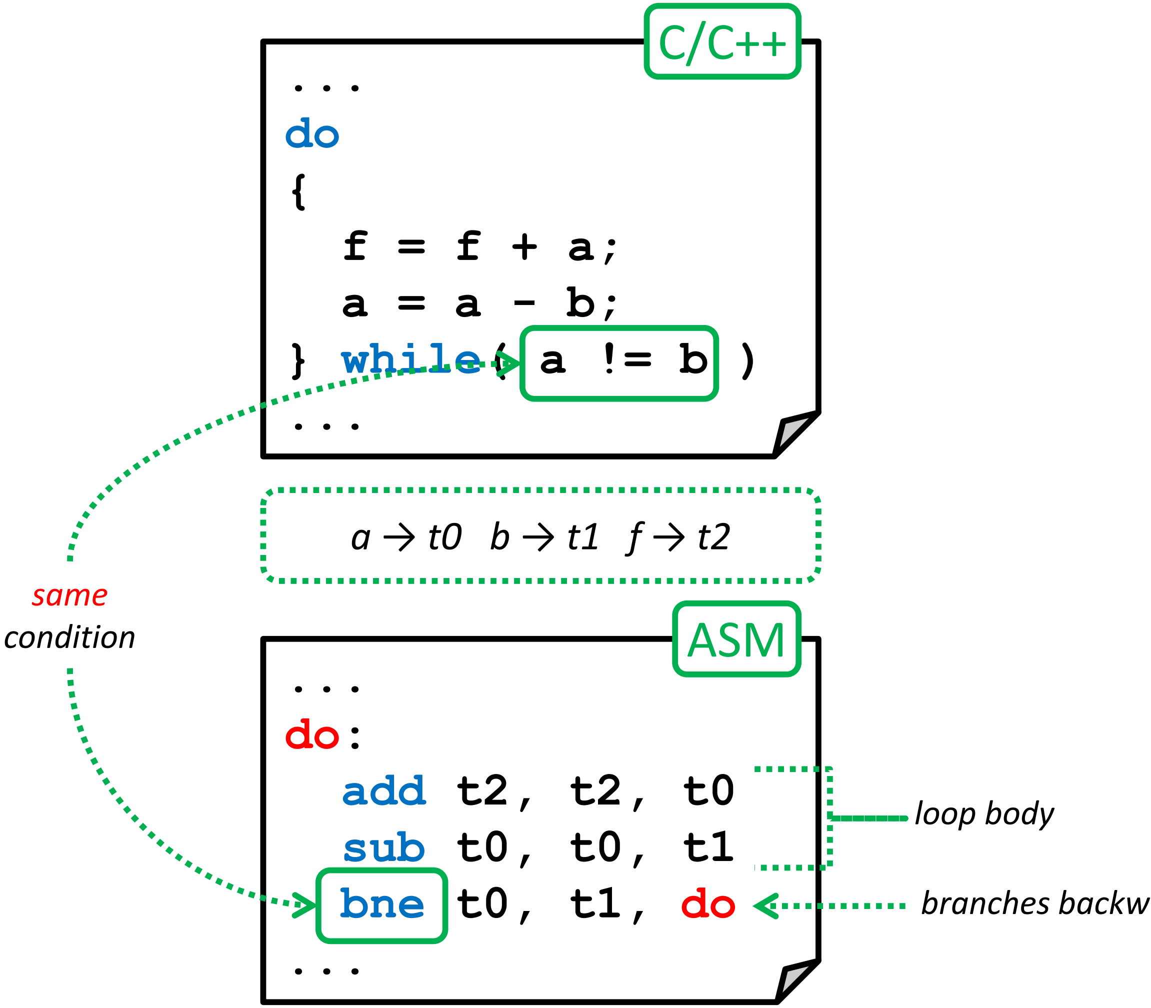
loop body branches backwards to repeat



Code organization

Iterative *while-do* block

- The execution of a block is repeated depending on the value of a condition that is evaluated at the beginning of the block.



C/C++

```
...
do
{
   f = f + a;
   a = a - b;
} while( a != b )
...
```

a → t0   b → t1   f → t2

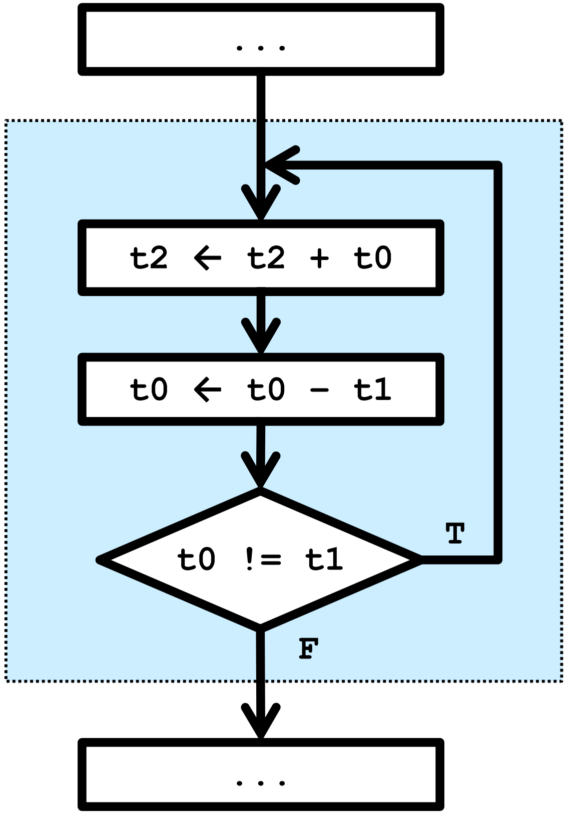
ASM

```
...
do:
   add t2, t2, t0
   sub t0, t0, t1
   bne t0, t1, do
...
```

same condition

loop body

branches backwards to repeat



Code organization

Iterative *for* loop

- The execution of a block is repeated a certain number of times

C/C++

```
...
for( a=0; a<10; a=a+1 )
{
  f = f + b;
}
...
```

a → t0
b → t1
f → t2

ASM

```
...
   mv  t0, x0
   li  t3, 10
for:
   bge t0, t3, efor
   add t2, t2, t1
   add t0, t0, 1
   j   for
efor:
...
```

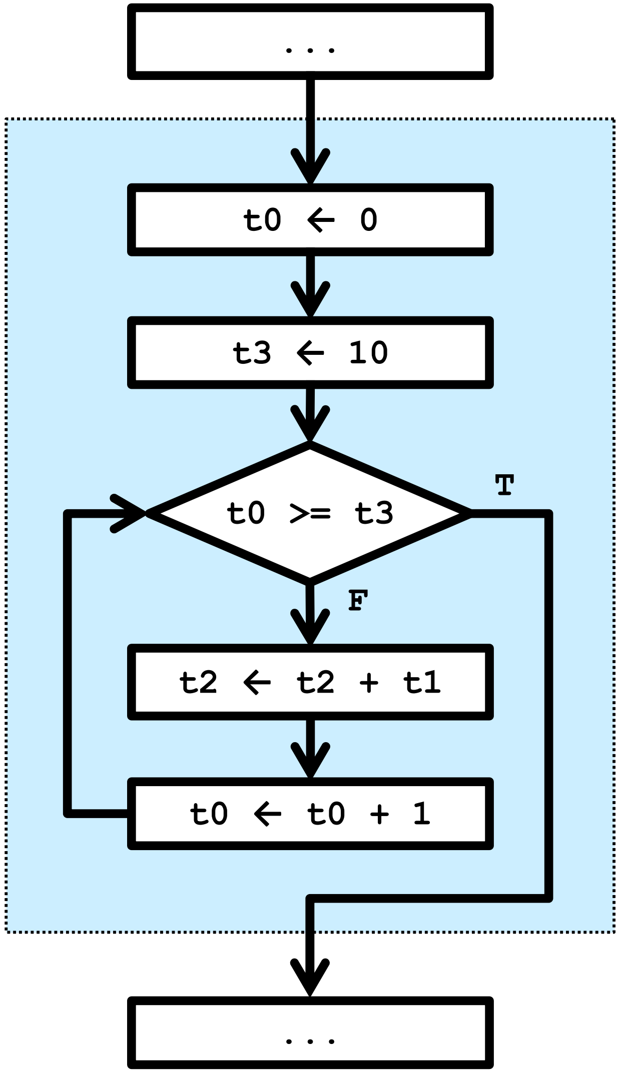
opposed condition

initializes the index

loop body

increments index

branches backwards to repeat



Functions

- Functions* allow reusing pieces of code and make a program more modular and readable.
  - Functions are not declared in assembly.
  - They are identified by its code initial address with a label.
- When a function (caller) calls another one (callee):
  - The caller function must pass the arguments and branch to the beginning of the callee function.
  - The callee function must return the result and branch to the instruction that is located after the one that made the function call.
  - Since registers and memory are accessible by both functions, the callee function should not modify anything that is needed by the caller function.
- In assembler, the argument and branch management is explicit
  - Each architecture defines a standard function call convention that must be followed in order to guarantee interoperability.

(*) Also called procedures, methods or subroutines

Functions

- The RISC-V registers are interchangeable, but in order to facilitate function management in assembly:
  - By convention, each register has a certain purpose, as well as an easy to remember alias.

# Reg.	Alias	Type	Usual purpose
x0	zero	N/A	Constant value 0
x1	ra	preserved	Stores the caller function return address
x2	sp	preserved	Stores the stack pointer address
x3	gp	N/A	Stores the global data region address of a program
x4	tp	N/A	Stores the local data region address of a thread
x5…x7	t0…t2	temporary	General purpose
x8	s0 fp	preserved	General purpose Stores the frame base address of a function
x9	s1	preserved	General purpose
x10…x11	a0…a1	temporary	Return values to the caller function
x12…x17	a2…a7	temporary	Pass arguments to the callee function
x18…x27	s2…s11	preserved	General purpose
x28…x31	t3…t6	temporary	General purpose

Functions

Call and return (i)

- By convention, a caller function in RISC-V assembly use:
  - Registers **a0** ... **a7** to pass up to 8 arguments to the callee function.
  - Register **ra** to store the return address.
  - Instruction **jal/jalr** to call (branch to) the callee function.

C/C++

```c
int a;
...
a = inc( a );
...
int inc( int x )
{
  return x+1;
}
...
```

ASM

```asm
a:  .space 4
   ...
   la   t0, a
   lw   a0, 0(t0)
   jal  ra, inc
   sw   a0, 0(t0)
   ...
inc:
   add  a0, a0, 1
   jalr x0, ra, 0
   ...
```

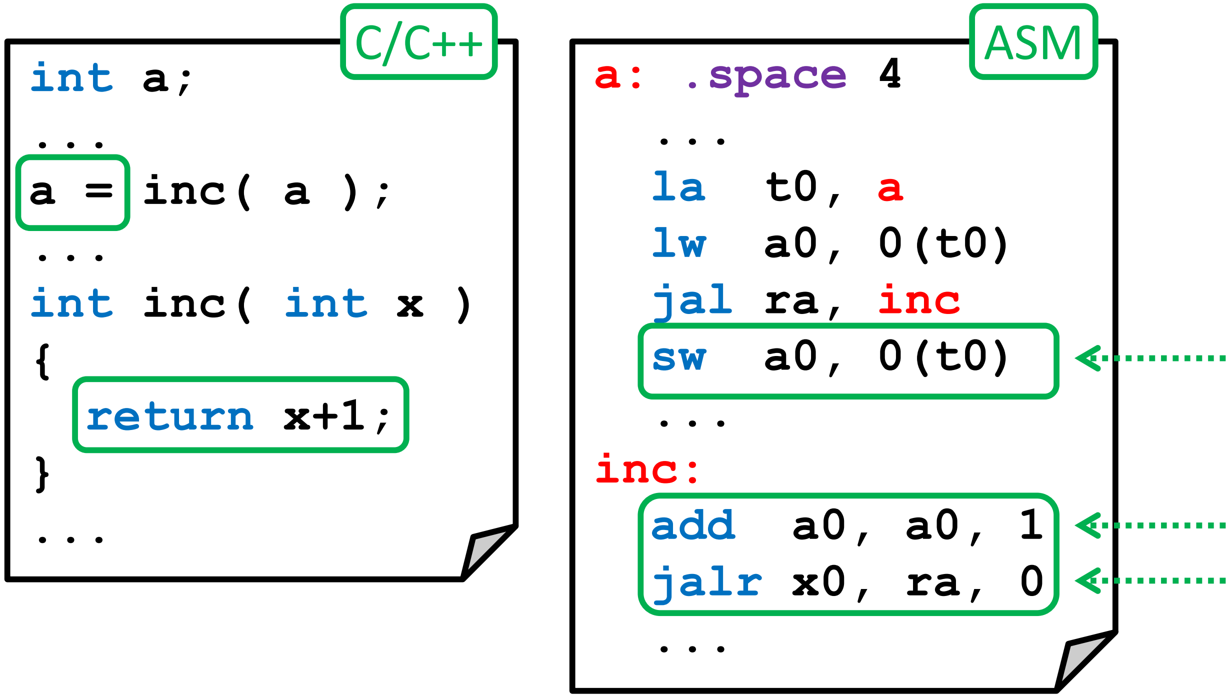
loads argument in a0

branches to the callee function, saving the return address in `ra`

Functions

Call and return (ii)

- By convention, a callee function in RISC-V assembly use:
  - Register `a0` to return the result to the caller function (if the return data is 64b, then `a1` will also be used for the upper part).
  - Instruction `jalr` to return to the caller function.



C/C++

```
int a;
...
a = inc( a );
...
int inc( int x )
{
   return x+1;
}
...
```

ASM

```
a:  .space 4
    ...
    la   t0, a
    lw   a0, 0(t0)
    jal  ra, inc
    sw   a0, 0(t0)
    ...
inc:
    add   a0, a0, 1
    jalr  x0, ra, 0
    ...
```

The caller function stores the returned result in `a0`

Stores the result into `a0`

Returns to the caller function

Functions

Call and return (iii)

- Since branching/returning to/from functions is very common:
  - The caller function can use the `call` pseudo-instruction
  - The callee function can use the `ret` pseudo-instruction

C/C++

```
int a;
...
a = inc( a );
...
int inc( int x )
{
  return x+1;
}
...
```

ASM

```
a:  .space 4
   ...
   la    t0, a
   lw    a0, 0(t0)
   call  inc
   sw    a0, 0(t0)
   ...
inc:
   add   a0, a0, 1
   ret
   ...
```

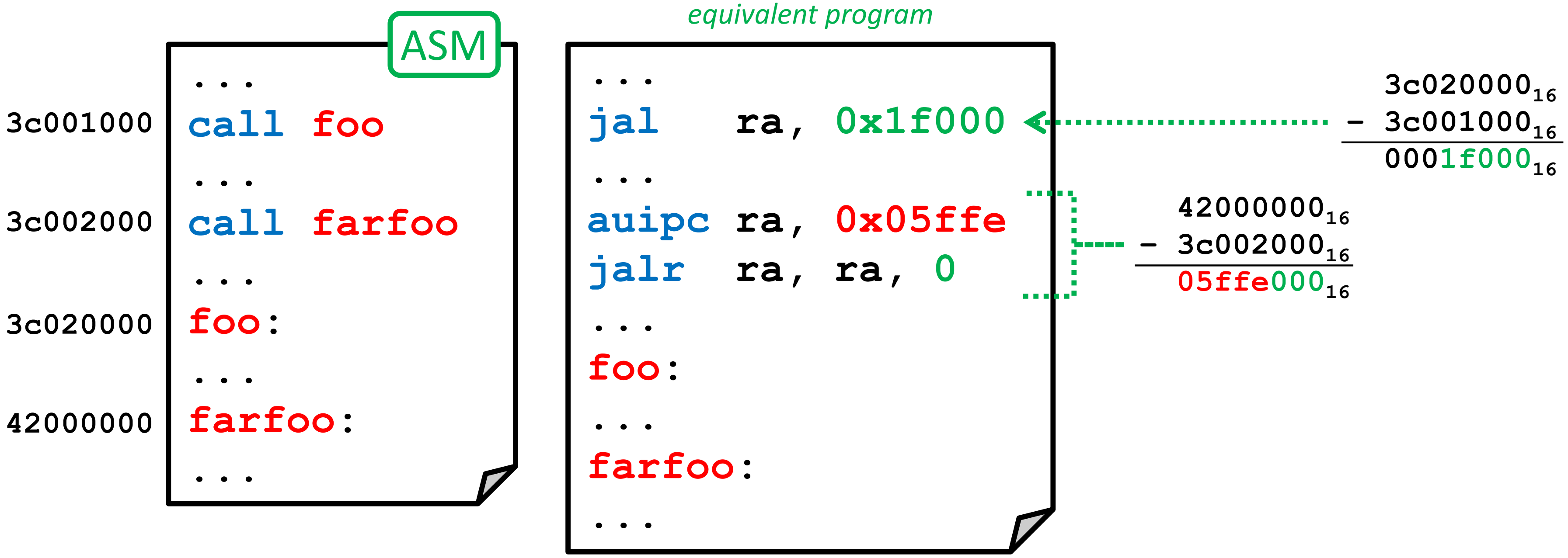
equivalent program

```
a:  .space 4
   ...
   la   t0, a
   lw   a0, 0(t0)
   jal  ra, inc
   sw   a0, 0(t0)
   ...
inc:
   addi a0, a0, 1
   jalr x0, ra, 0
   ...
```

Functions

Call and return (iv)

- The **`call`** pseudo-instruction is translated in the assembly process:
  - To a **`jal`** instruction, if the function is within a ±1MiB range.
    - The offset of this instruction is C2 21b PC-relative.
  - To the **`auipc+jalr`** instructions, if the function is located in a further range.
  - This exempts the programmer from knowing the location of the callee function.



ASM

```
         ...
3c001000 call  foo
         ...
3c002000 call  farfoo
         ...
3c020000 foo:
         ...
42000000 farfoo:
         ...
```

equivalent program

```
...
jal    ra, 0x1f000
...
auipc ra, 0x05ffe
jalr  ra, ra, 0
...
foo:
...
farfoo:
...
```

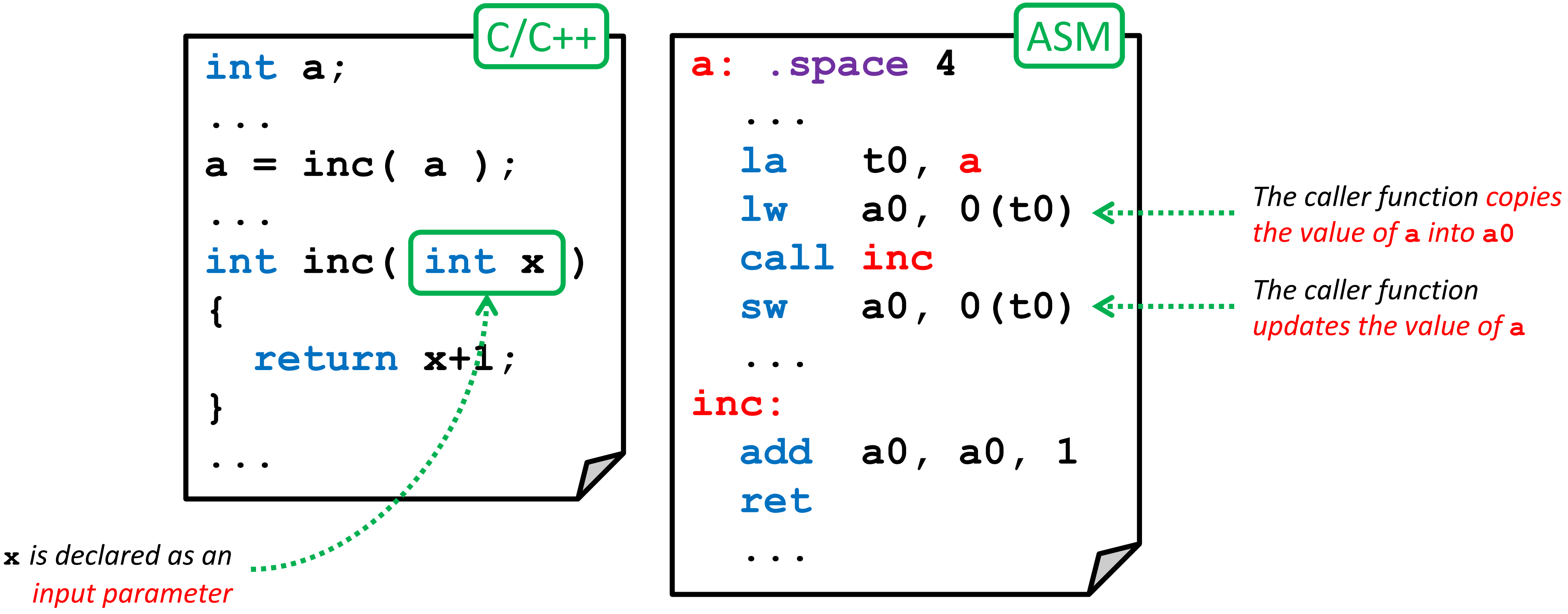
$$\begin{array}{r} 3c020000_{16} \\ -\ 3c001000_{16} \\ \hline 0001f000_{16} \end{array}$$

$$\begin{array}{r} 42000000_{16} \\ -\ 3c002000_{16} \\ \hline 05ffe000_{16} \end{array}$$

Functions

Parameters by value vs. reference (i)

- The input parameters are passed by value to the callee function:
  - The caller function passes, as an argument, a copy of the variable’s value to the callee function.
  - If that value has to be updated, it is done by the caller function.



```
int a;
...
a = inc( a );
...
int inc( int x )
{
   return x+1;
}
...
```

```
a:  .space 4
   ...
   la    t0, a
   lw    a0, 0(t0)
   call  inc
   sw    a0, 0(t0)
   ...
inc:
   add   a0, a0, 1
   ret
   ...
```

`x` is declared as an input parameter

The caller function copies the value of `a` into `a0`

The caller function updates the value of `a`

Functions

Parameters by value vs. reference (ii)

- The output parameters are passed by reference to the callee function:
  - The caller function passes, as an argument, the address of the variable that has to be updated to the callee function.
  - The update of the variable's value is done by the callee function.

C++

```
int a;
...
inc( a );
...
void inc( int & x )
{
   x = x + 1;
}
...
```

`x` *is declared as an output parameter*

ASM

```
a:  .space 4
    ...
    la    a0, a
    call  inc
    ...
inc:
    lw    t0, 0(a0)
    add   t0, t0, 1
    sw    t0, 0(a0)
    ret
    ...
```

The caller function copies the address of `a` *into* `a0`

The callee function updates the value of `a`

Functions

Parameters by value vs. reference (iii)

- The concept of C/C++ pointer is an abstraction of the address of a variable.
  - C output parameters are pointer-type arguments

C++

```
int a;
...
inc( a );
...
void inc( int & x )
{
  x = x + 1;
}
```

C

```
int a;
...
inc( &a );
...
void inc( int *x )
{
  *x = *x + 1;
}
```

ASM

```
a:  .space 4
   ...
   la    a0, a
   call  inc
   ...
inc:
   lw    t0, 0(a0)
   add   t0, t0, 1
   sw    t0, 0(a0)
   ret
```

Functions

Parameters by value vs. reference (iv)

- Arrays are passed by reference to the callee function

C/C++

```
int a[10];
...
incArray( a, 10 );
...
void incArray( int x[], int n )
{
   int i;

   for( i=0; i<n; i=i+1 )
      x[i] = x[i] + 1;
}
```

C/C++

```
int a[10];
...
incArray( a, 10 );
...
void incArray( int *x, int n )
{
   int i;

   for( i=0; i<n; i=i+1 )
      x[i] = x[i] + 1;
}
```

ASM

```
a: .space 4*10
   ...
   la   a0, a
   li   a1, 10
   call incArray
   ...
incArray:
   mv  t0, zero
for:
   bge t0, a1, efor
   sll t1, t0, 2
   add t1, a0, t1
   lw  t2, 0(t1)
   add t2, t2, 1
   sw  t2, 0(t1)
   add t0, t0, 1
   j   for
efor:
   ret
```

Functions

Temporary vs. preserved registers (i)

- The callee function may use the same registers that are used by the caller function.
  - If the callee function changes a register used by the caller function, the program will fail.

C/C++

```
int a;
...
a = inc( a );
...
int inc( int x )
{
  return x+1;
}
...
```

INCORRECT

ASM

```
a:  .space 4
    ...
    la    t0, a
    lw    a0, 0(t0)
    call  inc
    sw    a0, 0(t0)
    ...
inc:
    add   t0, a0, 1
    mv    a0, t0
    ret
    ...
```

The caller function uses `t0` *to store the address of* `a` *temporarily*

But the value of `t0` *has changed after the call and now it does not contain the address of* `a`

The callee function uses `t0` *to store the calculation result temporarily*

Functions

Temporary vs. preserved registers (ii)

- Registers are classified into preserved and temporary, by convention.
- Preserved registers (*callee-saved*): The programmer must guarantee that its content does not change after calling a function.
  - Its value after returning from the callee function must be the same that the one it had before.
  - Therefore, whether the register is not updated in the callee function, or the callee function saves the register at the beginning and restores it at the end.
  - The preserved registers are: **`s1` … `s11`, `sp`, `ra`, `s0/fp`**
- Temporary registers (*caller-saved*): Their content may be updated without restrictions in a function call.
  - Its value after returning from the callee function may be different from the one it had before.
  - If the caller function needs to keep its value, it must be saved before calling the function and restored after returning.
  - The temporary registers are: **`t0` … `t6`, `a0` … `a7`**

Functions

Temporary vs. preserved registers (iii)

- According to this convention, the correct approach would be that the caller function uses preserved registers when a data has to be kept.
  - The callee function may use temporary registers, but if it uses preserved registers, these will have to be saved and restored accordingly.

C/C++

```c
int a;
...
a = inc( a );
...
int inc( int x )
{
  return x+1;
}
...
```

CORRECT

ASM

```
a:  .space 4
   ...
   la    s0, a
   lw    a0, 0(s0)
   call  inc
   sw    a0, 0(s0)
   ...
inc:
   add   t0, a0, 1
   mv    a0, t0
   ret
   ...
```

The caller function uses `s0` *to store the address of* `a` *temporarily*

If `inc` *is correctly programmed,* `s0` *will still contain the address of* `a`

The callee function does not use `s0` *and therefore its value will not change during its execution*

Functions

Stack management (i)

- The stack is a memory region in which data can be stored temporarily without knowing the effective address where they are located:
  - Registers that must be preserved.
  - Arguments of a function (when there are more than 8).
  - Local variables of a function when there are not enough registers.

- Two operations can be performed on a stack:
  - Push: stores a data on the top of the stack.
  - Pop: recovers a data placed at the top of the stack.
  - Stacks work as LIFO structures (Last-in First-out): data are written following a certain order and are read in the reverse order.

- In the RISC-V assembly, by convention:
  - The stack is descending: it grows towards lower memory positions.
  - The **`sp`** register is used to keep the address of the top of the stack, which always contains the last pushed data.

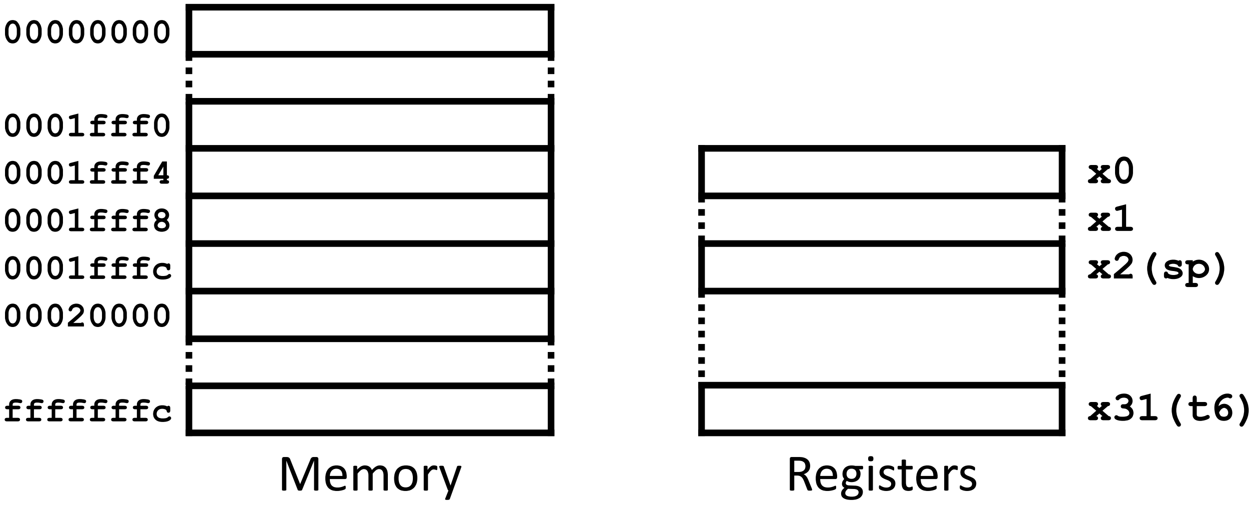
Functions

Stack management (ii)

- Pushing a data implies:
  - Decrementing **`sp`** (the number of bytes of the data, usually 4).
  - Storing the data on the top of the stack.

ASM

```
...
li   sp, 0x20000
...
li   t6, 0x1234
add  sp, sp, -4
sw   t6, 0(sp)
mv   t6, zero
...
lw   t6, 0(sp)
add  sp, sp, 4
...
li   t6, 0x9abc
add  sp, sp, -4
sw   t6, 0(sp)
...
```



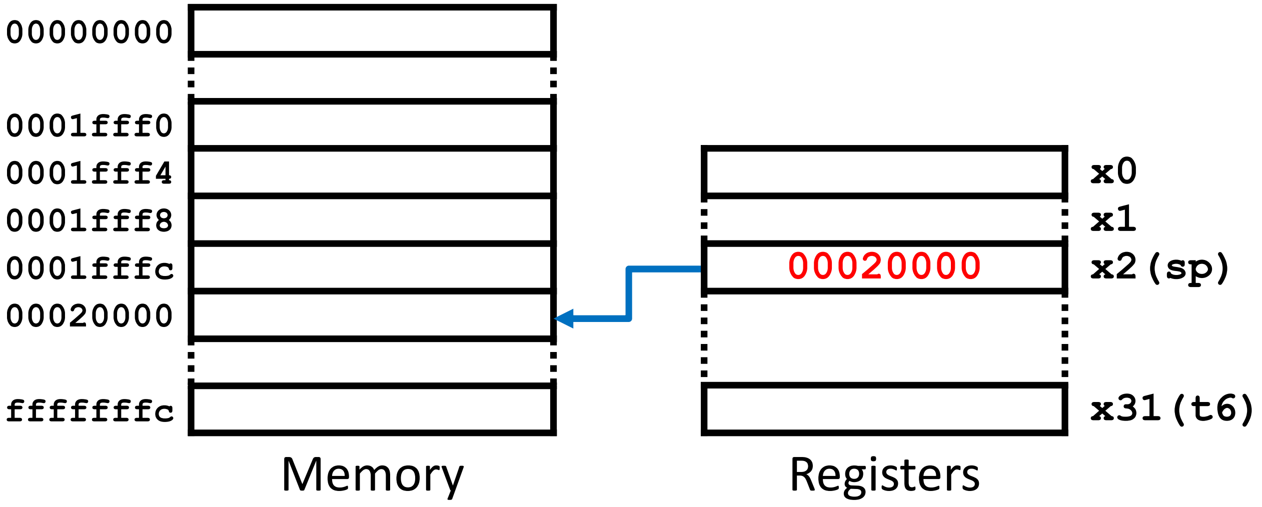
Functions

Stack management (ii)

- Pushing a data implies:
  - Decrementing **`sp`** (the number of bytes of the data, usually 4).
  - Storing the data on the top of the stack.

ASM

```
...
li   sp, 0x20000
...
li   t6, 0x1234
add  sp, sp, -4
sw   t6, 0(sp)
mv   t6, zero
...
lw   t6, 0(sp)
add  sp, sp, 4
...
li   t6, 0x9abc
add  sp, sp, -4
sw   t6, 0(sp)
...
```



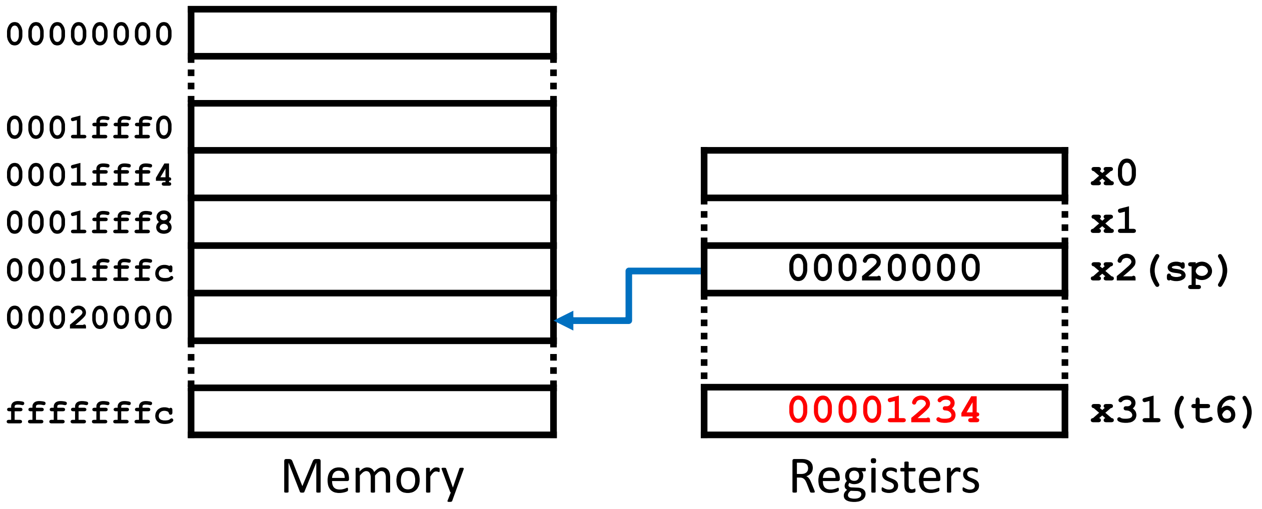
Functions

Stack management (ii)

- Pushing a data implies:
  - Decrementing **`sp`** (the number of bytes of the data, usually 4).
  - Storing the data on the top of the stack.

ASM

```
...
li   sp, 0x20000
...
li   t6, 0x1234
add  sp, sp, -4
sw   t6, 0(sp)
mv   t6, zero
...
lw   t6, 0(sp)
add  sp, sp, 4
...
li   t6, 0x9abc
add  sp, sp, -4
sw   t6, 0(sp)
...
```



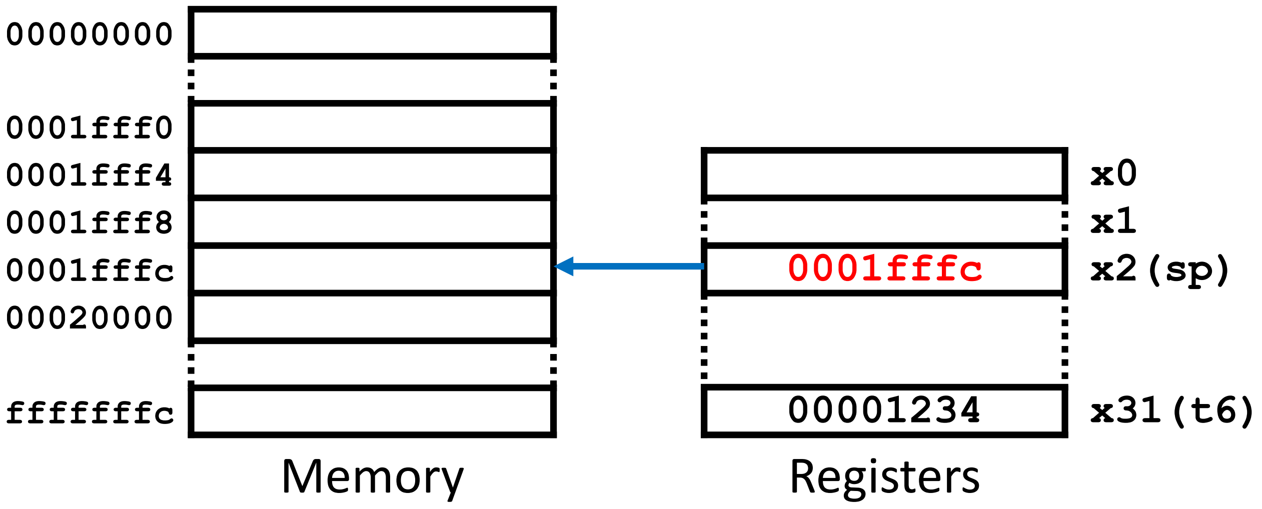
Functions

Stack management (ii)

- Pushing a data implies:
  - Decrementing **`sp`** (the number of bytes of the data, usually 4).
  - Storing the data on the top of the stack.

ASM

```
...
li   sp, 0x20000
...
li   t6, 0x1234
add  sp, sp, -4
sw   t6, 0(sp)
mv   t6, zero
...
lw   t6, 0(sp)
add  sp, sp, 4
...
li   t6, 0x9abc
add  sp, sp, -4
sw   t6, 0(sp)
...
```



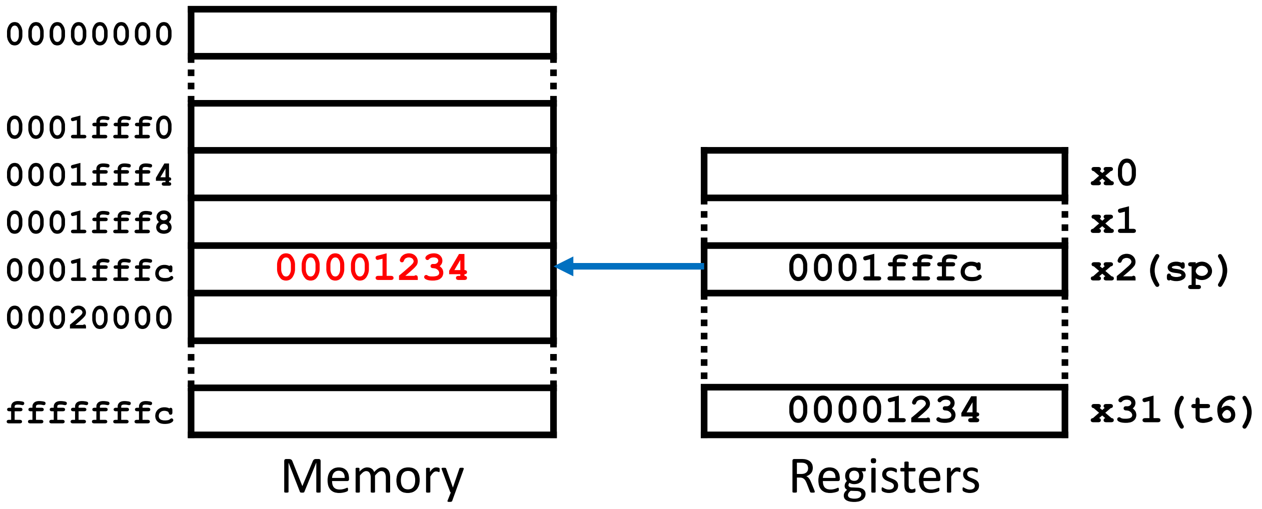
Functions

Stack management (ii)

- Pushing a data implies:
  - Decrementing **`sp`** (the number of bytes of the data, usually 4).
  - Storing the data on the top of the stack.

ASM

```
...
li   sp, 0x20000
...
li   t6, 0x1234
add  sp, sp, -4
sw   t6, 0(sp)
mv   t6, zero
...
lw   t6, 0(sp)
add  sp, sp, 4
...
li   t6, 0x9abc
add  sp, sp, -4
sw   t6, 0(sp)
...
```



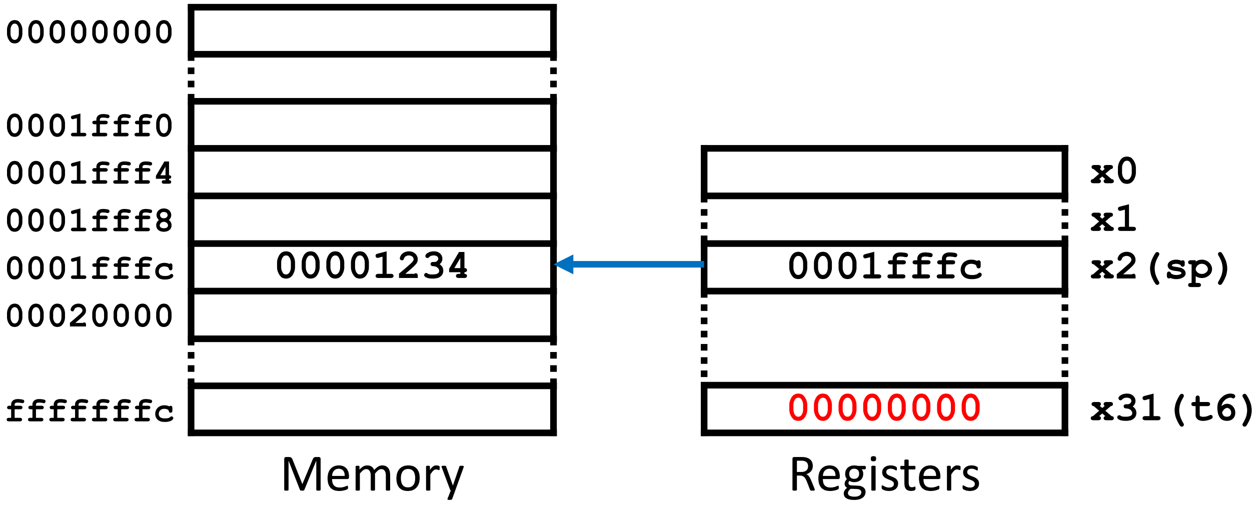
Functions

Stack management (ii)

- Pushing a data implies:
  - Decrementing **sp** (the number of bytes of the data, usually 4).
  - Storing the data on the top of the stack.

ASM

```
...
li   sp, 0x20000
...
li   t6, 0x1234
add  sp, sp, -4
sw   t6, 0(sp)
mv   t6, zero
...
lw   t6, 0(sp)
add  sp, sp, 4
...
li   t6, 0x9abc
add  sp, sp, -4
sw   t6, 0(sp)
...
```



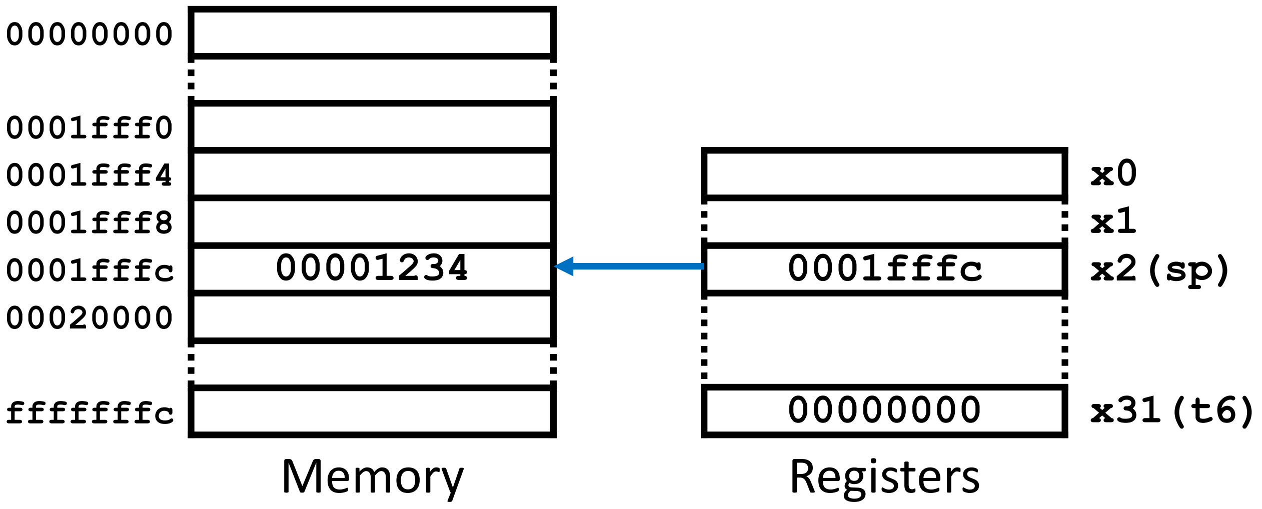
Functions

Stack management (iii)

- Popping a data implies:
  - Loading the data placed at the top of the stack.
  - Incrementing **`sp`** (the number of bytes of the data, usually 4).

ASM

```
...
li   sp, 0x20000
...
li   t6, 0x1234
add  sp, sp, -4
sw   t6, 0(sp)
mv   t6, zero
...
lw   t6, 0(sp)
add  sp, sp, 4
...
li   t6, 0x9abc
add  sp, sp, -4
sw   t6, 0(sp)
...
```



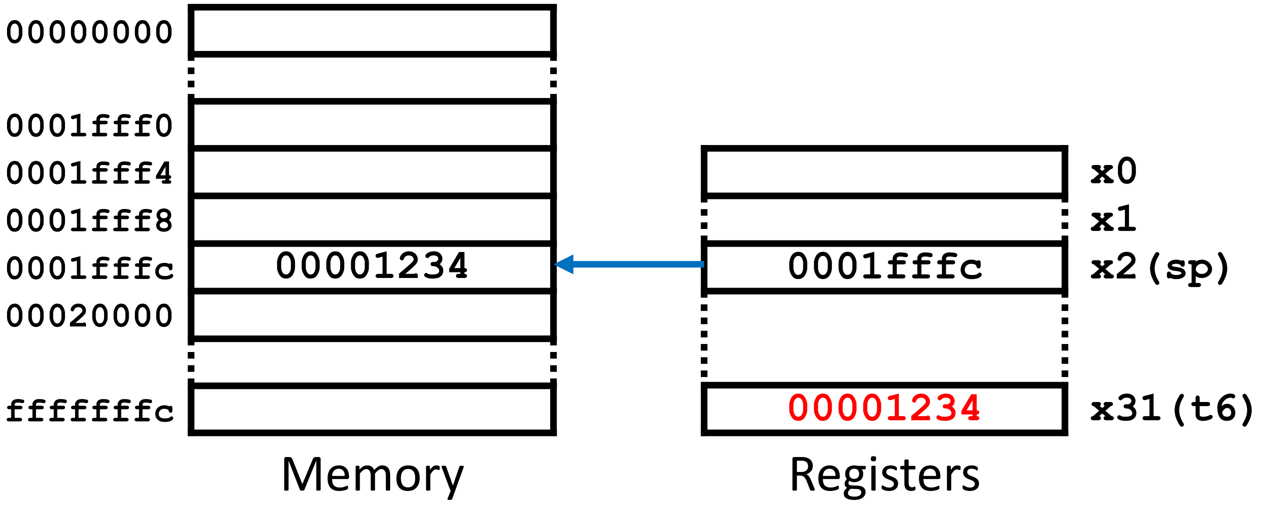
Functions

Stack management (iii)

- Popping a data implies:
  - Loading the data placed at the top of the stack.
  - Incrementing `sp` (the number of bytes of the data, usually 4).

ASM

```
...
li   sp, 0x20000
...
li   t6, 0x1234
add  sp, sp, -4
sw   t6, 0(sp)
mv   t6, zero
...
lw   t6, 0(sp)
add  sp, sp, 4
...
li   t6, 0x9abc
add  sp, sp, -4
sw   t6, 0(sp)
...
```



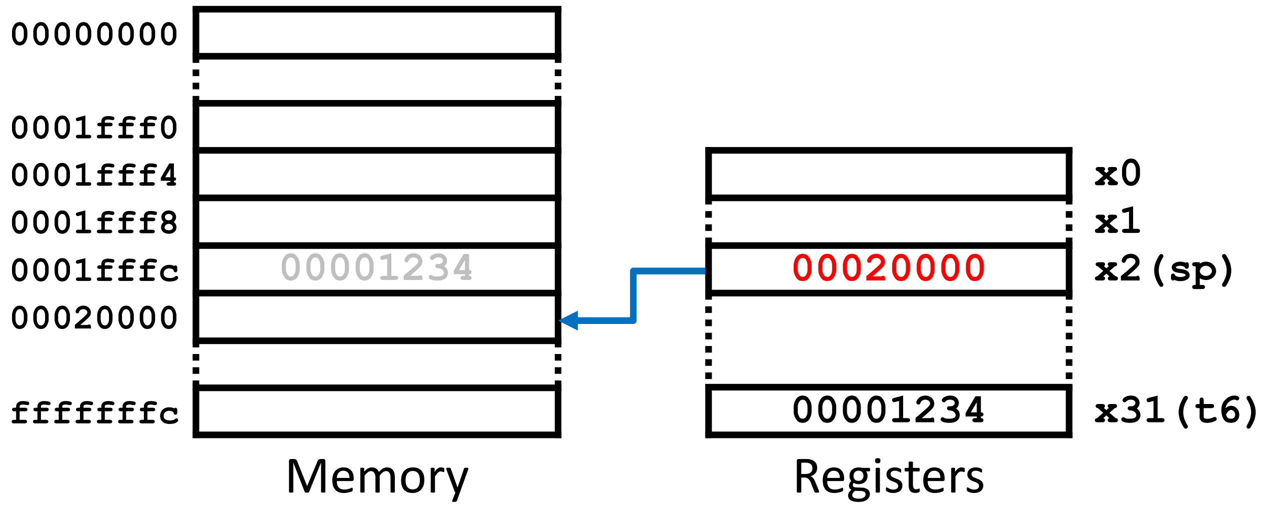
Functions

Stack management (iii)

- Popping a data implies:
  - Loading the data placed at the top of the stack.
  - Incrementing **`sp`** (the number of bytes of the data, usually 4).

ASM

```
...
li   sp, 0x20000
...
li   t6, 0x1234
add  sp, sp, -4
sw   t6, 0(sp)
mv   t6, zero
...
lw   t6, 0(sp)
add  sp, sp, 4
...
li   t6, 0x9abc
add  sp, sp, -4
sw   t6, 0(sp)
...
```



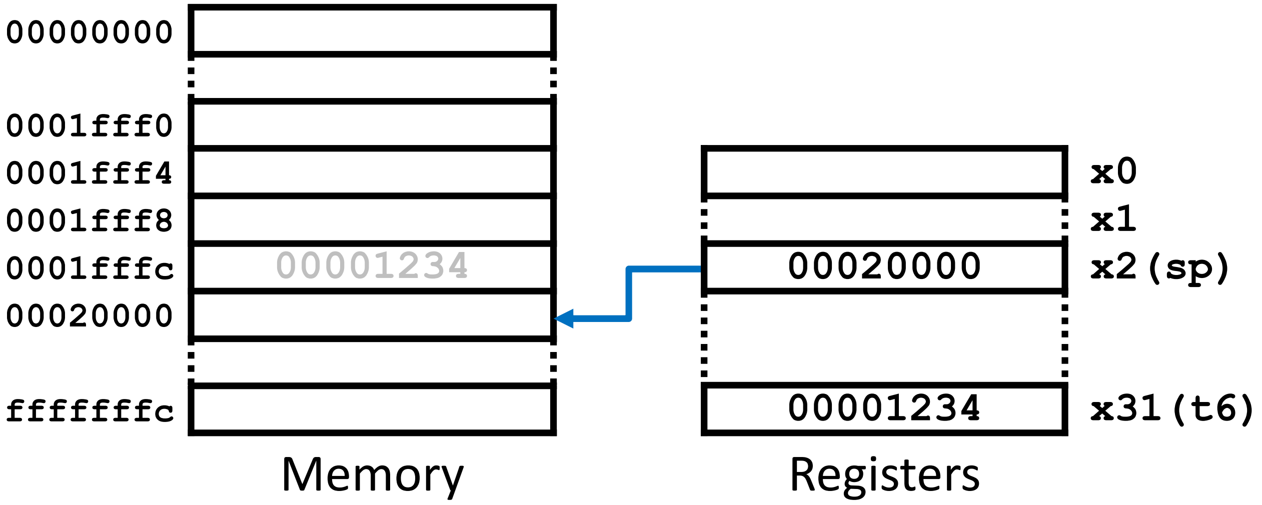
Functions

Stack management (iv)

- The popped data remain in the memory
  - But they should not be used anymore because they will be overwritten by the next pushed data.

ASM

```
...
li   sp, 0x20000
...
li   t6, 0x1234
add  sp, sp, -4
sw   t6, 0(sp)
mv   t6, zero
...
lw   t6, 0(sp)
add  sp, sp, 4
...
li   t6, 0x9abc
add  sp, sp, -4
sw   t6, 0(sp)
...
```



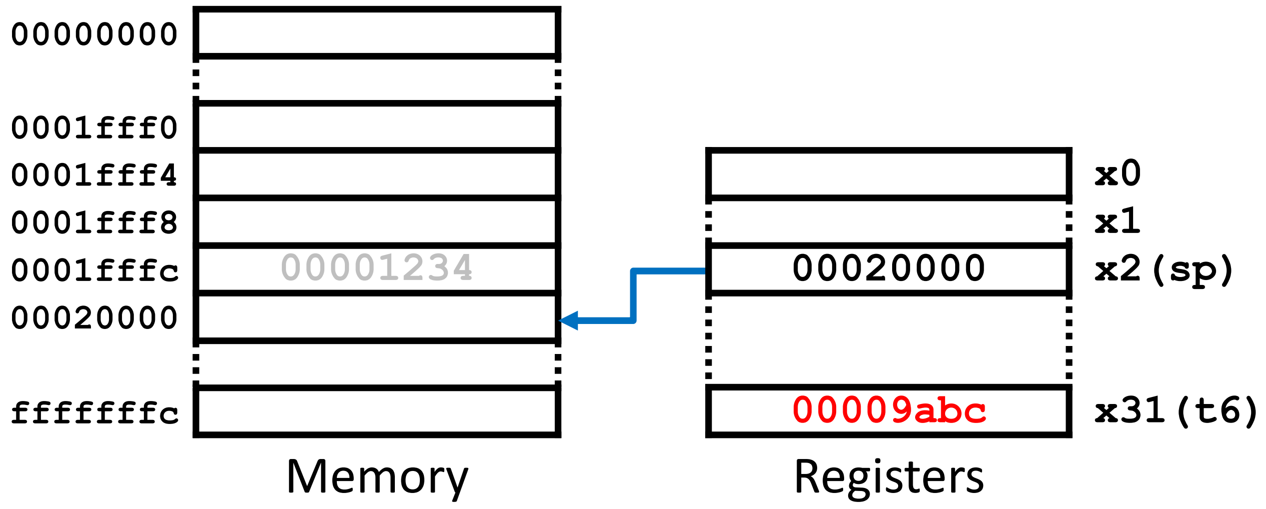
Functions

Stack management (iv)

- The popped data remain in the memory
  - But they should not be used anymore because they will be overwritten by the next pushed data.

ASM

```
...
li   sp, 0x20000
...
li   t6, 0x1234
add  sp, sp, -4
sw   t6, 0(sp)
mv   t6, zero
...
lw   t6, 0(sp)
add  sp, sp, 4
...
li   t6, 0x9abc
add  sp, sp, -4
sw   t6, 0(sp)
...
```



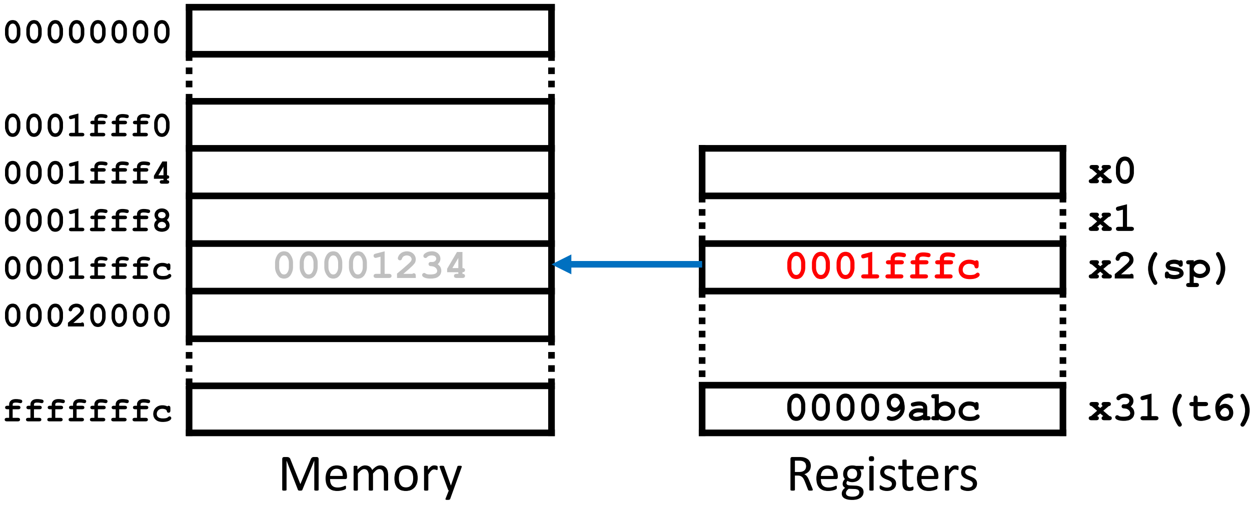
Functions

Stack management (iv)

- The popped data remain in the memory
    - But they should not be used anymore because they will be overwritten by the next pushed data.

ASM

```
...
li   sp, 0x20000
...
li   t6, 0x1234
add  sp, sp, -4
sw   t6, 0(sp)
mv   t6, zero
...
lw   t6, 0(sp)
add  sp, sp, 4
...
li   t6, 0x9abc
add  sp, sp, -4
sw   t6, 0(sp)
...
```



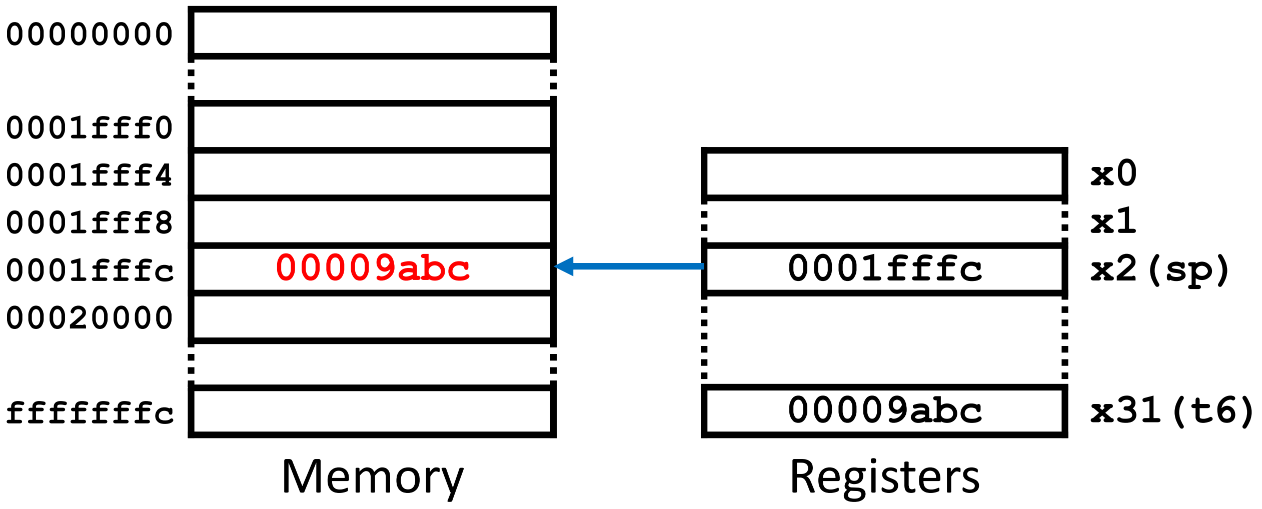
Functions

Stack management (iv)

- The popped data remain in the memory
  - But they should not be used anymore because they will be overwritten by the next pushed data.

ASM

```
...
li   sp, 0x20000
...
li   t6, 0x1234
add  sp, sp, -4
sw   t6, 0(sp)
mv   t6, zero
...
lw   t6, 0(sp)
add  sp, sp, 4
...
li   t6, 0x9abc
add  sp, sp, -4
sw   t6, 0(sp)
...
```



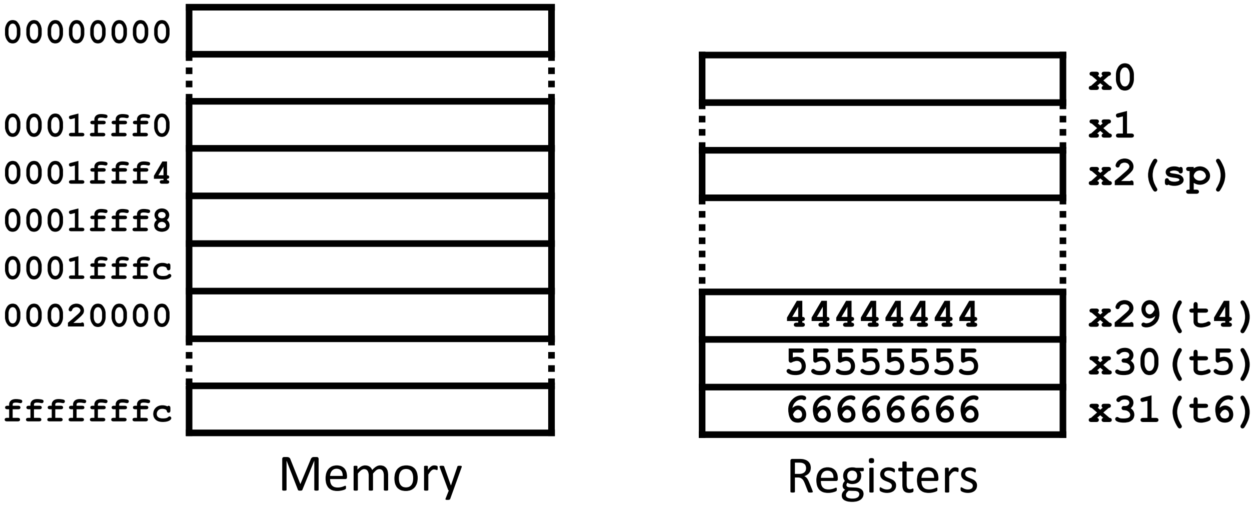
Functions

Stack management (v)

- Pushing a group of data implies:
  - Decrementing **`sp`** (the number of bytes of all of them).
  - Storing each data in consecutive addresses on the top of the stack.

ASM

```
...
li   sp, 0x20000
...
add  sp, sp, -12
sw   t4, 8(sp)
sw   t5, 4(sp)
sw   t6, 0(sp)
...
mv   t4, zero
...
lw   t4, 8(sp)
lw   t5, 4(sp)
lw   t6, 0(sp)
add  sp, sp, 12
...
```



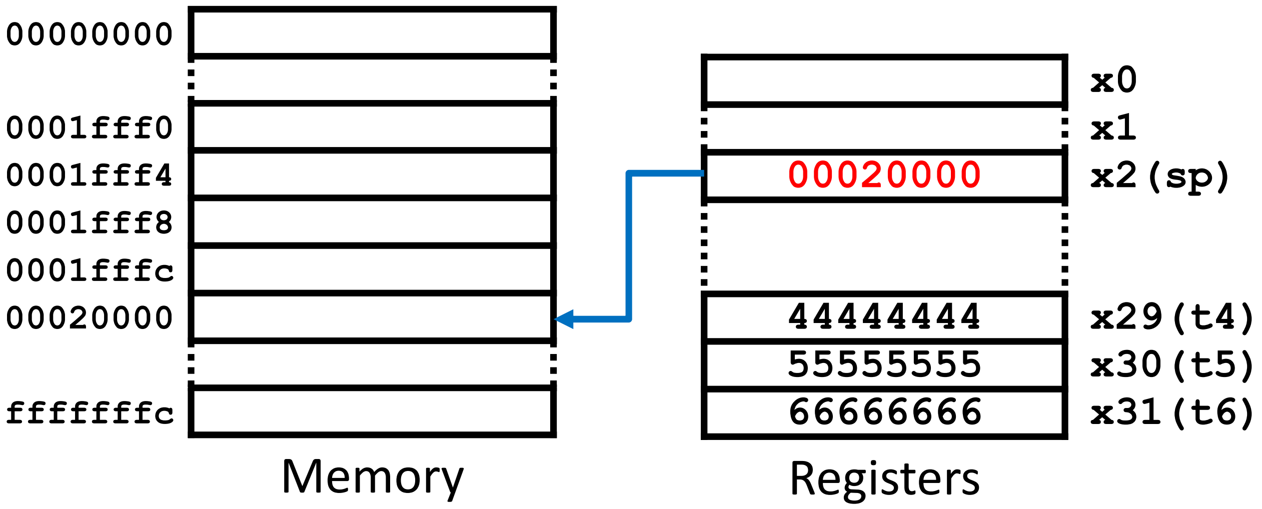
Functions

Stack management (v)

- Pushing a group of data implies:
  - Decrementing **`sp`** (the number of bytes of all of them).
  - Storing each data in consecutive addresses on the top of the stack.

ASM

```
...
li   sp, 0x20000
...
add  sp, sp, -12
sw   t4, 8(sp)
sw   t5, 4(sp)
sw   t6, 0(sp)
...
mv   t4, zero
...
lw   t4, 8(sp)
lw   t5, 4(sp)
lw   t6, 0(sp)
add  sp, sp, 12
...
```



00000000	
0001fff0	
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
	x1
00020000	x2(sp)
44444444	x29(t4)
55555555	x30(t5)
66666666	x31(t6)

Registers

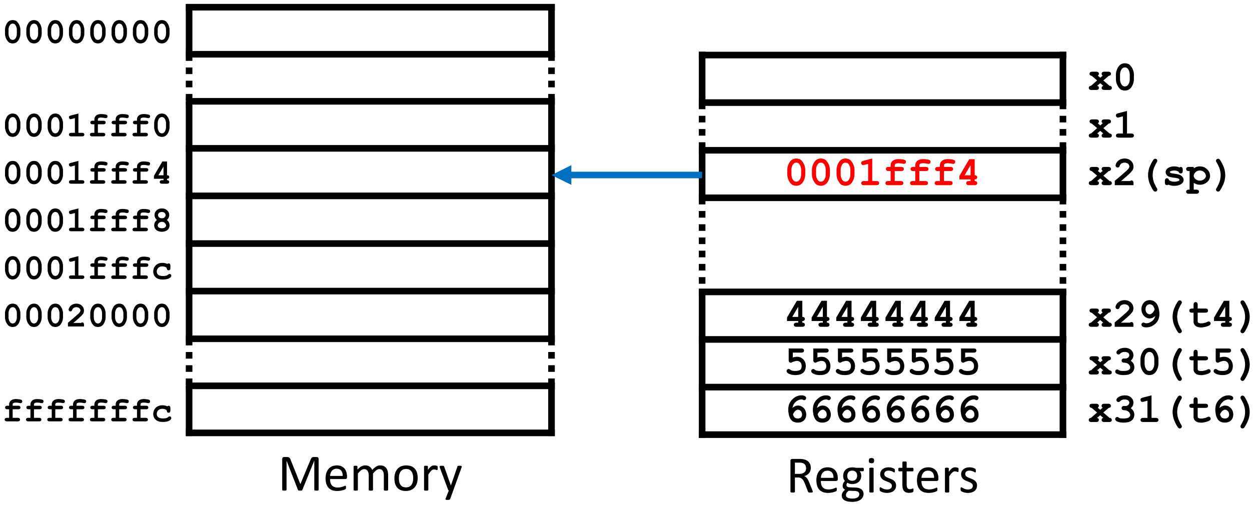
Functions

Stack management (v)

- Pushing a group of data implies:
  - Decrementing **`sp`** (the number of bytes of all of them).
  - Storing each data in consecutive addresses on the top of the stack.

ASM

```
...
li   sp, 0x20000
...
add  sp, sp, -12
sw   t4, 8(sp)
sw   t5, 4(sp)
sw   t6, 0(sp)
...
mv   t4, zero
...
lw   t4, 8(sp)
lw   t5, 4(sp)
lw   t6, 0(sp)
add  sp, sp, 12
...
```



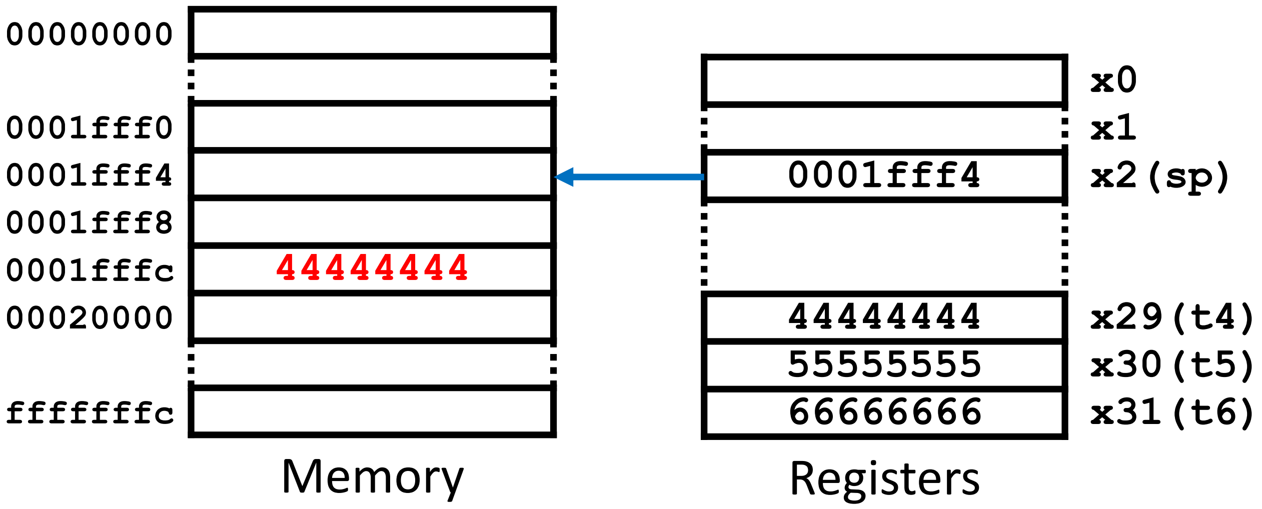
Functions

Stack management (v)

- Pushing a group of data implies:
  - Decrementing **`sp`** (the number of bytes of all of them).
  - Storing each data in consecutive addresses on the top of the stack.

ASM

```
...
li   sp, 0x20000
...
add  sp, sp, -12
sw   t4, 8(sp)
sw   t5, 4(sp)
sw   t6, 0(sp)
...
mv   t4, zero
...
lw   t4, 8(sp)
lw   t5, 4(sp)
lw   t6, 0(sp)
add  sp, sp, 12
...
```



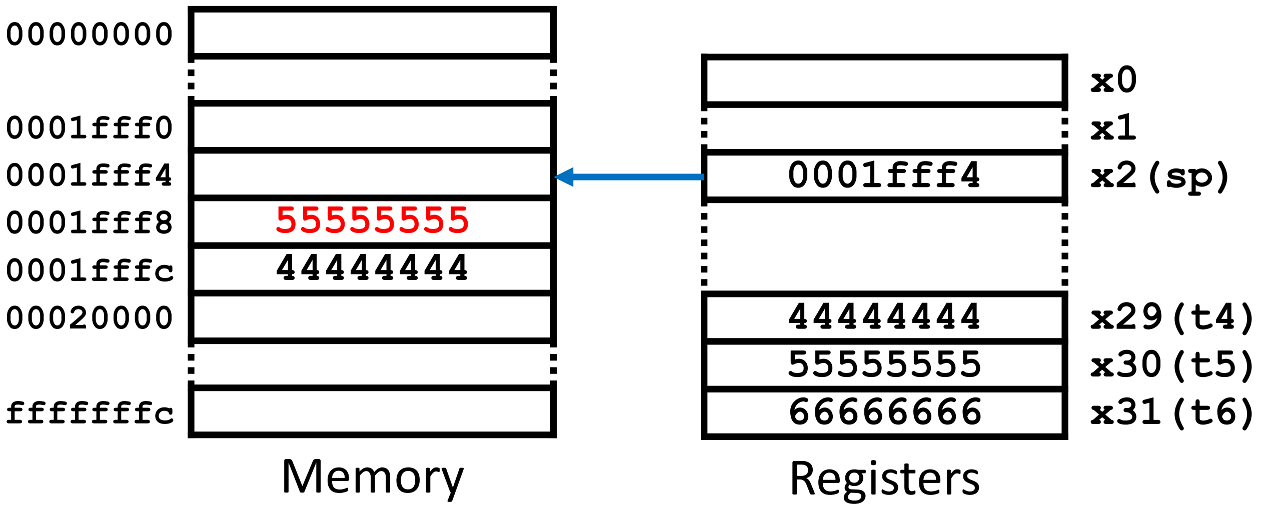
Functions

Stack management (v)

- Pushing a group of data implies:
  - Decrementing **`sp`** (the number of bytes of all of them).
  - Storing each data in consecutive addresses on the top of the stack.

ASM

```
...
li   sp, 0x20000
...
add  sp, sp, -12
sw   t4, 8(sp)
sw   t5, 4(sp)
sw   t6, 0(sp)
...
mv   t4, zero
...
lw   t4, 8(sp)
lw   t5, 4(sp)
lw   t6, 0(sp)
add  sp, sp, 12
...
```



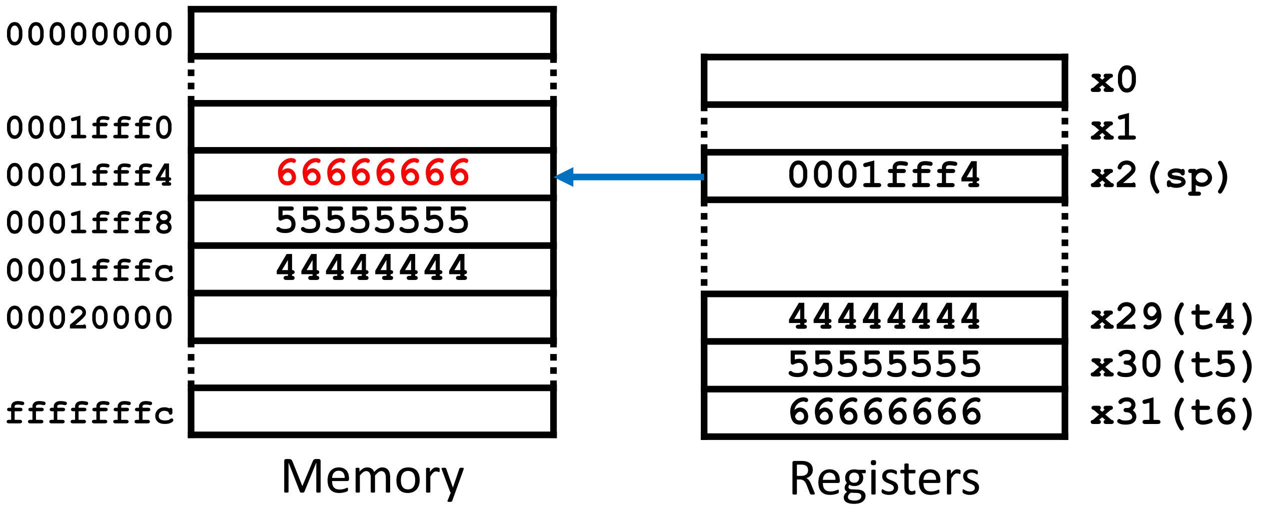
Functions

Stack management (v)

- Pushing a group of data implies:
  - Decrementing **`sp`** (the number of bytes of all of them).
  - Storing each data in consecutive addresses on the top of the stack.

ASM

```
...
li   sp, 0x20000
...
add  sp, sp, -12
sw   t4, 8(sp)
sw   t5, 4(sp)
sw   t6, 0(sp)
...
mv   t4, zero
...
lw   t4, 8(sp)
lw   t5, 4(sp)
lw   t6, 0(sp)
add  sp, sp, 12
...
```



00000000	
0001fff0	
0001fff4	66666666
0001fff8	55555555
0001fffc	44444444
00020000	
fffffffc	

Memory

	x0
	x1
0001fff4	x2(sp)
44444444	x29(t4)
55555555	x30(t5)
66666666	x31(t6)

Registers

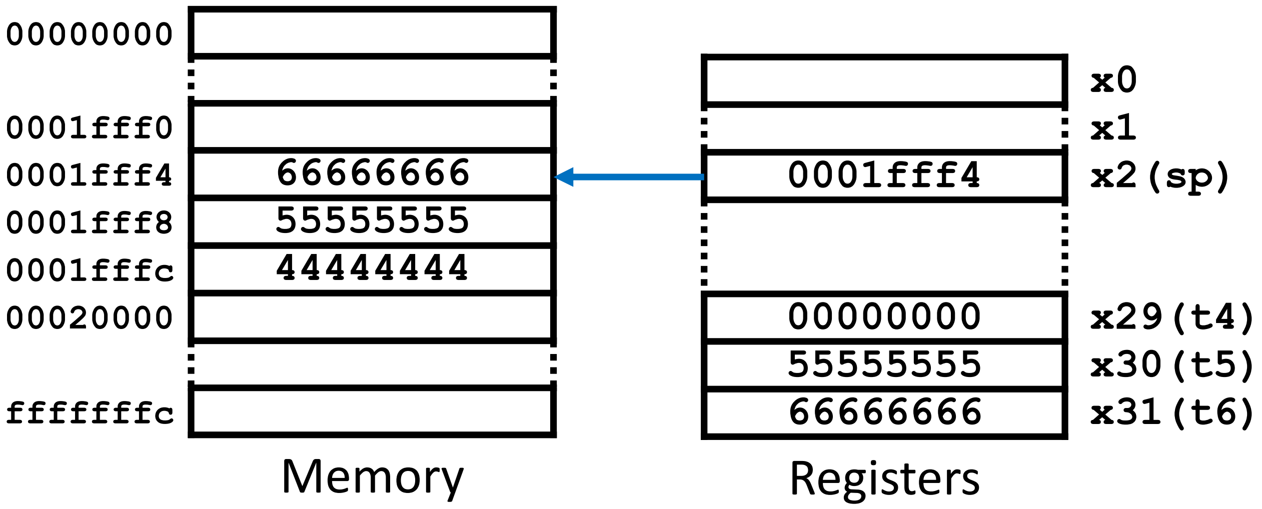
Functions

Stack management (v)

- Pushing a group of data implies:
  - Decrementing **`sp`** (the number of bytes of all of them).
  - Storing each data in consecutive addresses on the top of the stack.

ASM

```
...
li   sp, 0x20000
...
add  sp, sp, -12
sw   t4, 8(sp)
sw   t5, 4(sp)
sw   t6, 0(sp)
...
mv   t4, zero
...
lw   t4, 8(sp)
lw   t5, 4(sp)
lw   t6, 0(sp)
add  sp, sp, 12
...
```



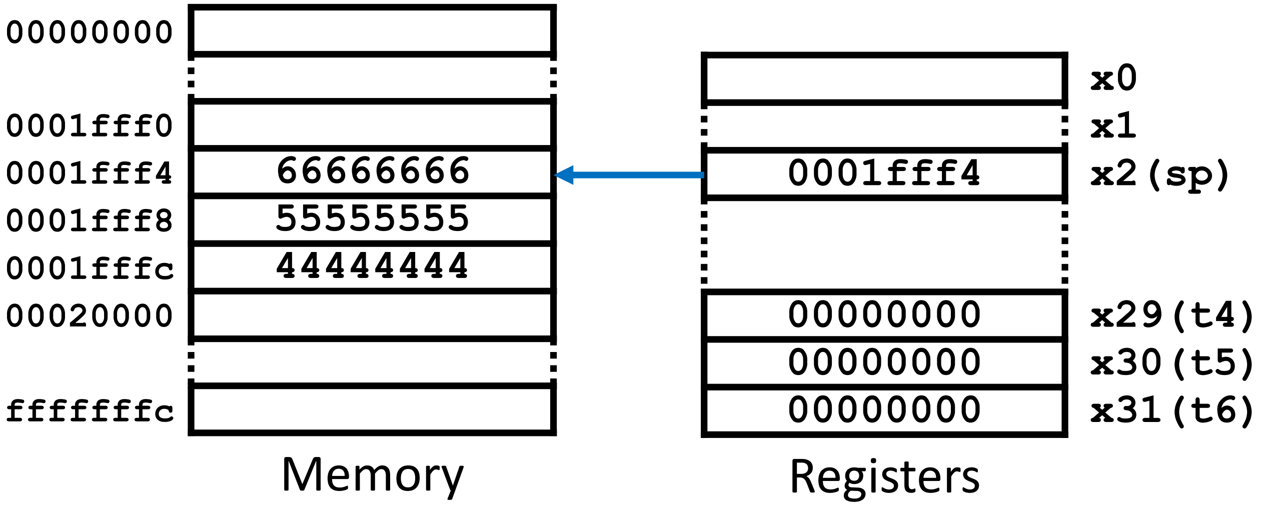
Functions

Stack management (vi)

- Popping a group of data implies:
  - Loading the data located in consecutive addresses on the top of the stack.
  - Incrementing **`sp`** (the number of bytes of all of them).

ASM

```
...
li   sp, 0x20000
...
add  sp, sp, -12
sw   t4, 8(sp)
sw   t5, 4(sp)
sw   t6, 0(sp)
...
mv   t4, zero
...
lw   t4, 8(sp)
lw   t5, 4(sp)
lw   t6, 0(sp)
add  sp, sp, 12
...
```



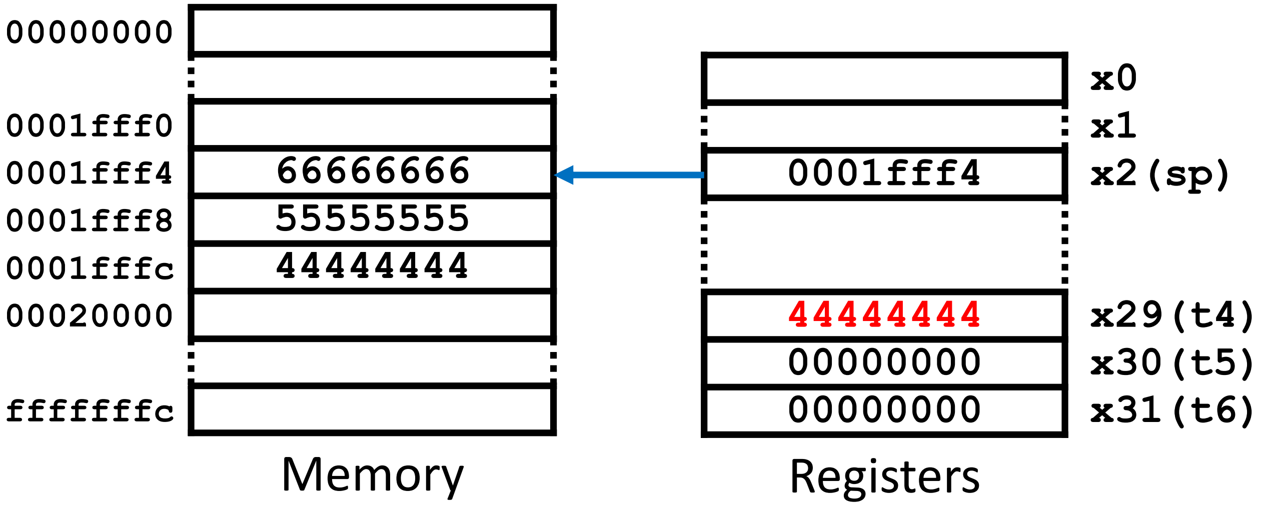
Functions

Stack management (vi)

- Popping a group of data implies:
  - Loading the data located in consecutive addresses on the top of the stack.
  - Incrementing **sp** (the number of bytes of all of them).

ASM

```
...
li   sp, 0x20000
...
add  sp, sp, -12
sw   t4, 8(sp)
sw   t5, 4(sp)
sw   t6, 0(sp)
...
mv   t4, zero
...
lw   t4, 8(sp)
lw   t5, 4(sp)
lw   t6, 0(sp)
add  sp, sp, 12
...
```



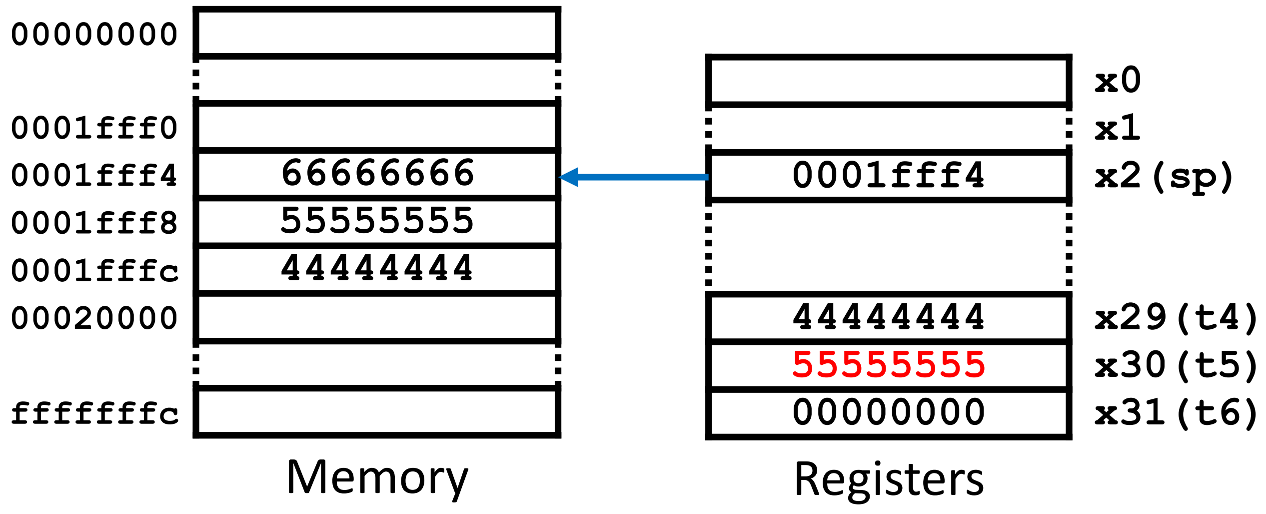
Functions

Stack management (vi)

- Popping a group of data implies:
    - Loading the data located in consecutive addresses on the top of the stack.
    - Incrementing **`sp`** (the number of bytes of all of them).

ASM

```
...
li   sp, 0x20000
...
add  sp, sp, -12
sw   t4, 8(sp)
sw   t5, 4(sp)
sw   t6, 0(sp)
...
mv   t4, zero
...
lw   t4, 8(sp)
lw   t5, 4(sp)
lw   t6, 0(sp)
add  sp, sp, 12
...
```



00000000	
0001fff0	
0001fff4	66666666
0001fff8	55555555
0001fffc	44444444
00020000	
fffffffc	

Memory

	x0
	x1
0001fff4	x2(sp)
44444444	x29(t4)
55555555	x30(t5)
00000000	x31(t6)

Registers

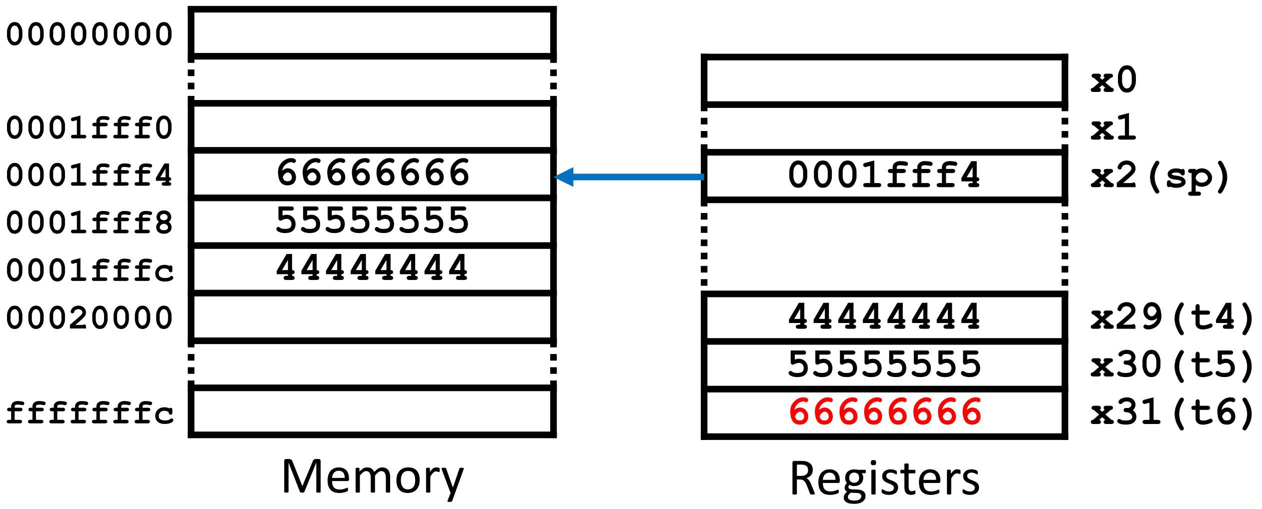
Functions

Stack management (vi)

- Popping a group of data implies:
  - Loading the data located in consecutive addresses on the top of the stack.
  - Incrementing **`sp`** (the number of bytes of all of them).

ASM

```
...
li   sp, 0x20000
...
add  sp, sp, -12
sw   t4, 8(sp)
sw   t5, 4(sp)
sw   t6, 0(sp)
...
mv   t4, zero
...
lw   t4, 8(sp)
lw   t5, 4(sp)
lw   t6, 0(sp)
add  sp, sp, 12
...
```



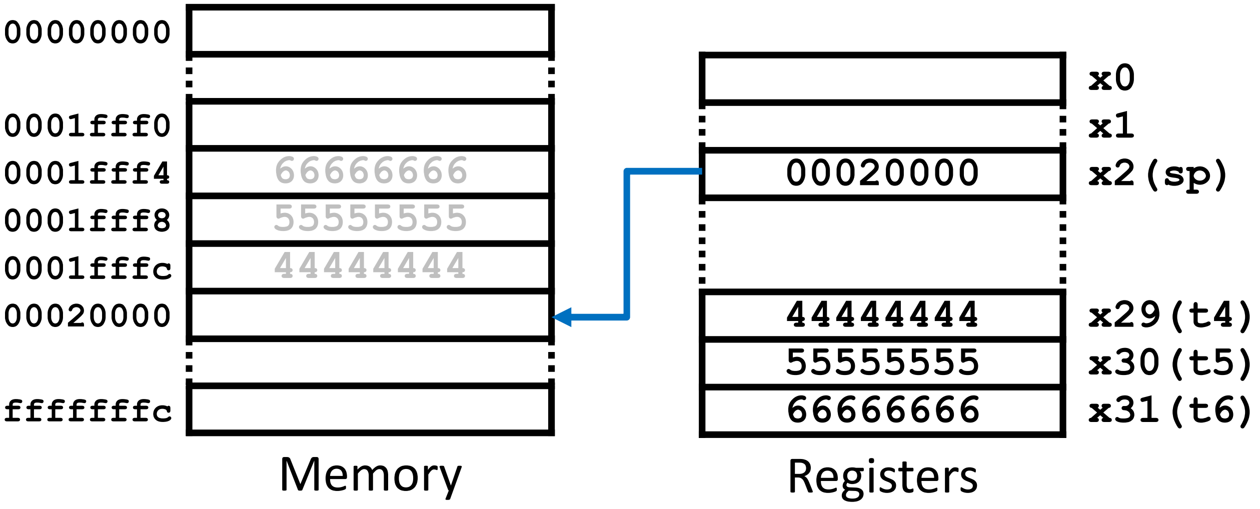
Functions

Stack management (vi)

- Popping a group of data implies:
  - Loading the data located in consecutive addresses on the top of the stack.
  - Incrementing **`sp`** (the number of bytes of all of them).

ASM

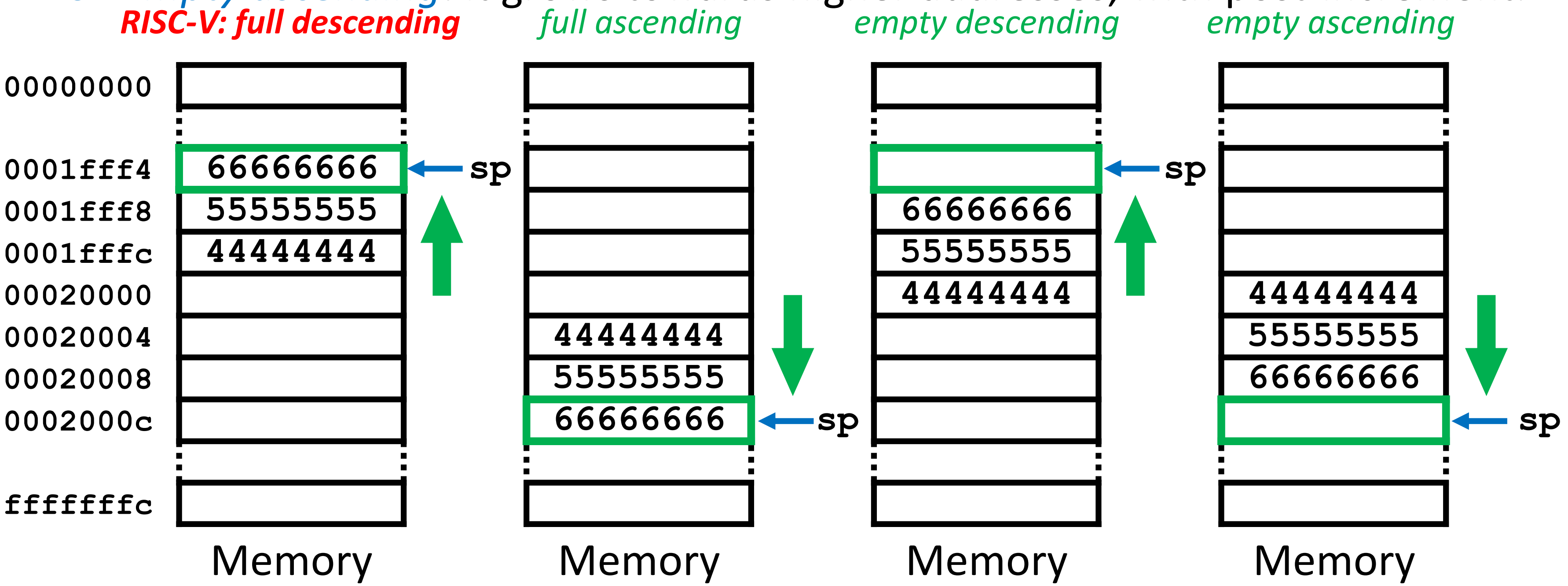
```
...
li   sp, 0x20000
...
add  sp, sp, -12
sw   t4, 8(sp)
sw   t5, 4(sp)
sw   t6, 0(sp)
...
mv   t4, zero
...
lw   t4, 8(sp)
lw   t5, 4(sp)
lw   t6, 0(sp)
add  sp, sp, 12
...
```



Functions

Stack management (vii)

- The RISC-V stack is *full descending* because:
  - It grows towards lower addresses and pushes data by pre-decrementing **`sp`**, i.e., first **`sp`** is decremented and after that the data is stored on the top of the stack.
- Other architectures may have different procedures:
  - *Full ascending*: it grows towards higher addresses, with pre-increment.
  - *Empty descending*: it grows towards lower addresses, with post-decrement.
  - *Empty ascending*: it grows towards higher addresses, with post-increment.



	RISC-V: full descending	*full ascending*	*empty descending*	*empty ascending*
`00000000`				
`0001fff4`	`66666666` ← `sp`		← `sp`	
`0001fff8`	`55555555`		`66666666`	
`0001fffc`	`44444444`		`55555555`	
`00020000`			`44444444`	`44444444`
`00020004`		`44444444`		`55555555`
`00020008`		`55555555`		`66666666`
`0002000c`		`66666666` ← `sp`		← `sp`
`fffffffc`				
	Memory	Memory	Memory	Memory

Functions

Saving registers (i)

- Rule of the caller function. The caller function must:
  - Push the temporary registers that contain values that will be needed later before branching to the callee function.
  - Pop those registers after returning from the callee function.

C/C++

```
int a;
...
a = inc( a );
...
int inc( int x )
{
   return x+1;
}
...
```

ASM

```
a:  .space 4
    ...
    la    t0, a
    lw    a0, 0(t0)
    add   sp, sp, -4
    sw    t0, 0(sp)
    call  inc
    lw    t0, 0(sp)
    add   sp, sp, 4
    sw    a0, 0(t0)
    ...
inc:
    add   t0, a0, 1
    mv    a0, t0
    ret
    ...
```

The caller function uses `t0` *to store the address of* `a`

The caller function pushes the value of `t0` *before branching to* `inc`*, because it will be needed later*

The caller function pops the value of t0 before using it again

Since `t0` *is a temporary register, the callee function may modify it without restrictions*

Functions

Saving registers (ii)

- Rule of the callee function. The callee function must:
  - Push the preserved registers that it uses before updating their value.
    - This set of registers is called the caller function context .
  - Pop those registers after returning to the caller function.

C/C++

```c
int a;
...
a = inc( a );
...
int inc( int x )
{
   return x+1;
}
...
```

ASM

```asm
a:  .space 4
    ...
    la    s0, a
    lw    a0, 0(s0)
    call  inc
    sw    a0, 0(s0)
    ...
inc:
    add   sp, sp, -4
    sw    s0, 0(sp)
    add   s0, a0, 1
    mv    a0, s0
    lw    s0, 0(sp)
    add   sp, sp, 4
    ret
    ...
```

The caller function uses `s0` *to store the address of* `a`

The caller function assumes that register `s0` *will keep its value after calling* `inc`

Since `s0` *is a preserved register, the callee function must push its value before modifying it.*

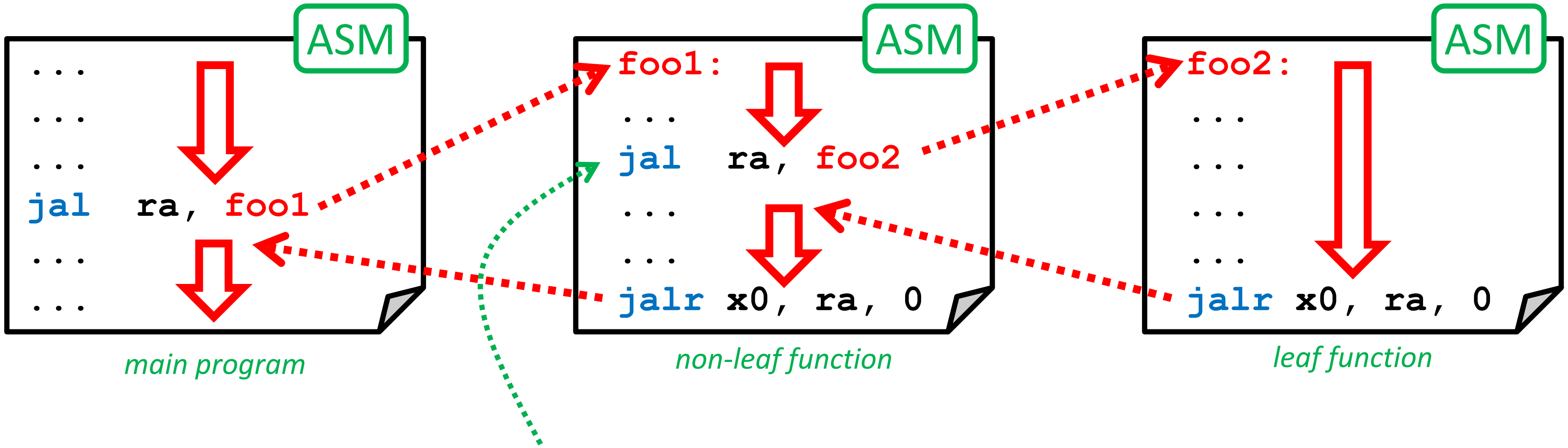
The callee function pops the value of `s0` *before returning to the caller function*

Functions

Nesting and recursion (i)

- If a function does not call another one (leaf function), it just has to push the preserved registers that it is going to use.
  - If it only uses temporary registers, it does not have to push anything.

- But if a function calls others (non-leaf function), apart from pushing the preserved registers that it uses, it also has to push the **`ra`** register.
  - Because this keeps the return address of the caller function.



```
...
...
...
jal  ra, foo1
...
...
```

```
foo1:
...
jal  ra, foo2
...
...
jalr x0, ra, 0
```

```
foo2:
...
...
...
...
jalr x0, ra, 0
```

This instruction branches to **`foo2`** *and overwrites* **`ra`** *with the return address to* **`foo1`**.
If **`ra`** *is not saved before branching,* **`foo1`** *does not return to the main program (it always returns to itself)*

Functions

Nesting and recursion (ii)

C/C++

```
int a[10];
...
incArray( a, 10 );
...
void incArray( int x[], int n )
{
  int i;

  for( i=0; i<n; i=i+1 )
    x[i] = inc( x[i] );
}

int inc( int x )
{
  return x+1;
}
...
```

x[] → s0, n → s1, i → s2, &x[i] → s3

ASM

```
a:  .space 4*10
    ...
    la    a0, a
    li    a1, 10
    jal   ra, incArray
    ...
incArray:
    add   sp, sp, -20
    sw    ra, 16(sp)
    sw    s0, 12(sp)
    ...
    mv    s0, a0
    mv    s1, a1
    mv    s2, zero
for:
    bge   s2, s1, efor
    sll   t0, s2, 2
    add   s3, s0, t0
    lw    a0, 0(s3)
    jal   ra, inc
    sw    a0, 0(s3)
    add   s2, s2, 1
    j     for
efor:
    lw    ra, 16(sp)
    lw    s0, 12(sp)
    ...
    add   sp, sp, 20
    jalr  x0, ra, 0
inc:
    add   a0, a0, 1
    jalr  x0, ra, 0
    ...
```

`incArray` *is a non-leaf function, and therefore it must push its return address before calling* `inc`*.*

also, the 4 used preserved registers must be pushed

overwrites register `ra`

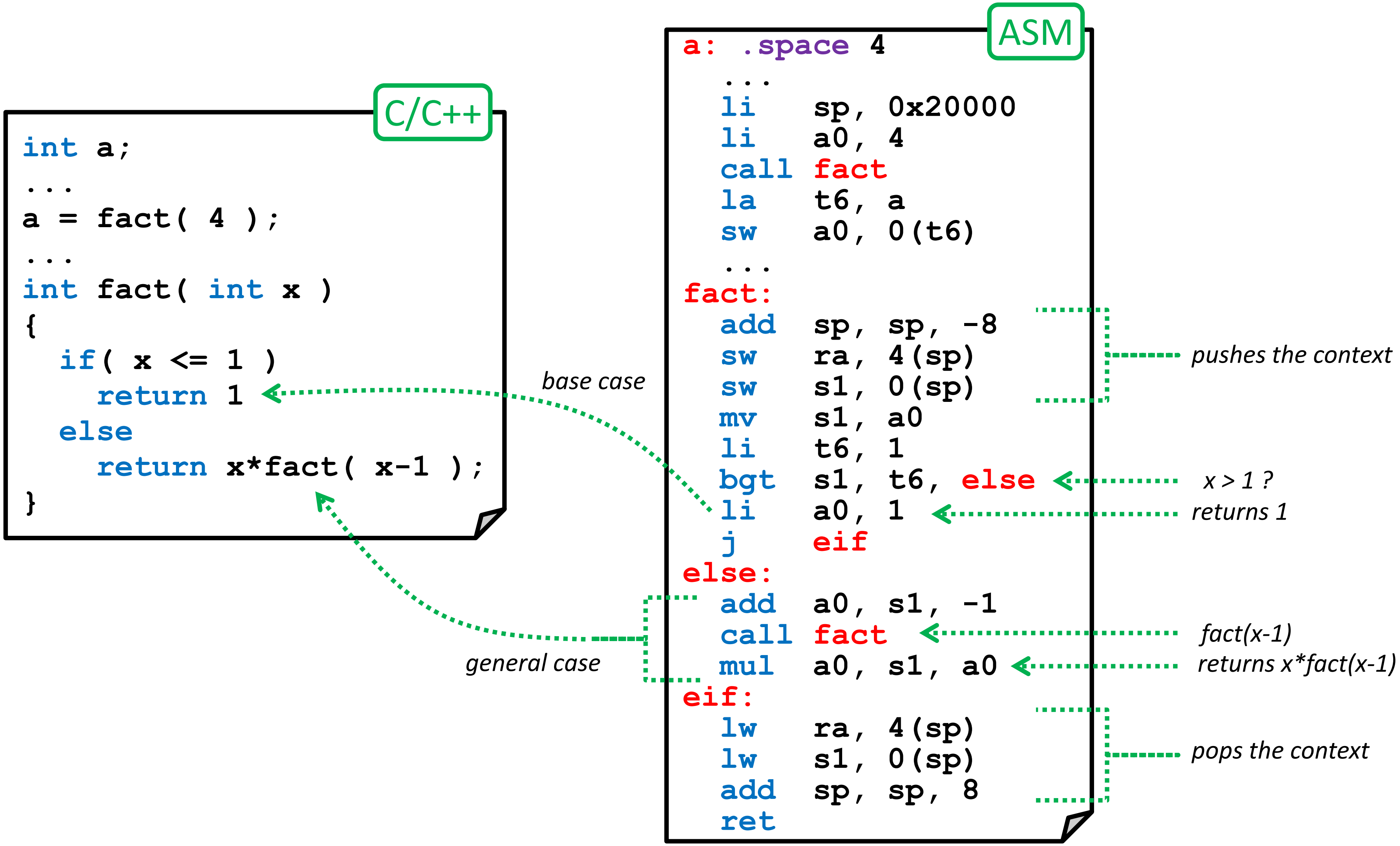
`incArray` *restores register* `ra` *in order to return to the main program*

also the 4 used preserved registers must be restored

Functions

Nesting and recursion (iii)

- Recursive functions are a specific case of nesting (they call themselves).



C/C++

```
int a;
...
a = fact( 4 );
...
int fact( int x )
{
   if( x <= 1 )
      return 1
   else
      return x*fact( x-1 );
}
```

ASM

```
a:  .space 4
    ...
    li   sp, 0x20000
    li   a0, 4
    call fact
    la   t6, a
    sw   a0, 0(t6)
    ...
fact:
    add  sp, sp, -8
    sw   ra, 4(sp)
    sw   s1, 0(sp)
    mv   s1, a0
    li   t6, 1
    bgt  s1, t6, else
    li   a0, 1
    j    eif
else:
    add  a0, s1, -1
    call fact
    mul  a0, s1, a0
eif:
    lw   ra, 4(sp)
    lw   s1, 0(sp)
    add  sp, sp, 8
    ret
```

base case

general case

pushes the context

x > 1 ?

returns 1

fact(x-1)

*returns x*fact(x-1)*

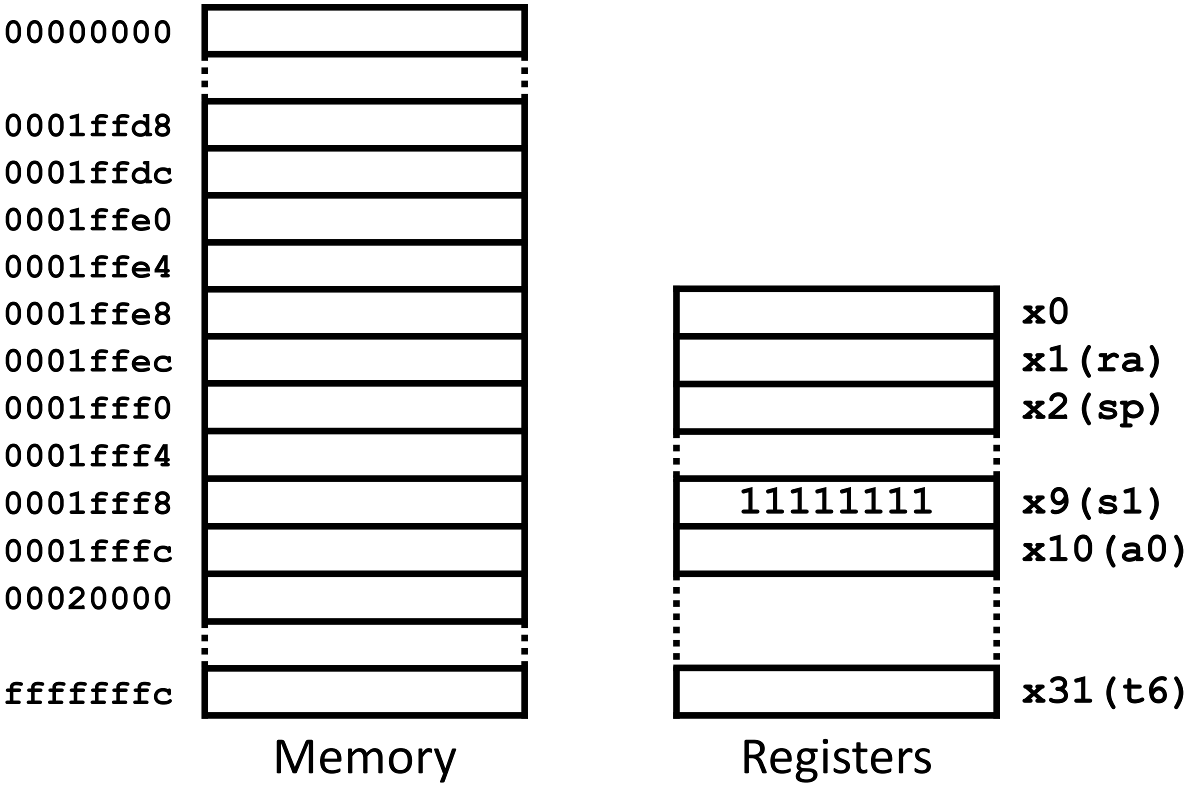
pops the context

Functions

Nesting and recursion (iv)

ASM

```
         a:  .space 4
             ...
             li   sp, 0x20000
             li   a0, 4
             call fact
0000d40c     la   t6, a
             sw   a0, 0(t6)
             ...
         fact:
             add  sp, sp, -8
             sw   ra, 4(sp)
             sw   s1, 0(sp)
             mv   s1, a0
             li   t6, 1
             bgt  s1, t6, else
             li   a0, 1
             j    eif
         else:
             add  a0, s1, -1
             call fact
0000fa28     mul  a0, s1, a0
         eif:
             lw   ra, 4(sp)
             lw   s1, 0(sp)
             add  sp, sp, 8
             ret
```

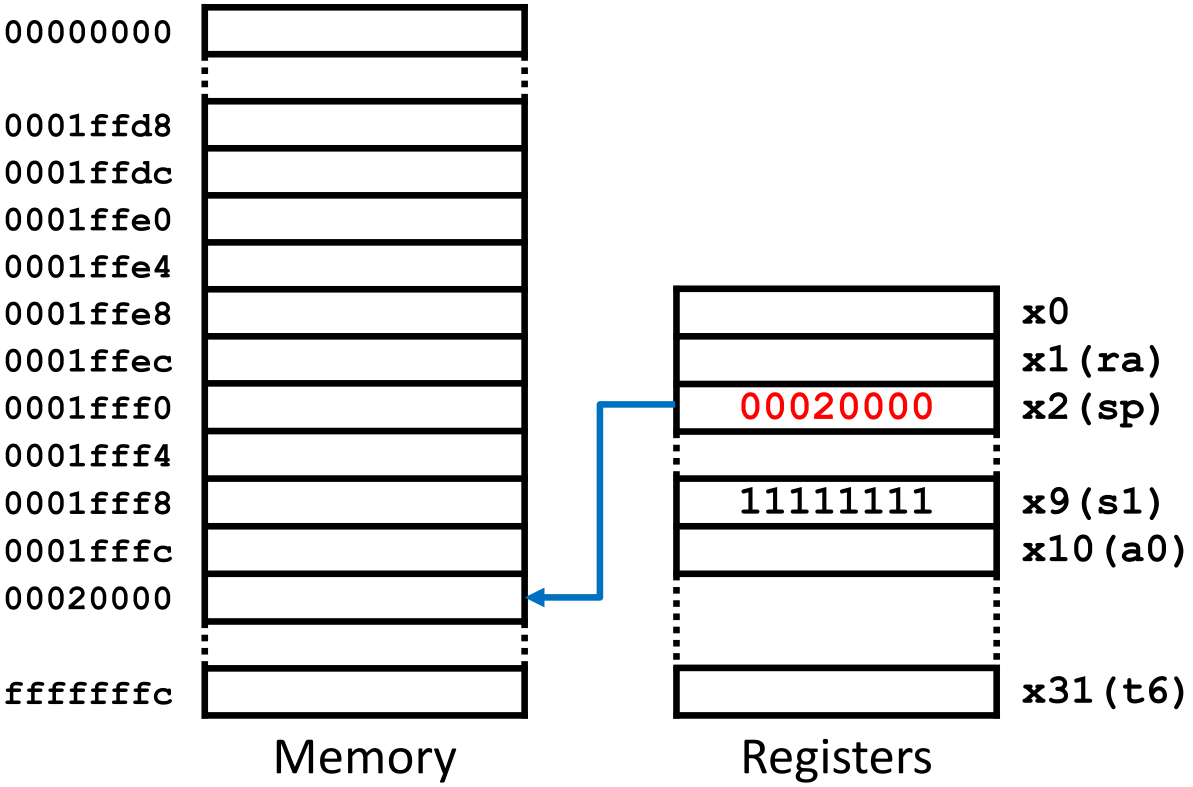


Functions

Nesting and recursion (iv)

ASM

```
         a:  .space 4
               ...
             li    sp, 0x20000
             li    a0, 4
             call  fact
0000d40c     la    t6, a
             sw    a0, 0(t6)
               ...
         fact:
             add   sp, sp, -8
             sw    ra, 4(sp)
             sw    s1, 0(sp)
             mv    s1, a0
             li    t6, 1
             bgt   s1, t6, else
             li    a0, 1
             j     eif
         else:
             add   a0, s1, -1
             call  fact
0000fa28     mul   a0, s1, a0
         eif:
             lw    ra, 4(sp)
             lw    s1, 0(sp)
             add   sp, sp, 8
             ret
```

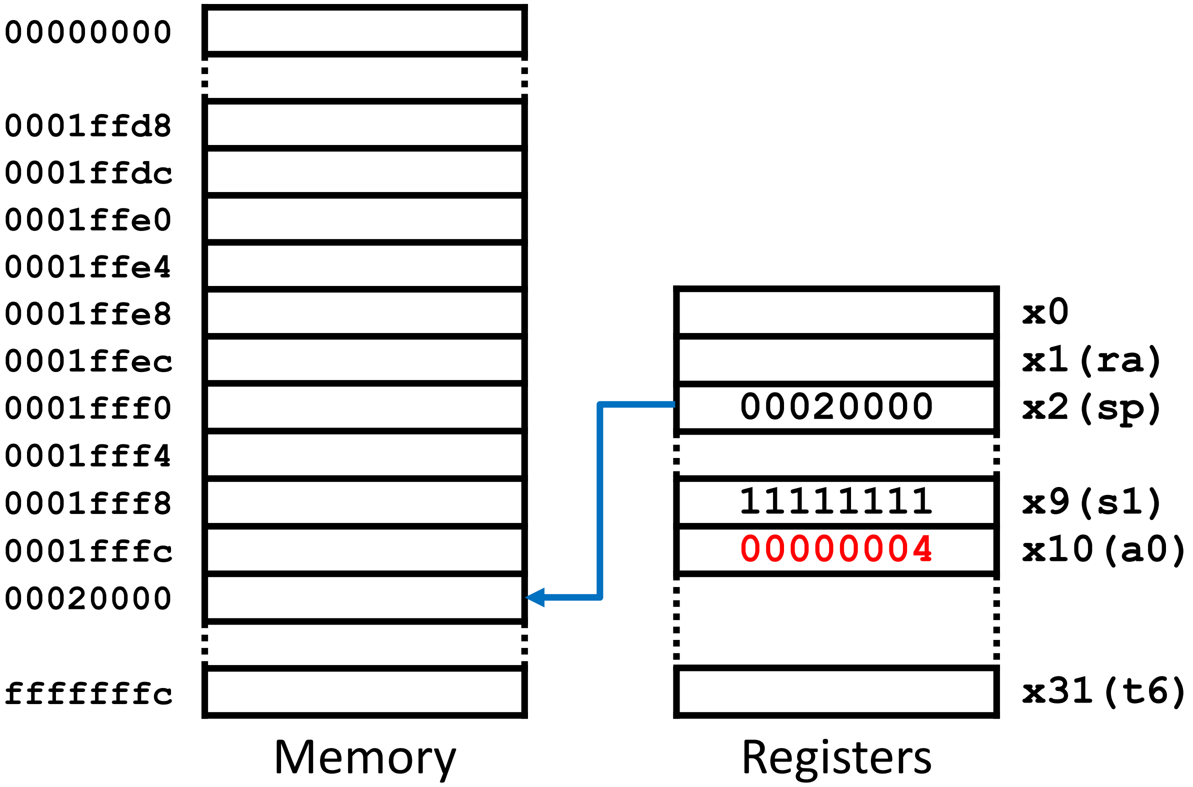


Functions

Nesting and recursion (iv)

ASM

```
         a:  .space 4
               ...
               li    sp, 0x20000
            ⇨  li    a0, 4
               call  fact
0000d40c       la    t6, a
               sw    a0, 0(t6)
               ...
         fact:
               add   sp, sp, -8
               sw    ra, 4(sp)
               sw    s1, 0(sp)
               mv    s1, a0
               li    t6, 1
               bgt   s1, t6, else
               li    a0, 1
               j     eif
         else:
               add   a0, s1, -1
               call  fact
0000fa28       mul   a0, s1, a0
         eif:
               lw    ra, 4(sp)
               lw    s1, 0(sp)
               add   sp, sp, 8
               ret
```

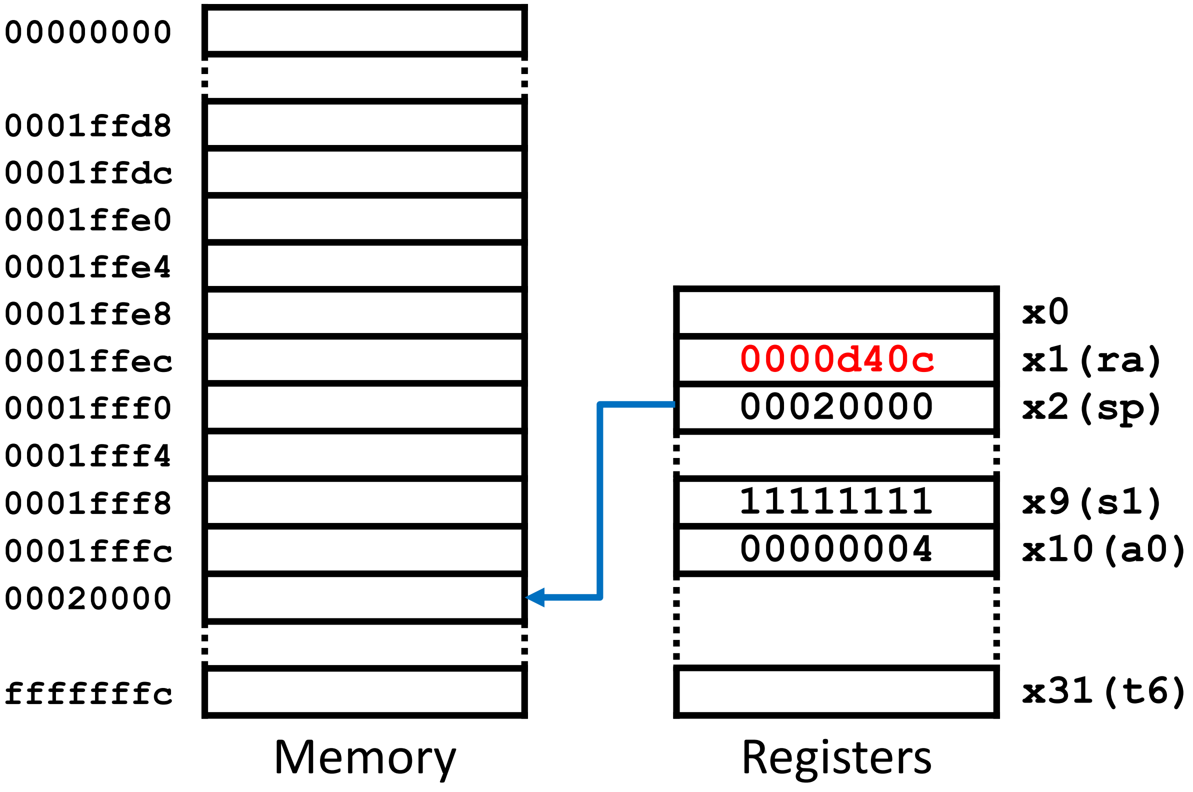


Functions

Nesting and recursion (iv)

ASM

```
a: .space 4
     ...
     li   sp, 0x20000
     li   a0, 4
     call fact
0000d40c la   t6, a
     sw   a0, 0(t6)
     ...
fact:
     add  sp, sp, -8
     sw   ra, 4(sp)
     sw   s1, 0(sp)
     mv   s1, a0
     li   t6, 1
     bgt  s1, t6, else
     li   a0, 1
     j    eif
else:
     add  a0, s1, -1
     call fact
0000fa28 mul  a0, s1, a0
eif:
     lw   ra, 4(sp)
     lw   s1, 0(sp)
     add  sp, sp, 8
     ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
00020000	x2(sp)
11111111	x9(s1)
00000004	x10(a0)
	x31(t6)

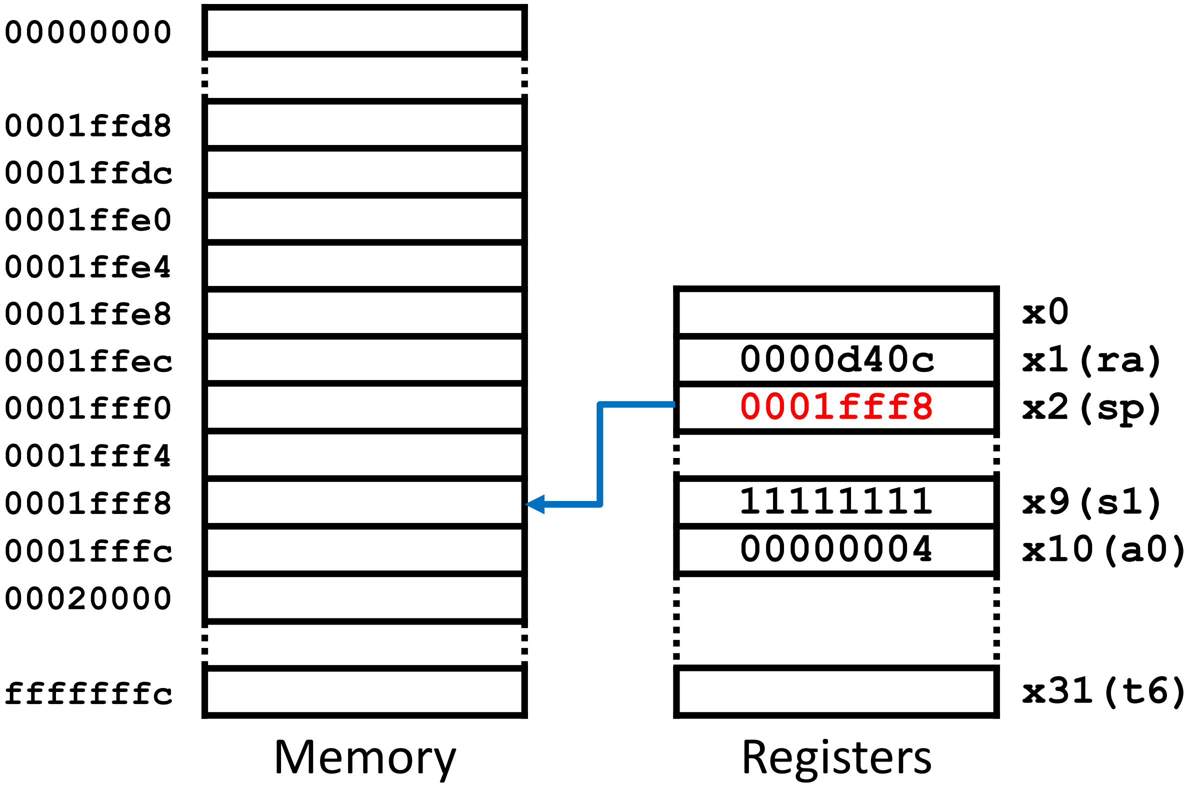
Registers

Functions

Nesting and recursion (iv)

ASM

```
          a:  .space 4
                ...
                li    sp, 0x20000
                li    a0, 4
                call  fact
0000d40c        la    t6, a
                sw    a0, 0(t6)
                ...
          fact:
        ⇨       add   sp, sp, -8
                sw    ra, 4(sp)
                sw    s1, 0(sp)
                mv    s1, a0
                li    t6, 1
                bgt   s1, t6, else
                li    a0, 1
                j     eif
          else:
                add   a0, s1, -1
                call  fact
0000fa28        mul   a0, s1, a0
          eif:
                lw    ra, 4(sp)
                lw    s1, 0(sp)
                add   sp, sp, 8
                ret
```

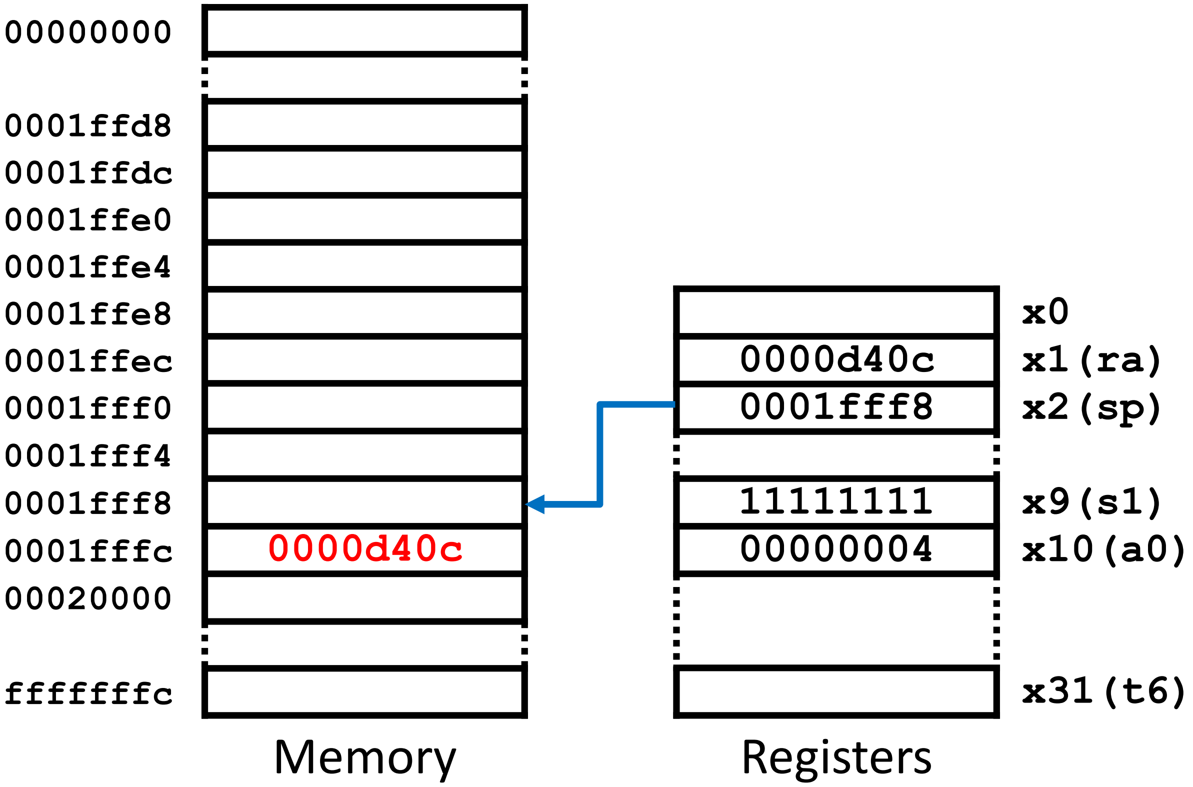


Functions

Nesting and recursion (iv)

ASM

```
         a: .space 4
              ...
              li   sp, 0x20000
              li   a0, 4
              call fact
0000d40c      la   t6, a
              sw   a0, 0(t6)
              ...
         fact:
              add  sp, sp, -8
              sw   ra, 4(sp)
              sw   s1, 0(sp)
              mv   s1, a0
              li   t6, 1
              bgt  s1, t6, else
              li   a0, 1
              j    eif
         else:
              add  a0, s1, -1
              call fact
0000fa28      mul  a0, s1, a0
         eif:
              lw   ra, 4(sp)
              lw   s1, 0(sp)
              add  sp, sp, 8
              ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	
0001fff4	
0001fff8	
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001fff8	x2(sp)
11111111	x9(s1)
00000004	x10(a0)
	x31(t6)

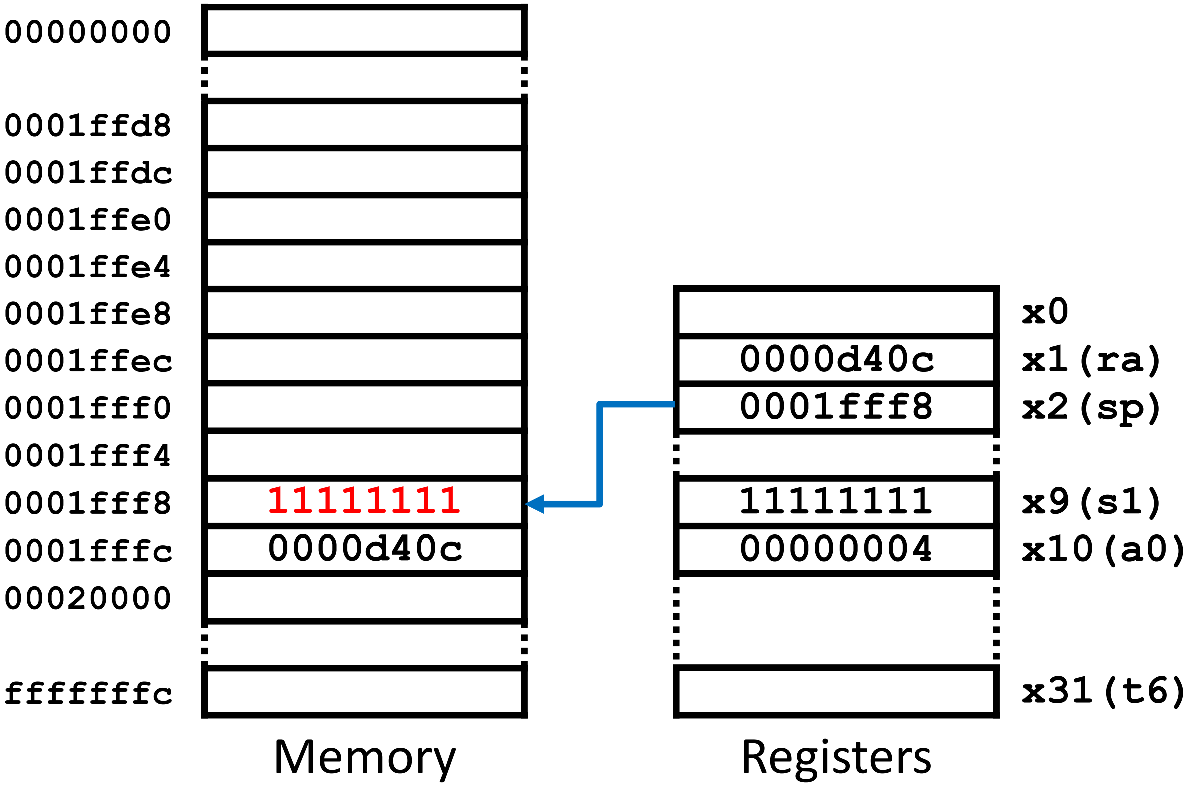
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
    ...
    li    sp, 0x20000
    li    a0, 4
    call  fact
0000d40c la    t6, a
    sw    a0, 0(t6)
    ...
fact:
    add   sp, sp, -8
    sw    ra, 4(sp)
    sw    s1, 0(sp)
    mv    s1, a0
    li    t6, 1
    bgt   s1, t6, else
    li    a0, 1
    j     eif
else:
    add   a0, s1, -1
    call  fact
0000fa28 mul   a0, s1, a0
eif:
    lw    ra, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 8
    ret
```

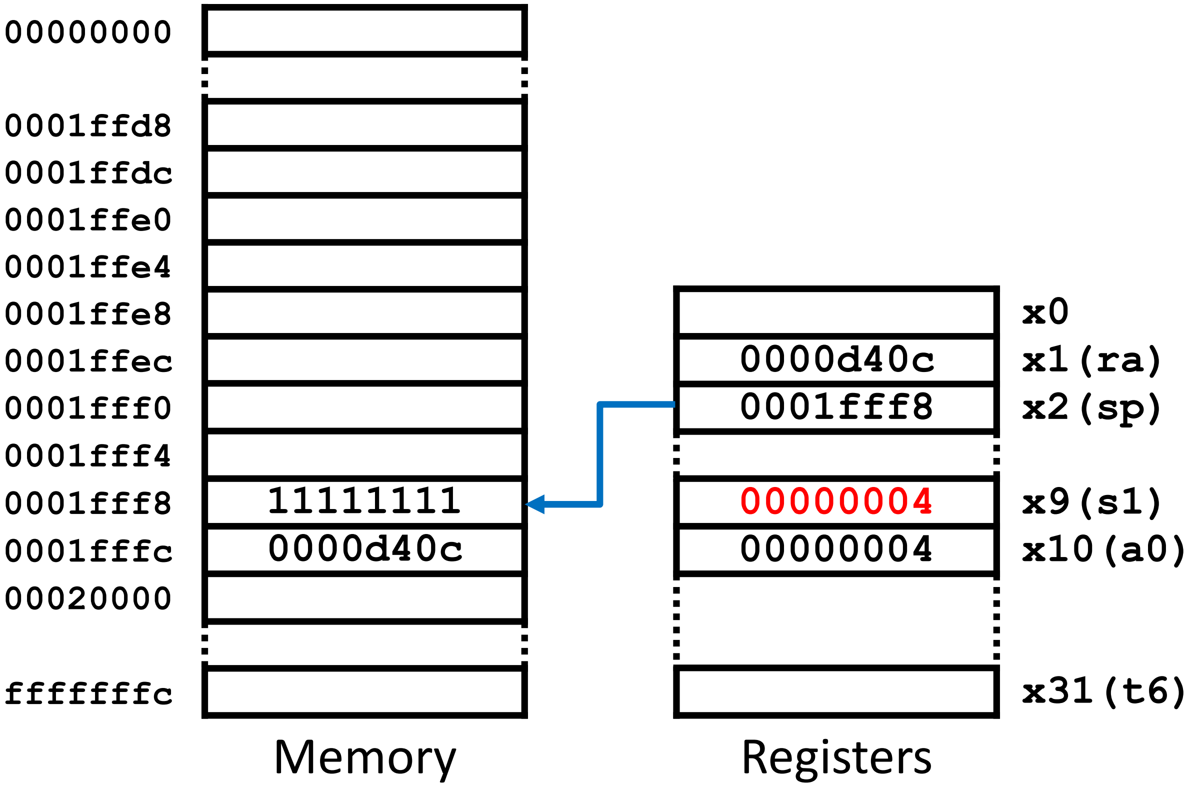


Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
     ...
     li    sp, 0x20000
     li    a0, 4
     call  fact
0000d40c la    t6, a
     sw    a0, 0(t6)
     ...
fact:
     add   sp, sp, -8
     sw    ra, 4(sp)
     sw    s1, 0(sp)
⇨    mv    s1, a0
     li    t6, 1
     bgt   s1, t6, else
     li    a0, 1
     j     eif
else:
     add   a0, s1, -1
     call  fact
0000fa28 mul   a0, s1, a0
eif:
     lw    ra, 4(sp)
     lw    s1, 0(sp)
     add   sp, sp, 8
     ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	
0001fff4	
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001fff8	x2(sp)
00000004	x9(s1)
00000004	x10(a0)
	x31(t6)

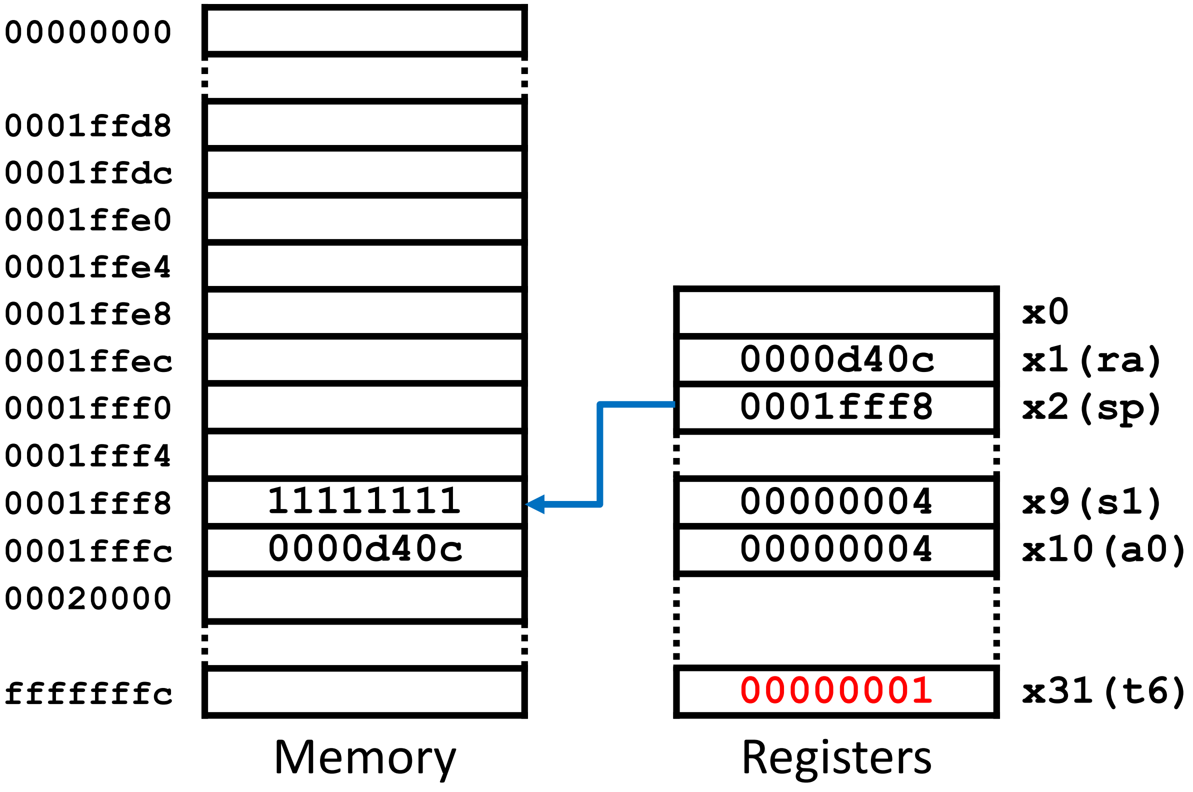
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
      ...
      li    sp, 0x20000
      li    a0, 4
      call  fact
0000d40c la    t6, a
      sw    a0, 0(t6)
      ...
fact:
      add   sp, sp, -8
      sw    ra, 4(sp)
      sw    s1, 0(sp)
      mv    s1, a0
 ⇨    li    t6, 1
      bgt   s1, t6, else
      li    a0, 1
      j     eif
else:
      add   a0, s1, -1
      call  fact
0000fa28 mul   a0, s1, a0
eif:
      lw    ra, 4(sp)
      lw    s1, 0(sp)
      add   sp, sp, 8
      ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	
0001fff4	
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001fff8	x2(sp)
00000004	x9(s1)
00000004	x10(a0)
00000001	x31(t6)

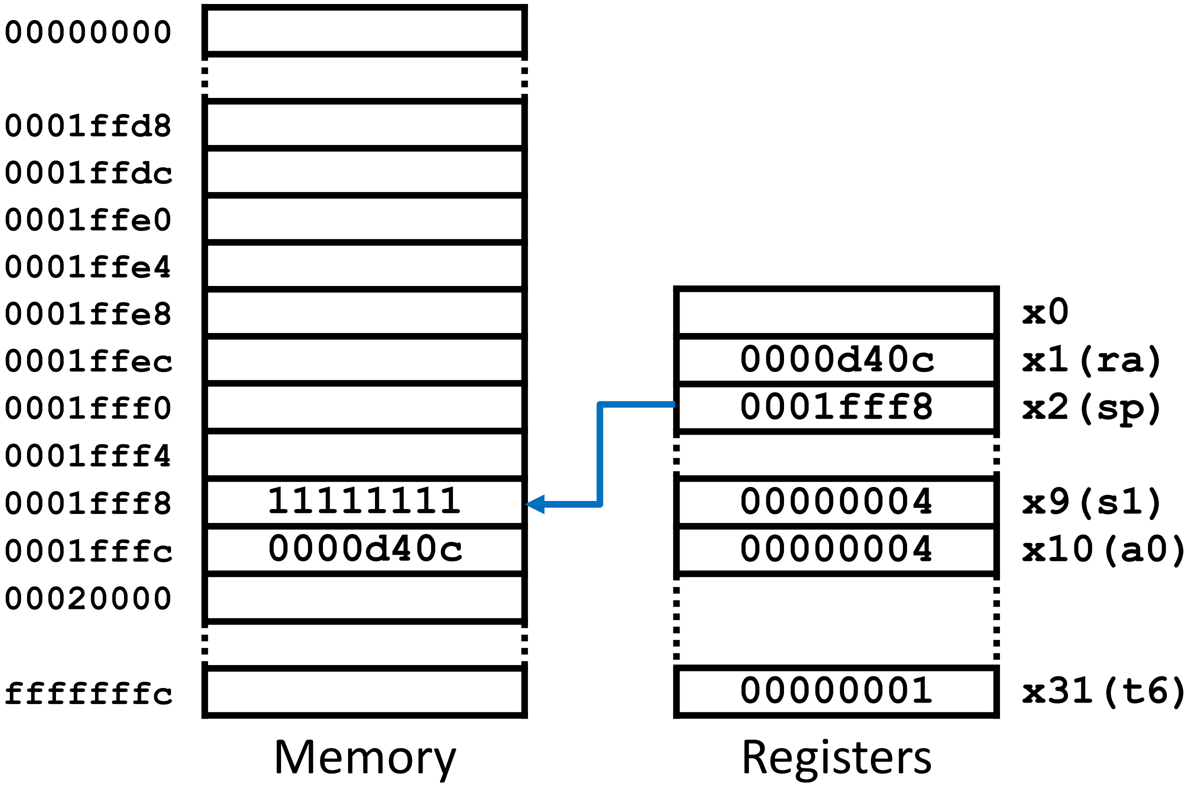
Registers

Functions

Nesting and recursion (iv)

ASM

```
         a: .space 4
            ...
            li   sp, 0x20000
            li   a0, 4
            call fact
0000d40c    la   t6, a
            sw   a0, 0(t6)
            ...
         fact:
            add  sp, sp, -8
            sw   ra, 4(sp)
            sw   s1, 0(sp)
            mv   s1, a0
            li   t6, 1
         ⇨  bgt  s1, t6, else
            li   a0, 1
            j    eif
         else:
            add  a0, s1, -1
            call fact
0000fa28    mul  a0, s1, a0
         eif:
            lw   ra, 4(sp)
            lw   s1, 0(sp)
            add  sp, sp, 8
            ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	
0001fff4	
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001fff8	x2(sp)
00000004	x9(s1)
00000004	x10(a0)
00000001	x31(t6)

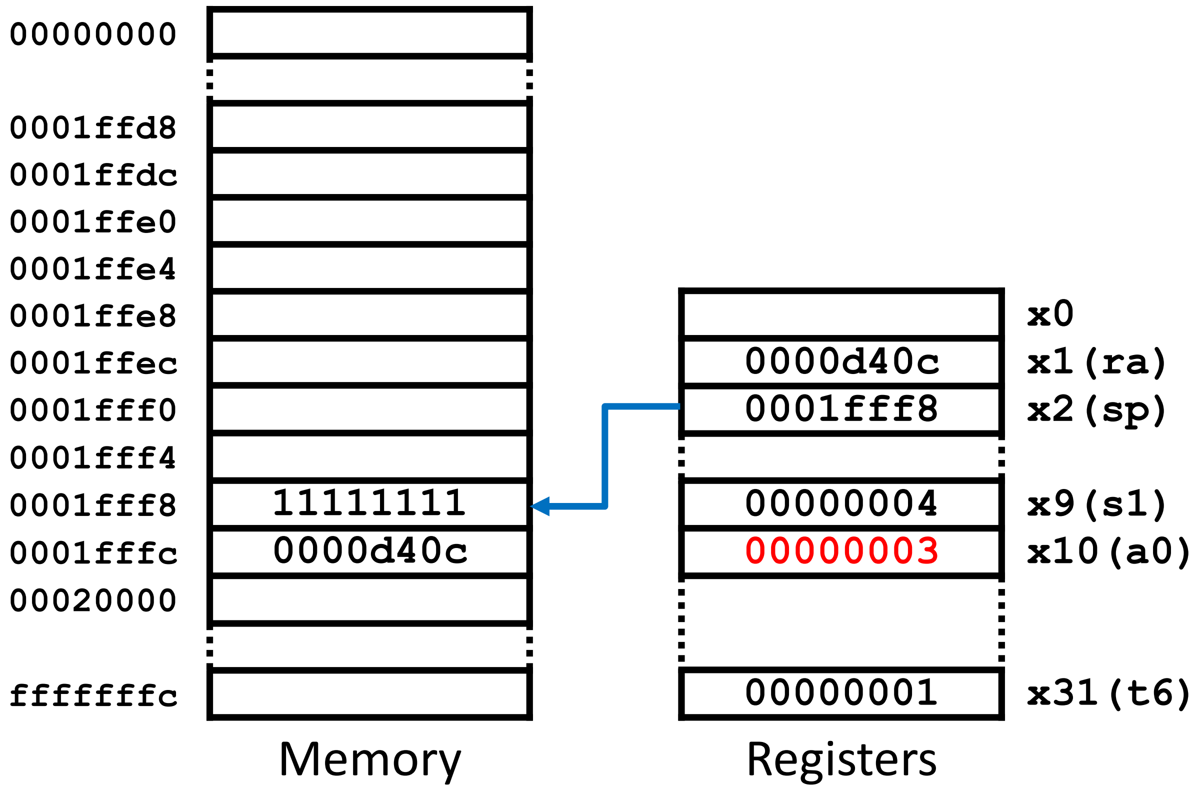
Registers

Functions

Nesting and recursion (iv)

ASM

```
a: .space 4
   ...
   li   sp, 0x20000
   li   a0, 4
   call fact
0000d40c la   t6, a
   sw   a0, 0(t6)
   ...
fact:
   add  sp, sp, -8
   sw   ra, 4(sp)
   sw   s1, 0(sp)
   mv   s1, a0
   li   t6, 1
   bgt  s1, t6, else
   li   a0, 1
   j    eif
else:
   add  a0, s1, -1
   call fact
0000fa28 mul  a0, s1, a0
eif:
   lw   ra, 4(sp)
   lw   s1, 0(sp)
   add  sp, sp, 8
   ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	
0001fff4	
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001fff8	x2(sp)
00000004	x9(s1)
00000003	x10(a0)
00000001	x31(t6)

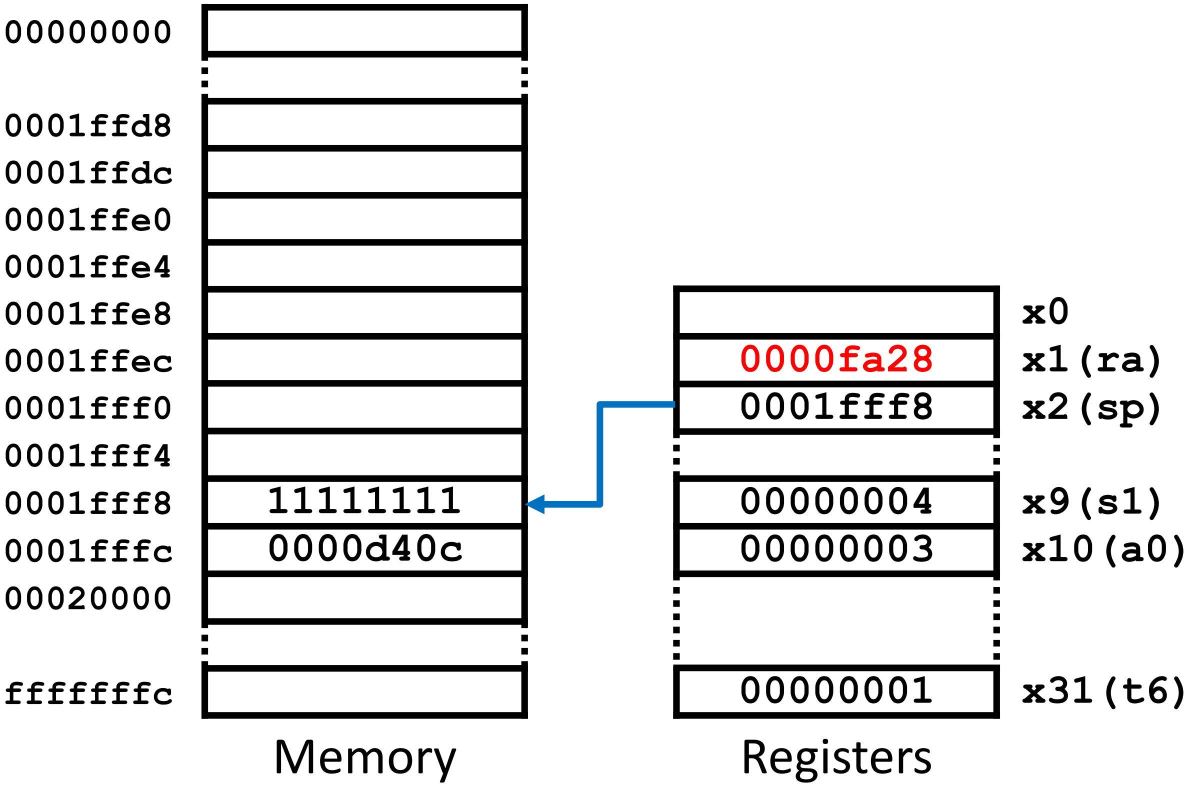
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
    ...
         li    sp, 0x20000
         li    a0, 4
         call  fact
0000d40c la    t6, a
         sw    a0, 0(t6)
         ...
fact:
         add   sp, sp, -8
         sw    ra, 4(sp)
         sw    s1, 0(sp)
         mv    s1, a0
         li    t6, 1
         bgt   s1, t6, else
         li    a0, 1
         j     eif
else:
         add   a0, s1, -1
         call  fact
0000fa28 mul   a0, s1, a0
eif:
         lw    ra, 4(sp)
         lw    s1, 0(sp)
         add   sp, sp, 8
         ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	
0001fff4	
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001fff8	x2(sp)
00000004	x9(s1)
00000003	x10(a0)
00000001	x31(t6)

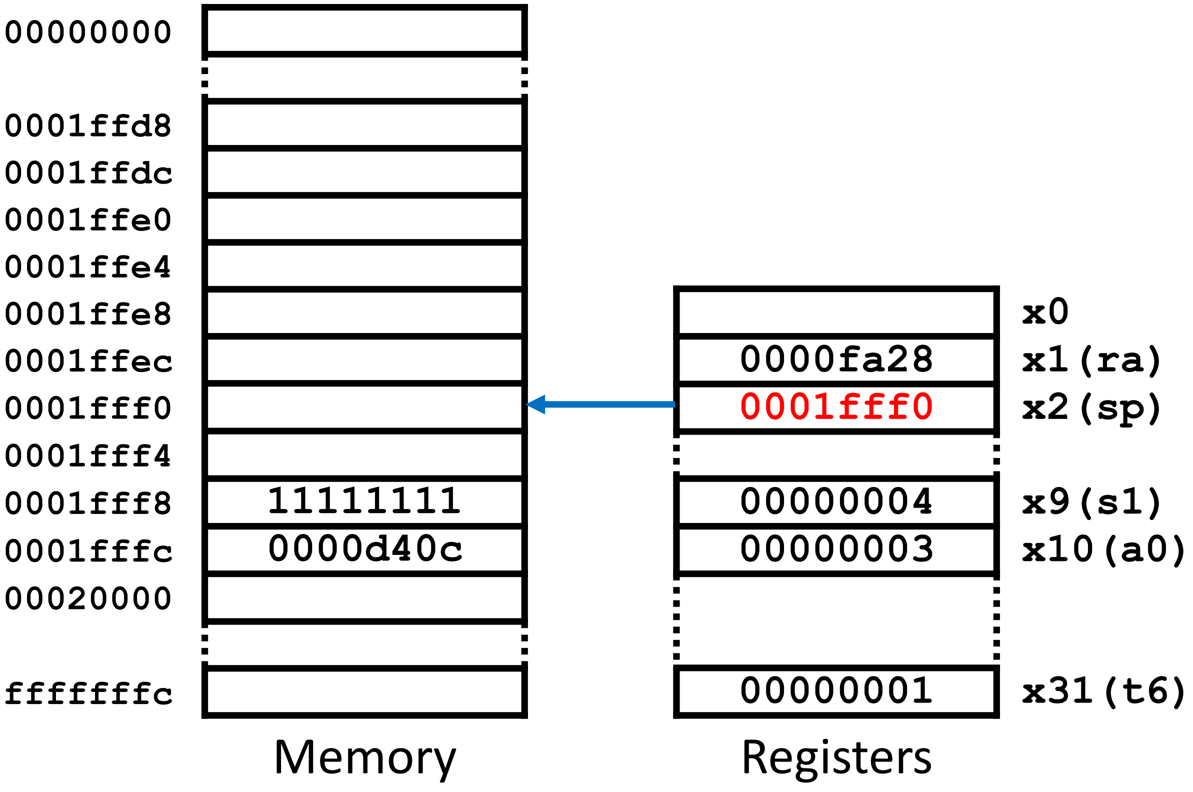
Registers

Functions

Nesting and recursion (iv)

ASM

```
a: .space 4
   ...
   li   sp, 0x20000
   li   a0, 4
   call fact
0000d40c la   t6, a
   sw   a0, 0(t6)
   ...
fact:
   add  sp, sp, -8
   sw   ra, 4(sp)
   sw   s1, 0(sp)
   mv   s1, a0
   li   t6, 1
   bgt  s1, t6, else
   li   a0, 1
   j    eif
else:
   add  a0, s1, -1
   call fact
0000fa28 mul  a0, s1, a0
eif:
   lw   ra, 4(sp)
   lw   s1, 0(sp)
   add  sp, sp, 8
   ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	
0001fff4	
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001fff0	x2(sp)
00000004	x9(s1)
00000003	x10(a0)
00000001	x31(t6)

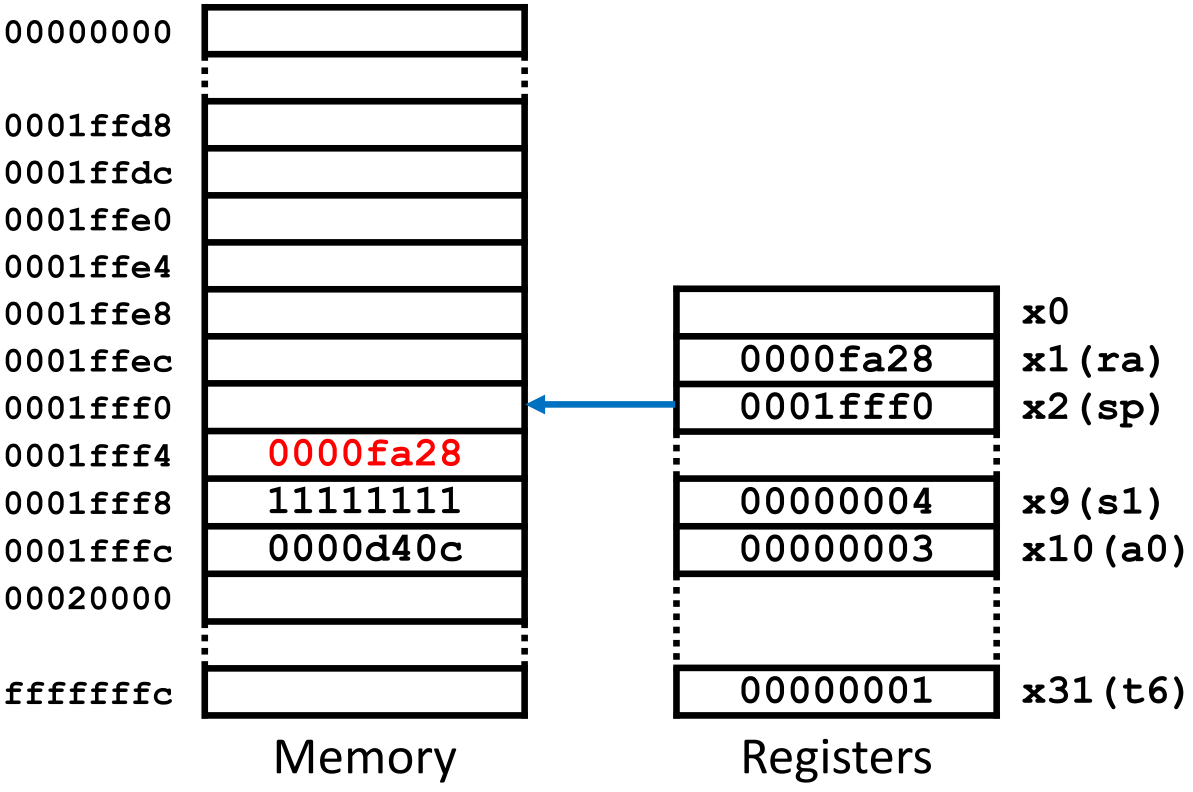
Registers

Functions

Nesting and recursion (iv)

ASM

```
a: .space 4
   ...
   li   sp, 0x20000
   li   a0, 4
   call fact
0000d40c la   t6, a
   sw   a0, 0(t6)
   ...
fact:
   add  sp, sp, -8
   sw   ra, 4(sp)
   sw   s1, 0(sp)
   mv   s1, a0
   li   t6, 1
   bgt  s1, t6, else
   li   a0, 1
   j    eif
else:
   add  a0, s1, -1
   call fact
0000fa28 mul  a0, s1, a0
eif:
   lw   ra, 4(sp)
   lw   s1, 0(sp)
   add  sp, sp, 8
   ret
```

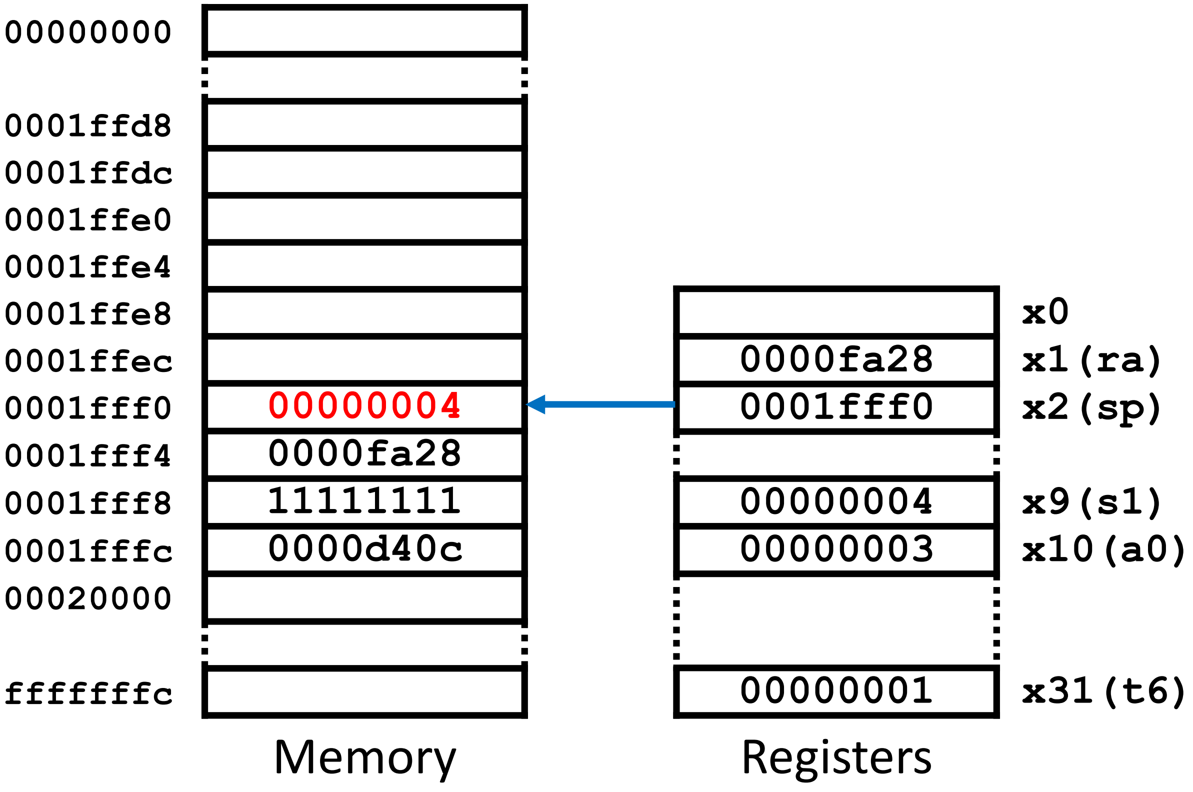


Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
         ...
         li    sp, 0x20000
         li    a0, 4
         call  fact
0000d40c la    t6, a
         sw    a0, 0(t6)
         ...
fact:
         add   sp, sp, -8
         sw    ra, 4(sp)
         sw    s1, 0(sp)
         mv    s1, a0
         li    t6, 1
         bgt   s1, t6, else
         li    a0, 1
         j     eif
else:
         add   a0, s1, -1
         call  fact
0000fa28 mul   a0, s1, a0
eif:
         lw    ra, 4(sp)
         lw    s1, 0(sp)
         add   sp, sp, 8
         ret
```



Memory

00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Registers

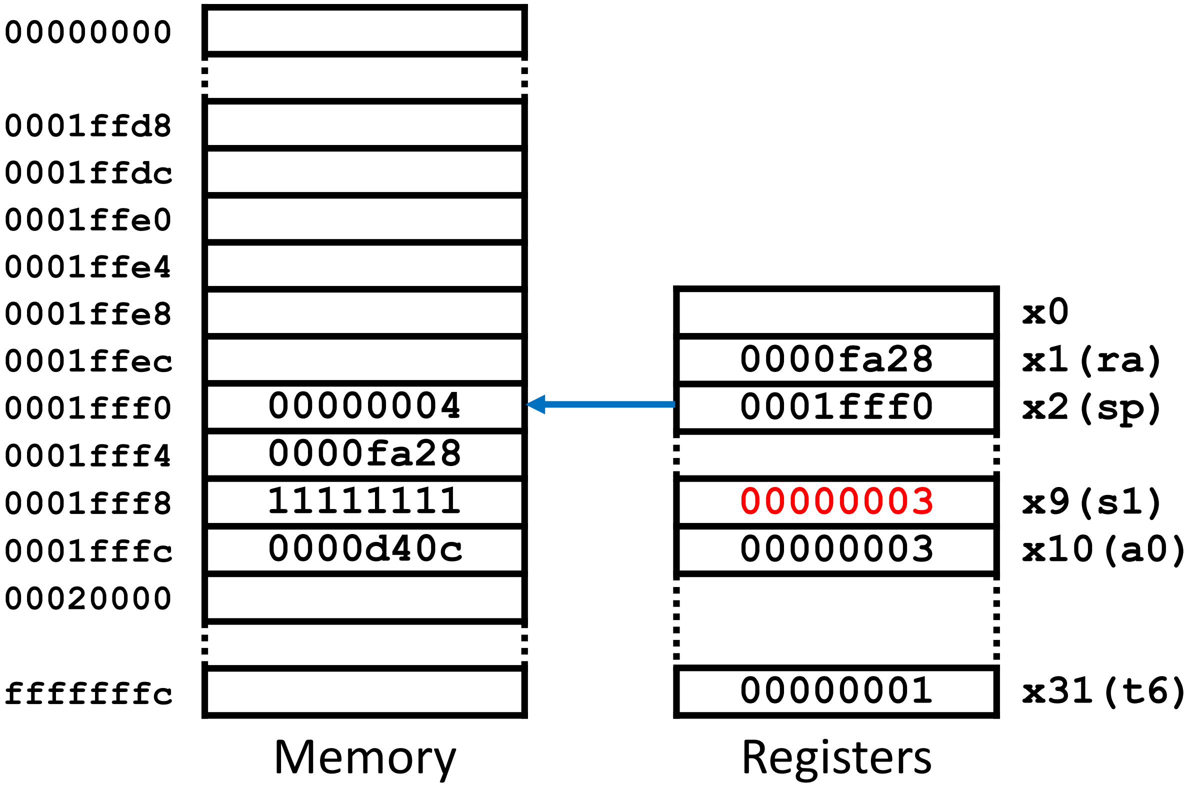
	x0
0000fa28	x1(ra)
0001fff0	x2(sp)
00000004	x9(s1)
00000003	x10(a0)
00000001	x31(t6)

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
          ...
          li   sp, 0x20000
          li   a0, 4
          call fact
0000d40c  la   t6, a
          sw   a0, 0(t6)
          ...
fact:
          add  sp, sp, -8
          sw   ra, 4(sp)
          sw   s1, 0(sp)
      =>  mv   s1, a0
          li   t6, 1
          bgt  s1, t6, else
          li   a0, 1
          j    eif
else:
          add  a0, s1, -1
          call fact
0000fa28  mul  a0, s1, a0
eif:
          lw   ra, 4(sp)
          lw   s1, 0(sp)
          add  sp, sp, 8
          ret
```

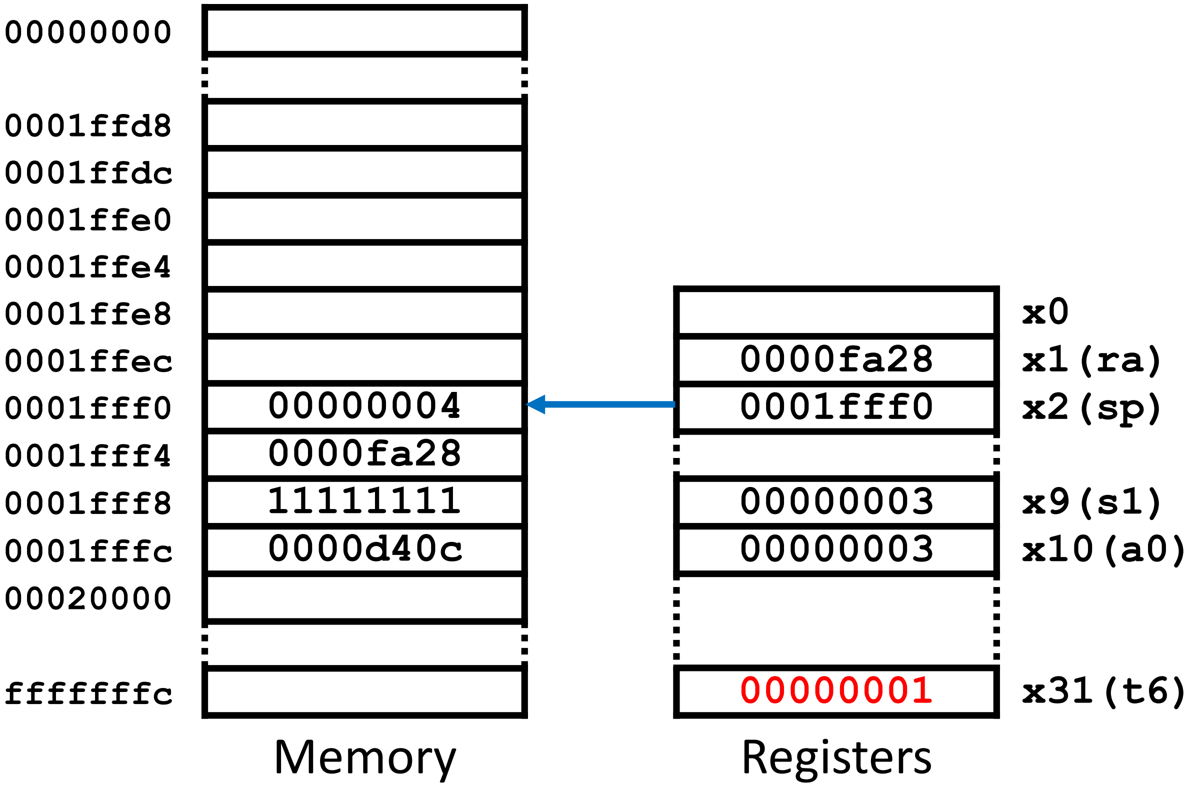


Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
     ...
     li   sp, 0x20000
     li   a0, 4
     call fact
0000d40c la   t6, a
     sw   a0, 0(t6)
     ...
fact:
     add  sp, sp, -8
     sw   ra, 4(sp)
     sw   s1, 0(sp)
     mv   s1, a0
     li   t6, 1
     bgt  s1, t6, else
     li   a0, 1
     j    eif
else:
     add  a0, s1, -1
     call fact
0000fa28 mul  a0, s1, a0
eif:
     lw   ra, 4(sp)
     lw   s1, 0(sp)
     add  sp, sp, 8
     ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1 (ra)
0001fff0	x2 (sp)
00000003	x9 (s1)
00000003	x10 (a0)
00000001	x31 (t6)

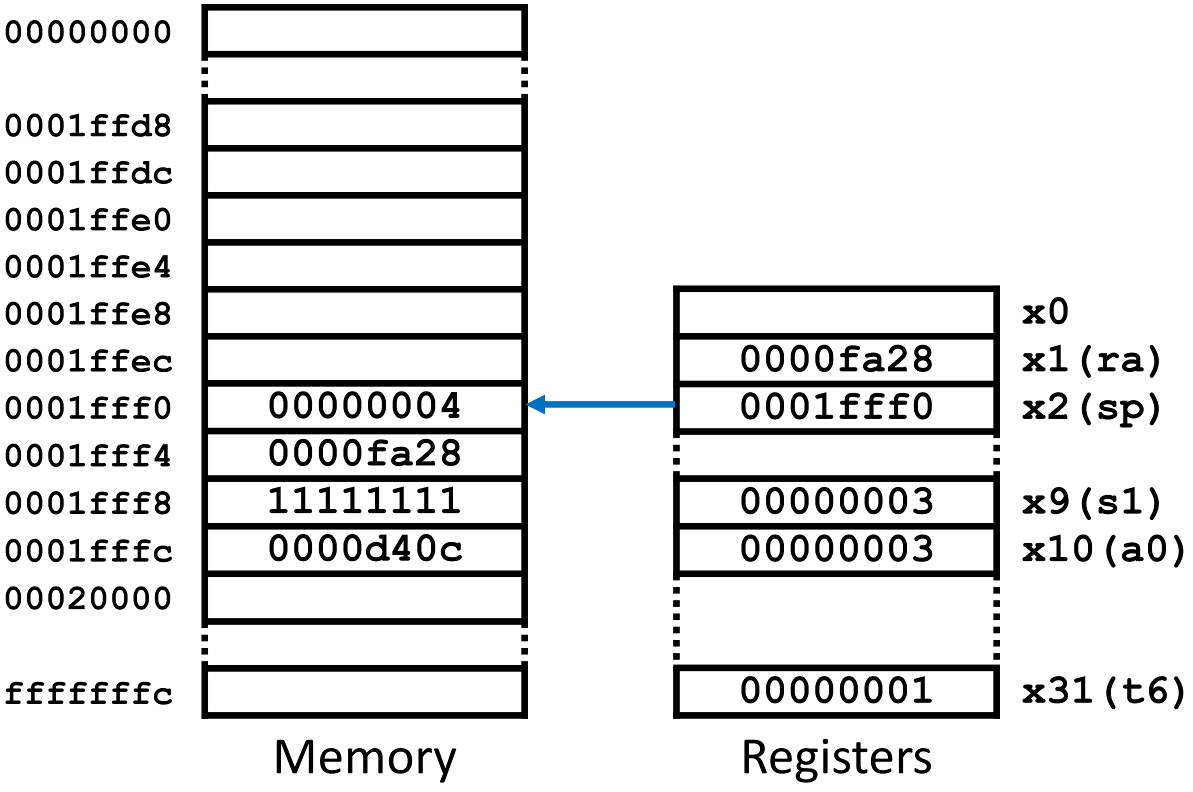
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
    ...
    li    sp, 0x20000
    li    a0, 4
    call  fact
0000d40c la    t6, a
    sw    a0, 0(t6)
    ...
fact:
    add   sp, sp, -8
    sw    ra, 4(sp)
    sw    s1, 0(sp)
    mv    s1, a0
    li    t6, 1
  ⇨ bgt   s1, t6, else
    li    a0, 1
    j     eif
else:
    add   a0, s1, -1
    call  fact
0000fa28 mul   a0, s1, a0
eif:
    lw    ra, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 8
    ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001fff0	x2(sp)
00000003	x9(s1)
00000003	x10(a0)
00000001	x31(t6)

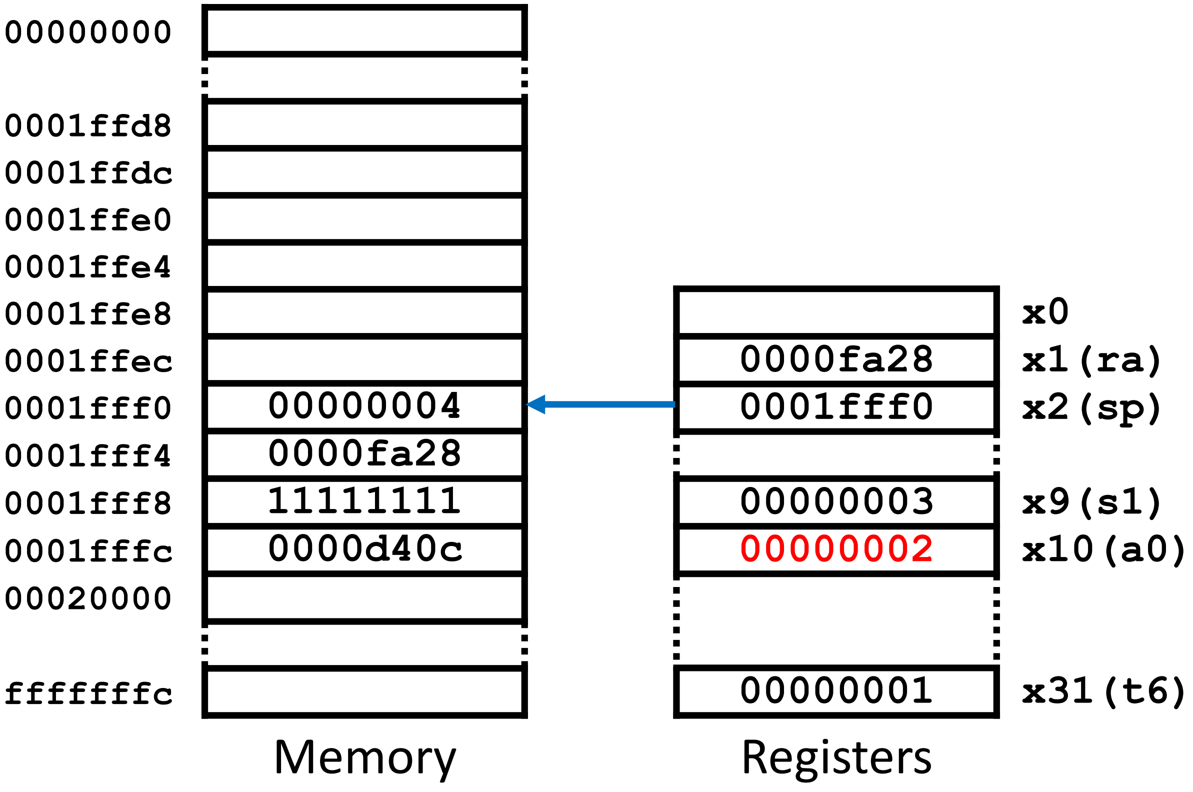
Registers

Functions

Nesting and recursion (iv)

ASM

```
         a:  .space 4
              ...
              li    sp, 0x20000
              li    a0, 4
              call  fact
0000d40c      la    t6, a
              sw    a0, 0(t6)
              ...
         fact:
              add   sp, sp, -8
              sw    ra, 4(sp)
              sw    s1, 0(sp)
              mv    s1, a0
              li    t6, 1
              bgt   s1, t6, else
              li    a0, 1
              j     eif
         else:
            ⇨ add   a0, s1, -1
              call  fact
0000fa28      mul   a0, s1, a0
         eif:
              lw    ra, 4(sp)
              lw    s1, 0(sp)
              add   sp, sp, 8
              ret
```



Memory

00000000	
⋮	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
⋮	
fffffffc	

Registers

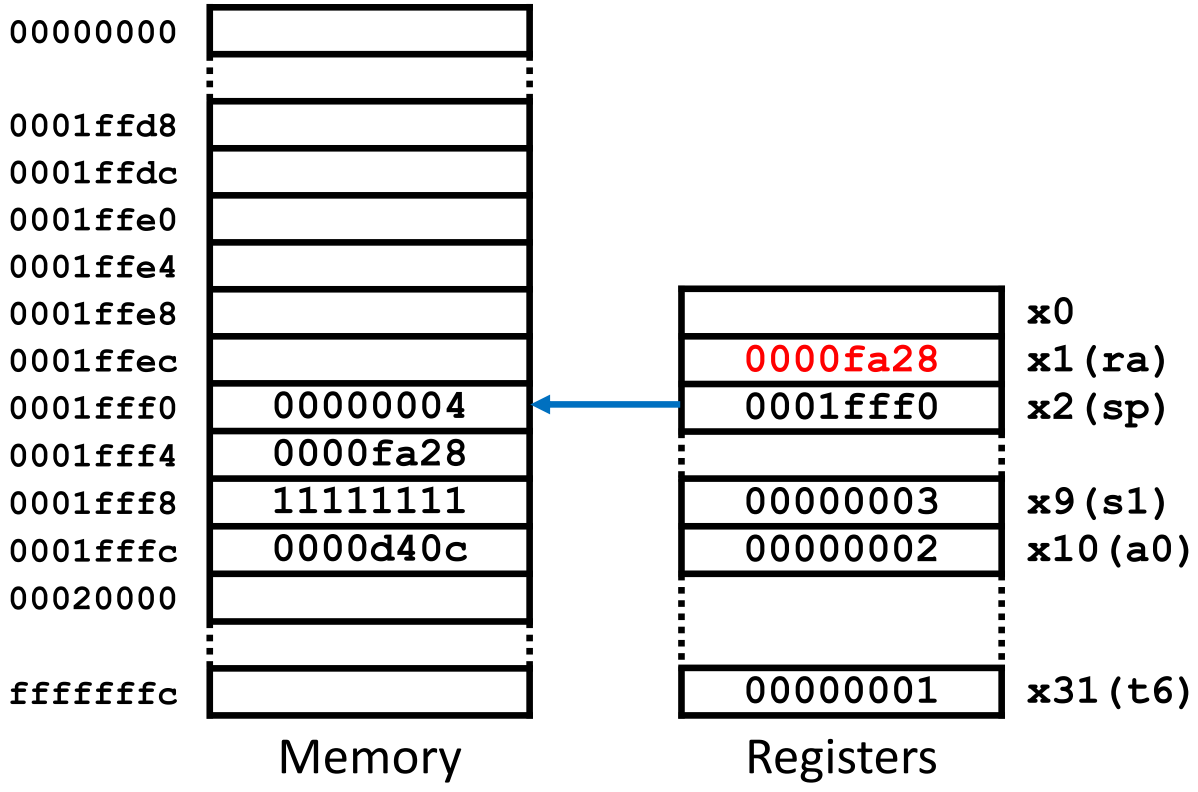
	x0
0000fa28	x1(ra)
0001fff0	x2(sp)
⋮	
00000003	x9(s1)
00000002	x10(a0)
⋮	
00000001	x31(t6)

Functions

Nesting and recursion (iv)

ASM

```
a: .space 4
          ...
          li   sp, 0x20000
          li   a0, 4
          call fact
0000d40c  la   t6, a
          sw   a0, 0(t6)
          ...
fact:
          add  sp, sp, -8
          sw   ra, 4(sp)
          sw   s1, 0(sp)
          mv   s1, a0
          li   t6, 1
          bgt  s1, t6, else
          li   a0, 1
          j    eif
else:
          add  a0, s1, -1
       ⇨  call fact
0000fa28  mul  a0, s1, a0
eif:
          lw   ra, 4(sp)
          lw   s1, 0(sp)
          add  sp, sp, 8
          ret
```



Memory

00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Registers

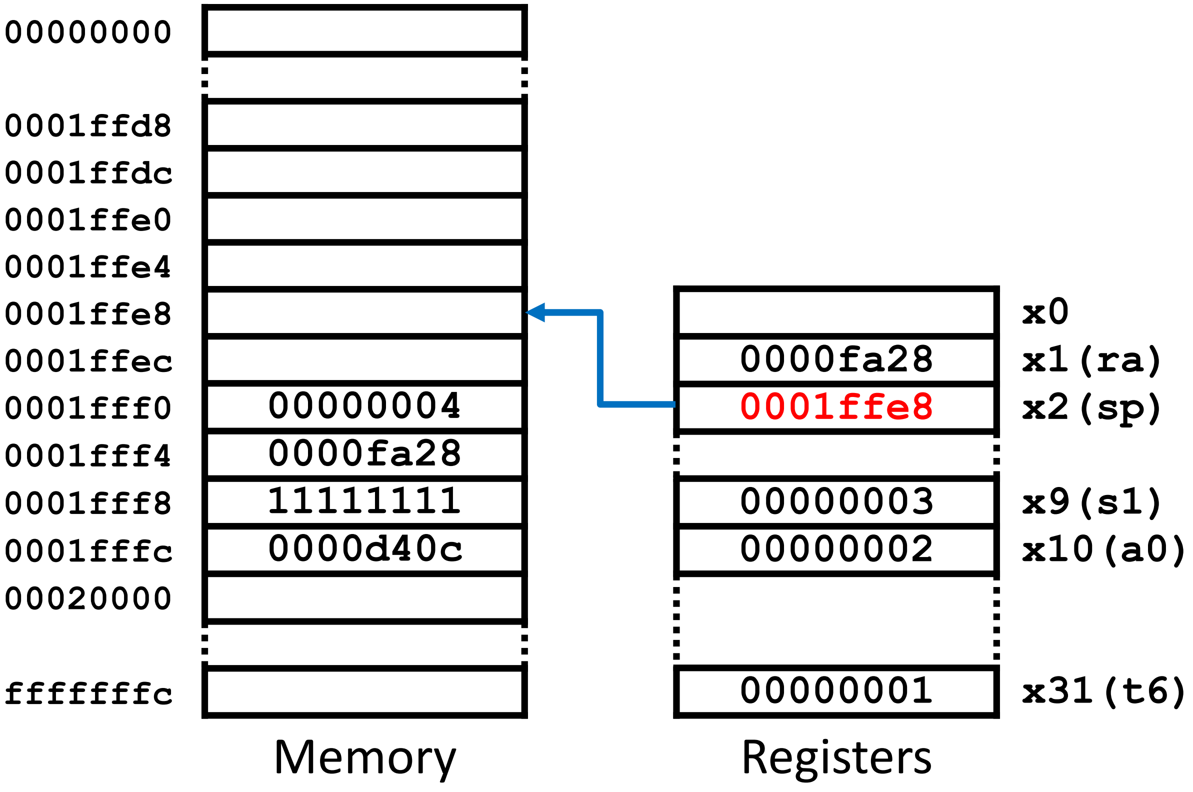
	x0
0000fa28	x1(ra)
0001fff0	x2(sp)
00000003	x9(s1)
00000002	x10(a0)
00000001	x31(t6)

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
    ...
    li    sp, 0x20000
    li    a0, 4
    call  fact
0000d40c la    t6, a
    sw    a0, 0(t6)
    ...
fact:
    add   sp, sp, -8
    sw    ra, 4(sp)
    sw    s1, 0(sp)
    mv    s1, a0
    li    t6, 1
    bgt   s1, t6, else
    li    a0, 1
    j     eif
else:
    add   a0, s1, -1
    call  fact
0000fa28 mul   a0, s1, a0
eif:
    lw    ra, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 8
    ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1 (ra)
0001ffe8	x2 (sp)
00000003	x9 (s1)
00000002	x10 (a0)
00000001	x31 (t6)

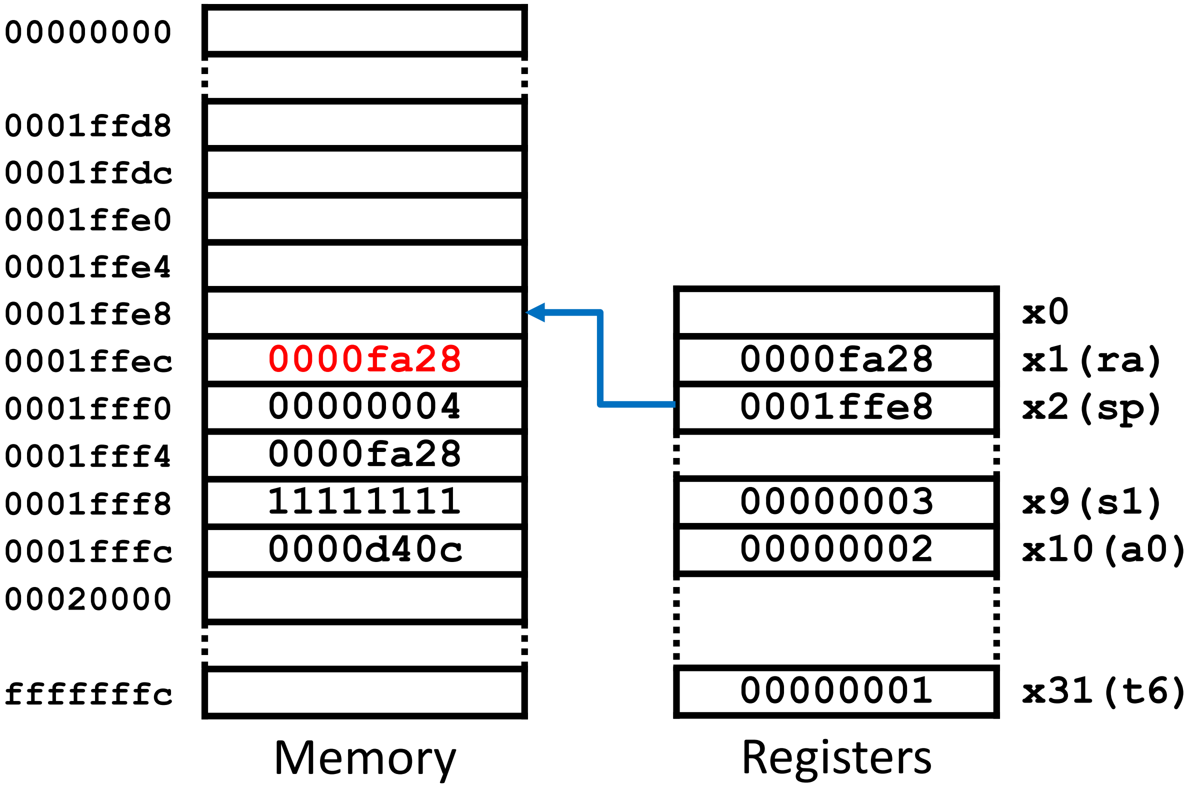
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
    ...
    li    sp, 0x20000
    li    a0, 4
    call  fact
0000d40c la    t6, a
    sw    a0, 0(t6)
    ...
fact:
    add   sp, sp, -8
    sw    ra, 4(sp)
    sw    s1, 0(sp)
    mv    s1, a0
    li    t6, 1
    bgt   s1, t6, else
    li    a0, 1
    j     eif
else:
    add   a0, s1, -1
    call  fact
0000fa28 mul   a0, s1, a0
eif:
    lw    ra, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 8
    ret
```

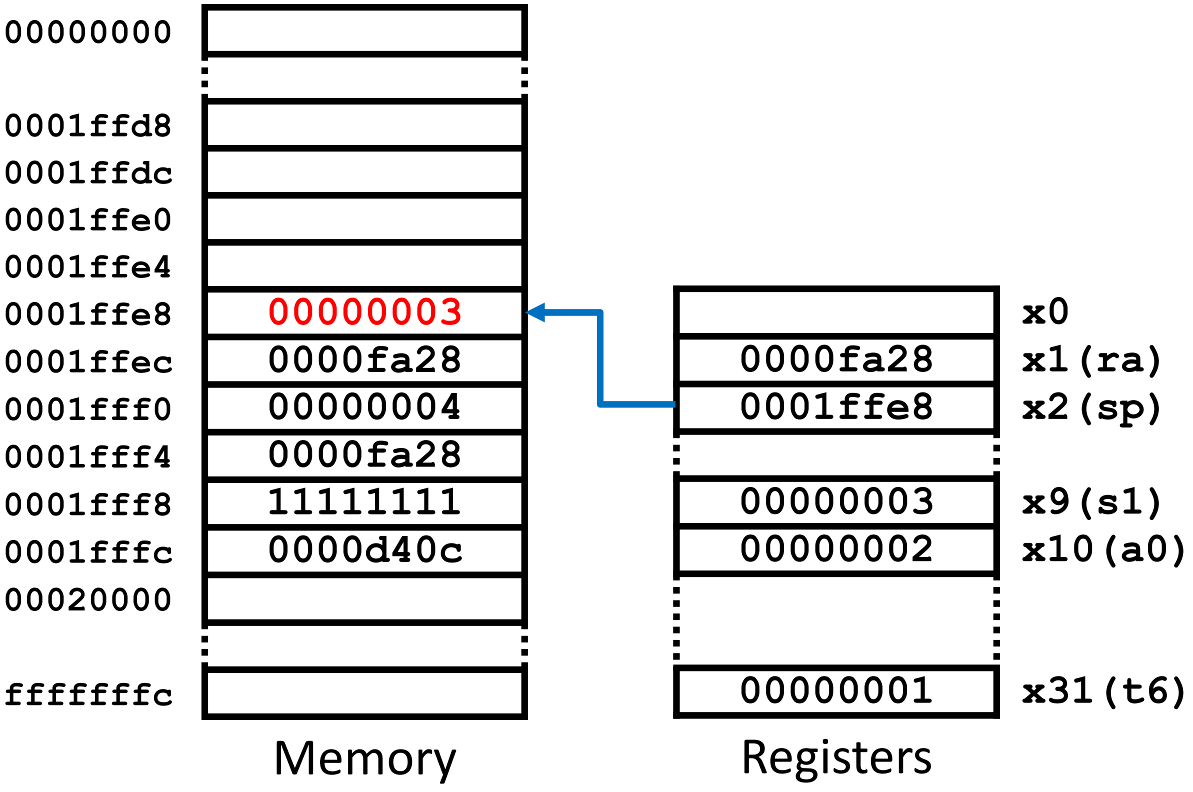


Functions

Nesting and recursion (iv)

ASM

```
         a:  .space 4
             ...
             li   sp, 0x20000
             li   a0, 4
             call fact
0000d40c     la   t6, a
             sw   a0, 0(t6)
             ...
         fact:
             add  sp, sp, -8
             sw   ra, 4(sp)
             sw   s1, 0(sp)
             mv   s1, a0
             li   t6, 1
             bgt  s1, t6, else
             li   a0, 1
             j    eif
         else:
             add  a0, s1, -1
             call fact
0000fa28     mul  a0, s1, a0
         eif:
             lw   ra, 4(sp)
             lw   s1, 0(sp)
             add  sp, sp, 8
             ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe8	x2(sp)
00000003	x9(s1)
00000002	x10(a0)
00000001	x31(t6)

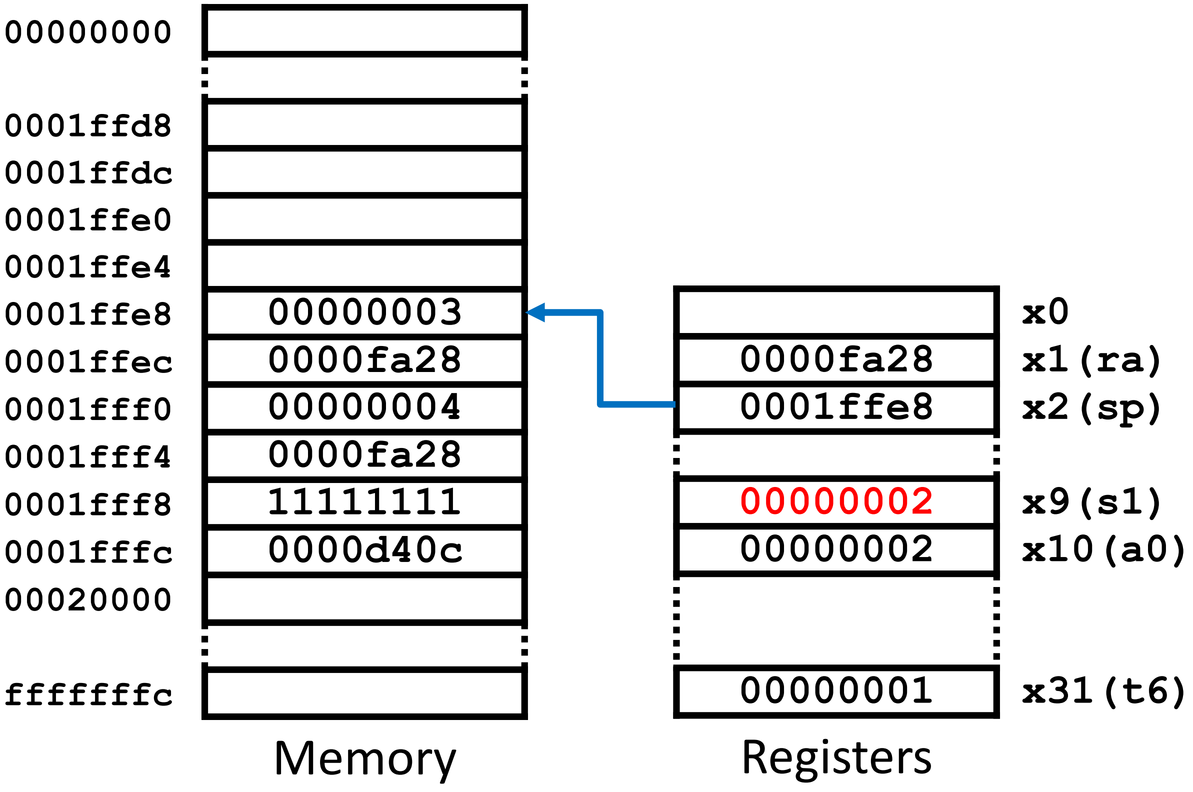
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
    ...
    li    sp, 0x20000
    li    a0, 4
    call  fact
0000d40c la    t6, a
    sw    a0, 0(t6)
    ...
fact:
    add   sp, sp, -8
    sw    ra, 4(sp)
    sw    s1, 0(sp)
    mv    s1, a0
    li    t6, 1
    bgt   s1, t6, else
    li    a0, 1
    j     eif
else:
    add   a0, s1, -1
    call  fact
0000fa28 mul   a0, s1, a0
eif:
    lw    ra, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 8
    ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe8	x2(sp)
00000002	x9(s1)
00000002	x10(a0)
00000001	x31(t6)

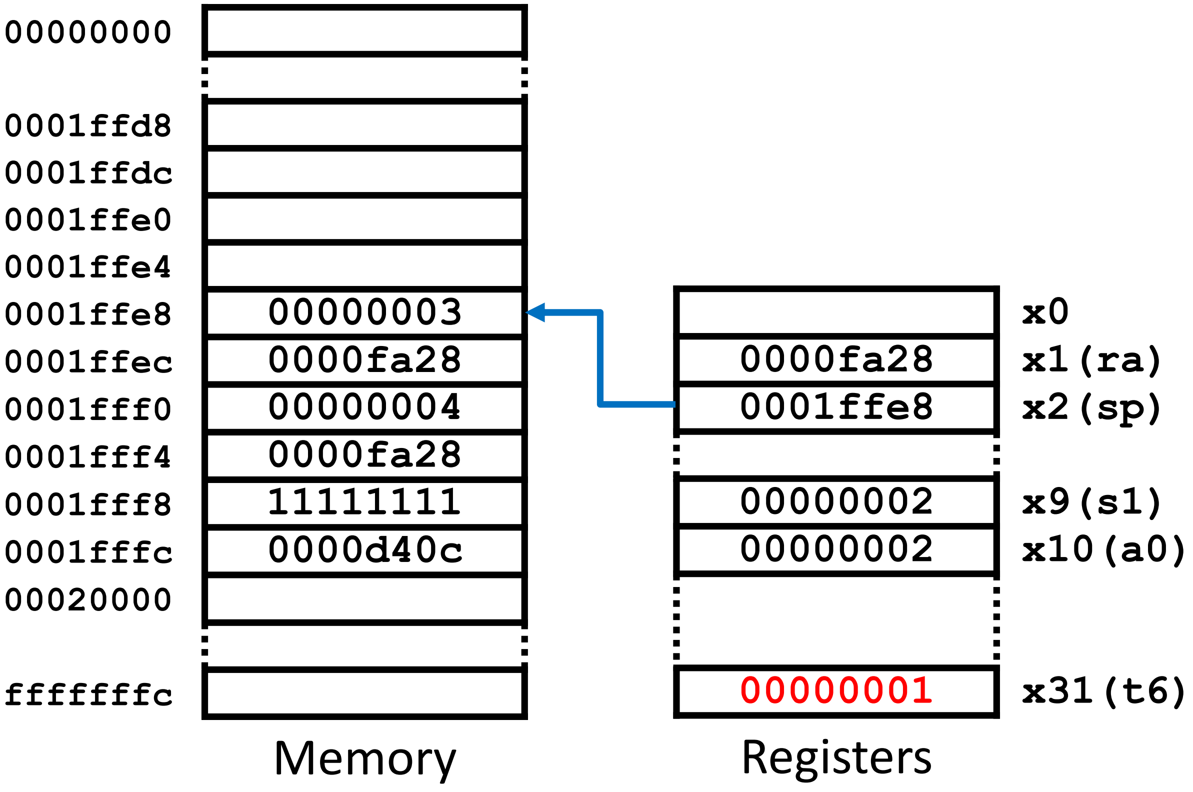
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
    ...
    li    sp, 0x20000
    li    a0, 4
    call  fact
0000d40c la    t6, a
    sw    a0, 0(t6)
    ...
fact:
    add   sp, sp, -8
    sw    ra, 4(sp)
    sw    s1, 0(sp)
    mv    s1, a0
    li    t6, 1
    bgt   s1, t6, else
    li    a0, 1
    j     eif
else:
    add   a0, s1, -1
    call  fact
0000fa28 mul   a0, s1, a0
eif:
    lw    ra, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 8
    ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1 (ra)
0001ffe8	x2 (sp)
00000002	x9 (s1)
00000002	x10 (a0)
00000001	x31 (t6)

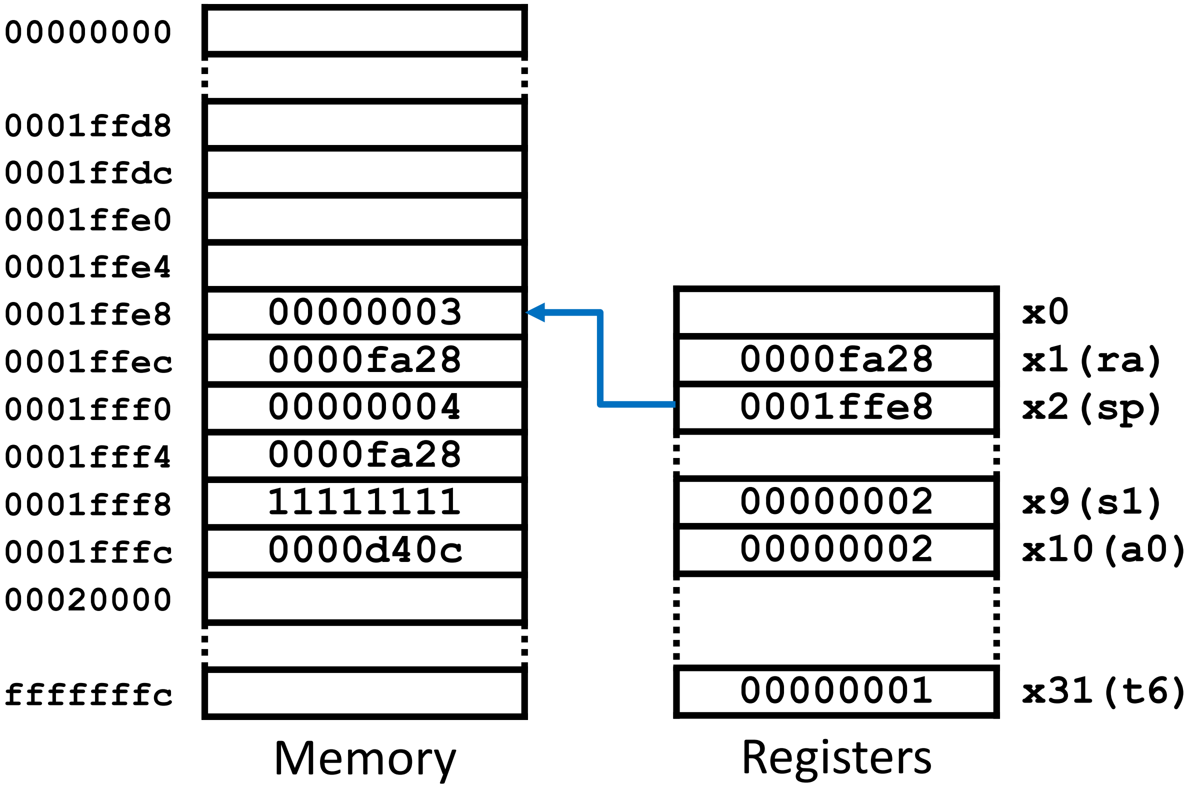
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
      ...
      li   sp, 0x20000
      li   a0, 4
      call fact
0000d40c la   t6, a
      sw   a0, 0(t6)
      ...
fact:
      add  sp, sp, -8
      sw   ra, 4(sp)
      sw   s1, 0(sp)
      mv   s1, a0
      li   t6, 1
      bgt  s1, t6, else
      li   a0, 1
      j    eif
else:
      add  a0, s1, -1
      call fact
0000fa28 mul  a0, s1, a0
eif:
      lw   ra, 4(sp)
      lw   s1, 0(sp)
      add  sp, sp, 8
      ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe8	x2(sp)
00000002	x9(s1)
00000002	x10(a0)
00000001	x31(t6)

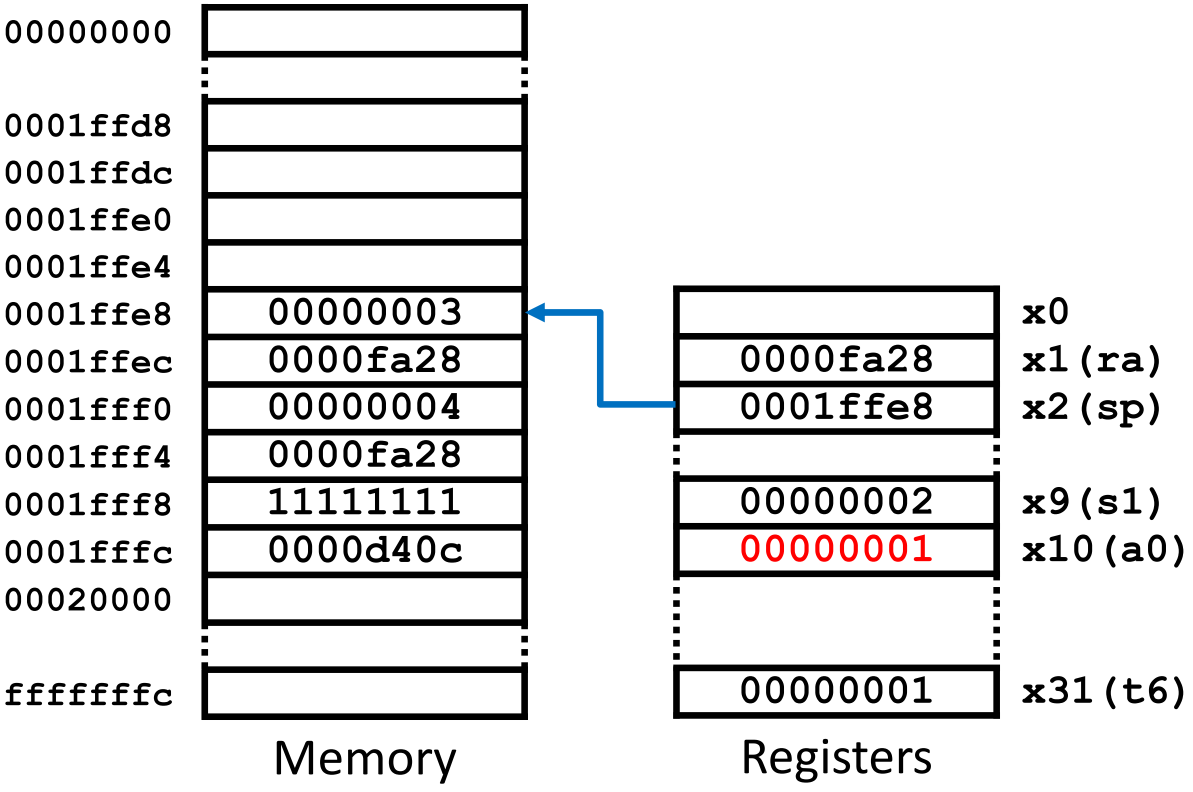
Registers

Functions

Nesting and recursion (iv)

ASM

```
a: .space 4
    ...
    li   sp, 0x20000
    li   a0, 4
    call fact
0000d40c la   t6, a
    sw   a0, 0(t6)
    ...
fact:
    add  sp, sp, -8
    sw   ra, 4(sp)
    sw   s1, 0(sp)
    mv   s1, a0
    li   t6, 1
    bgt  s1, t6, else
    li   a0, 1
    j    eif
else:
    add  a0, s1, -1
    call fact
0000fa28 mul  a0, s1, a0
eif:
    lw   ra, 4(sp)
    lw   s1, 0(sp)
    add  sp, sp, 8
    ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe8	x2(sp)
00000002	x9(s1)
00000001	x10(a0)
00000001	x31(t6)

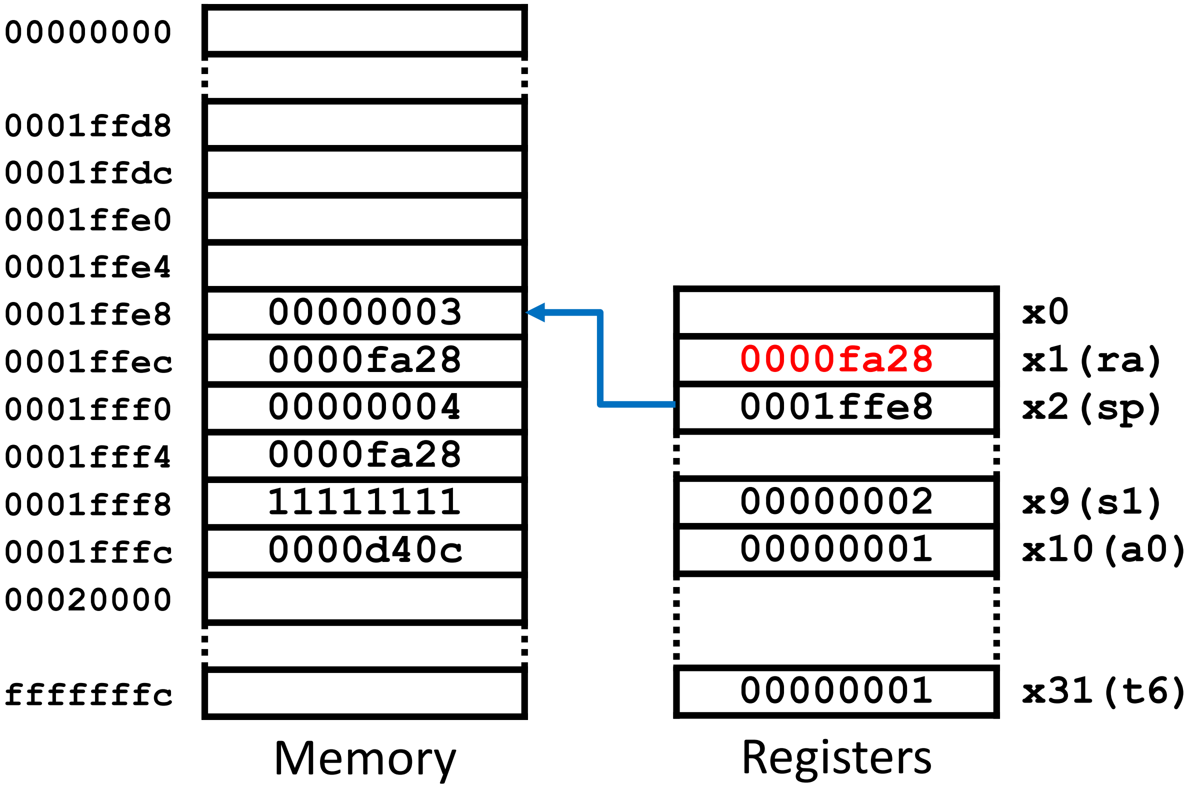
Registers

Functions

Nesting and recursion (iv)

ASM

```
a: .space 4
   ...
   li   sp, 0x20000
   li   a0, 4
   call fact
0000d40c la   t6, a
   sw   a0, 0(t6)
   ...
fact:
   add  sp, sp, -8
   sw   ra, 4(sp)
   sw   s1, 0(sp)
   mv   s1, a0
   li   t6, 1
   bgt  s1, t6, else
   li   a0, 1
   j    eif
else:
   add  a0, s1, -1
   call fact
0000fa28 mul  a0, s1, a0
eif:
   lw   ra, 4(sp)
   lw   s1, 0(sp)
   add  sp, sp, 8
   ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe8	x2(sp)
00000002	x9(s1)
00000001	x10(a0)
00000001	x31(t6)

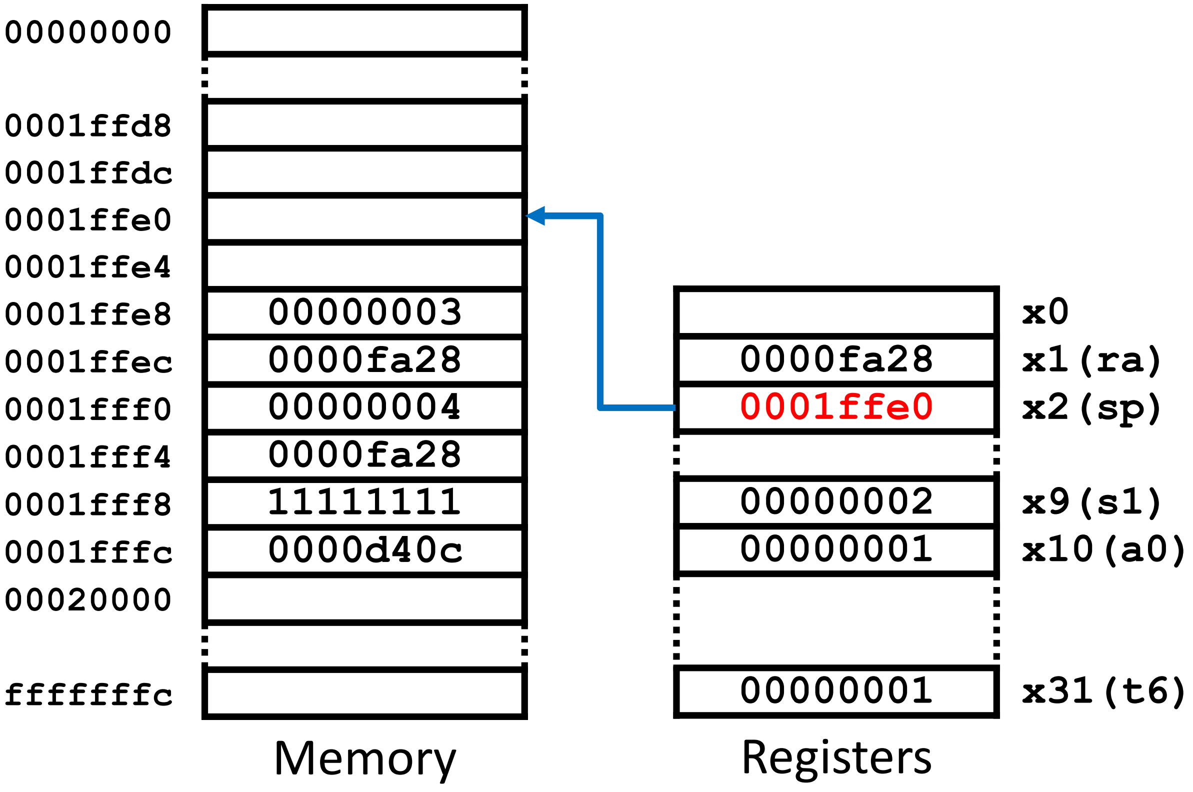
Registers

Functions

Nesting and recursion (iv)

ASM

```
a: .space 4
    ...
    li   sp, 0x20000
    li   a0, 4
    call fact
0000d40c la   t6, a
    sw   a0, 0(t6)
    ...
fact:
    add  sp, sp, -8
    sw   ra, 4(sp)
    sw   s1, 0(sp)
    mv   s1, a0
    li   t6, 1
    bgt  s1, t6, else
    li   a0, 1
    j    eif
else:
    add  a0, s1, -1
    call fact
0000fa28 mul  a0, s1, a0
eif:
    lw   ra, 4(sp)
    lw   s1, 0(sp)
    add  sp, sp, 8
    ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe0	x2(sp)
00000002	x9(s1)
00000001	x10(a0)
00000001	x31(t6)

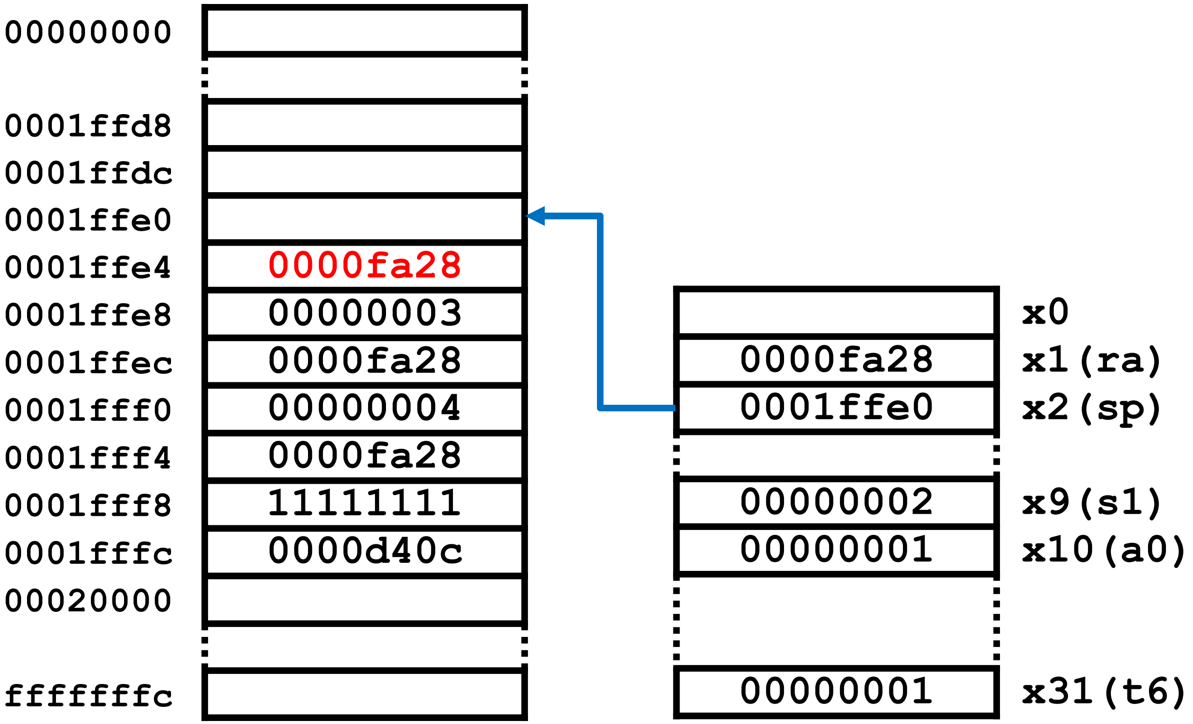
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
        ...
        li   sp, 0x20000
        li   a0, 4
        call fact
0000d40c la   t6, a
        sw   a0, 0(t6)
        ...
fact:
        add  sp, sp, -8
   ⇨    sw   ra, 4(sp)
        sw   s1, 0(sp)
        mv   s1, a0
        li   t6, 1
        bgt  s1, t6, else
        li   a0, 1
        j    eif
else:
        add  a0, s1, -1
        call fact
0000fa28 mul  a0, s1, a0
eif:
        lw   ra, 4(sp)
        lw   s1, 0(sp)
        add  sp, sp, 8
        ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe0	x2(sp)
00000002	x9(s1)
00000001	x10(a0)
00000001	x31(t6)

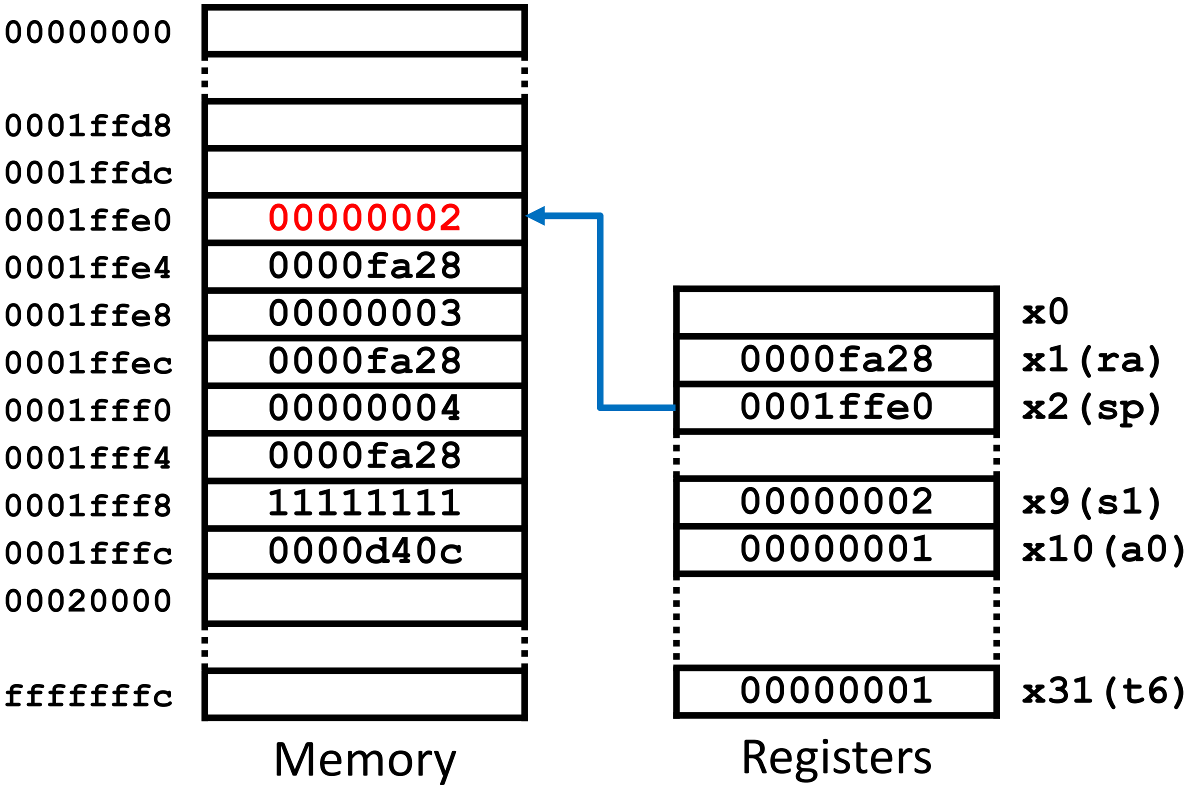
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
         ...
         li   sp, 0x20000
         li   a0, 4
         call fact
0000d40c la   t6, a
         sw   a0, 0(t6)
         ...
fact:
         add  sp, sp, -8
         sw   ra, 4(sp)
    ⇨    sw   s1, 0(sp)
         mv   s1, a0
         li   t6, 1
         bgt  s1, t6, else
         li   a0, 1
         j    eif
else:
         add  a0, s1, -1
         call fact
0000fa28 mul  a0, s1, a0
eif:
         lw   ra, 4(sp)
         lw   s1, 0(sp)
         add  sp, sp, 8
         ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1 (ra)
0001ffe0	x2 (sp)
00000002	x9 (s1)
00000001	x10 (a0)
00000001	x31 (t6)

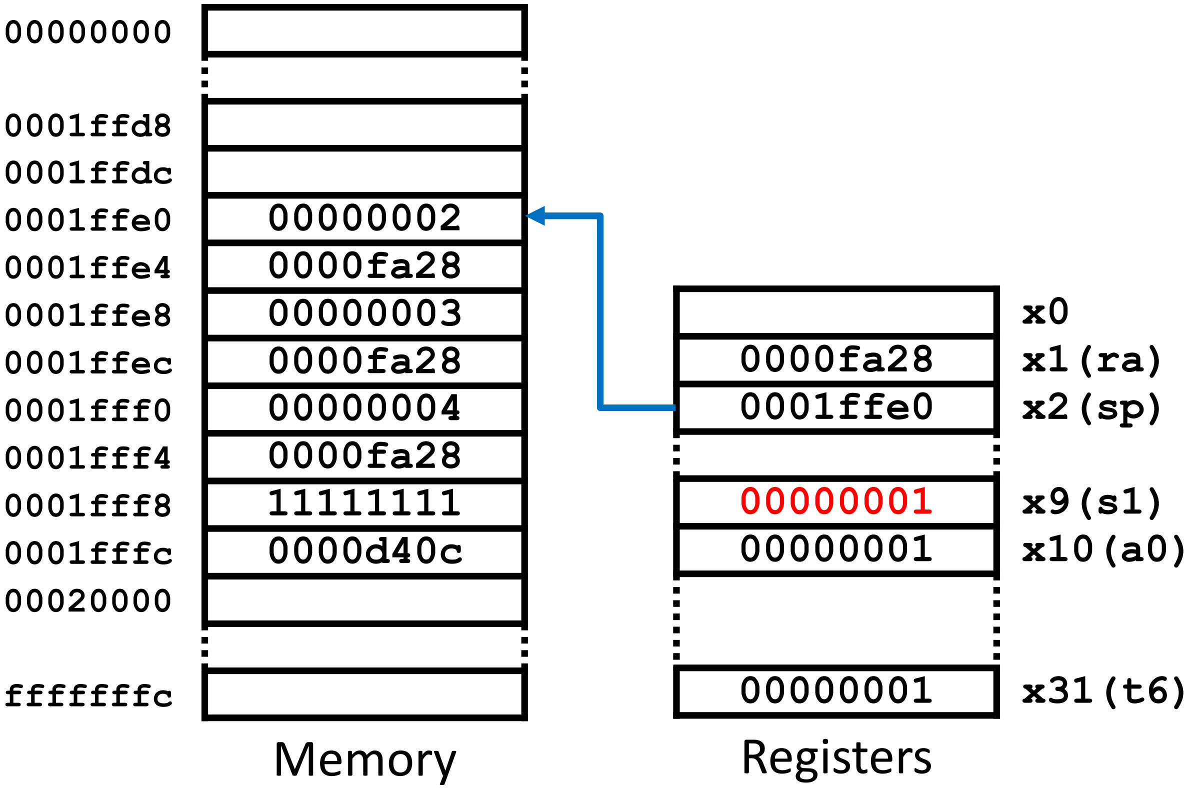
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
         ...
         li    sp, 0x20000
         li    a0, 4
         call  fact
0000d40c la    t6, a
         sw    a0, 0(t6)
         ...
fact:
         add   sp, sp, -8
         sw    ra, 4(sp)
         sw    s1, 0(sp)
         mv    s1, a0
         li    t6, 1
         bgt   s1, t6, else
         li    a0, 1
         j     eif
else:
         add   a0, s1, -1
         call  fact
0000fa28 mul   a0, s1, a0
eif:
         lw    ra, 4(sp)
         lw    s1, 0(sp)
         add   sp, sp, 8
         ret
```



Memory

00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Registers

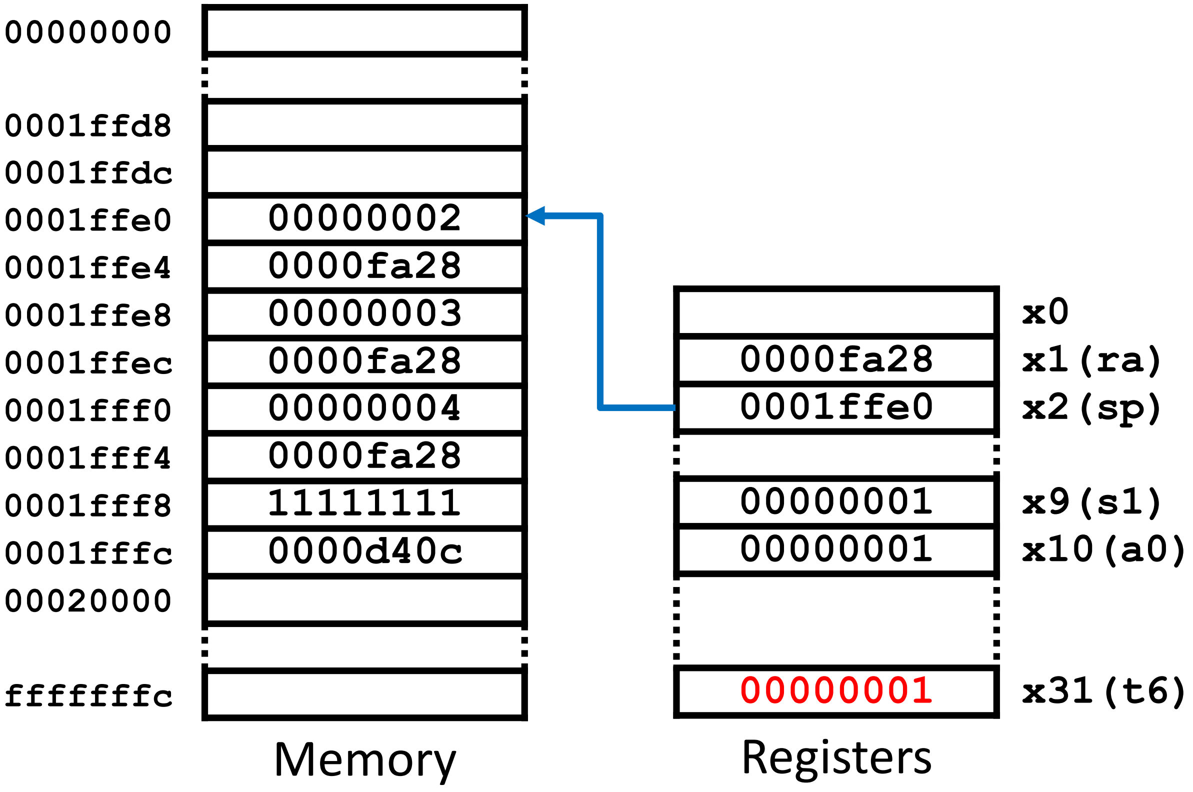
	x0
0000fa28	x1(ra)
0001ffe0	x2(sp)
00000001	x9(s1)
00000001	x10(a0)
00000001	x31(t6)

Functions

Nesting and recursion (iv)

ASM

```
          a:  .space 4
                ...
                li   sp, 0x20000
                li   a0, 4
                call fact
0000d40c        la   t6, a
                sw   a0, 0(t6)
                ...
          fact:
                add  sp, sp, -8
                sw   ra, 4(sp)
                sw   s1, 0(sp)
                mv   s1, a0
       ⇨        li   t6, 1
                bgt  s1, t6, else
                li   a0, 1
                j    eif
          else:
                add  a0, s1, -1
                call fact
0000fa28        mul  a0, s1, a0
          eif:
                lw   ra, 4(sp)
                lw   s1, 0(sp)
                add  sp, sp, 8
                ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe0	x2(sp)
00000001	x9(s1)
00000001	x10(a0)
00000001	x31(t6)

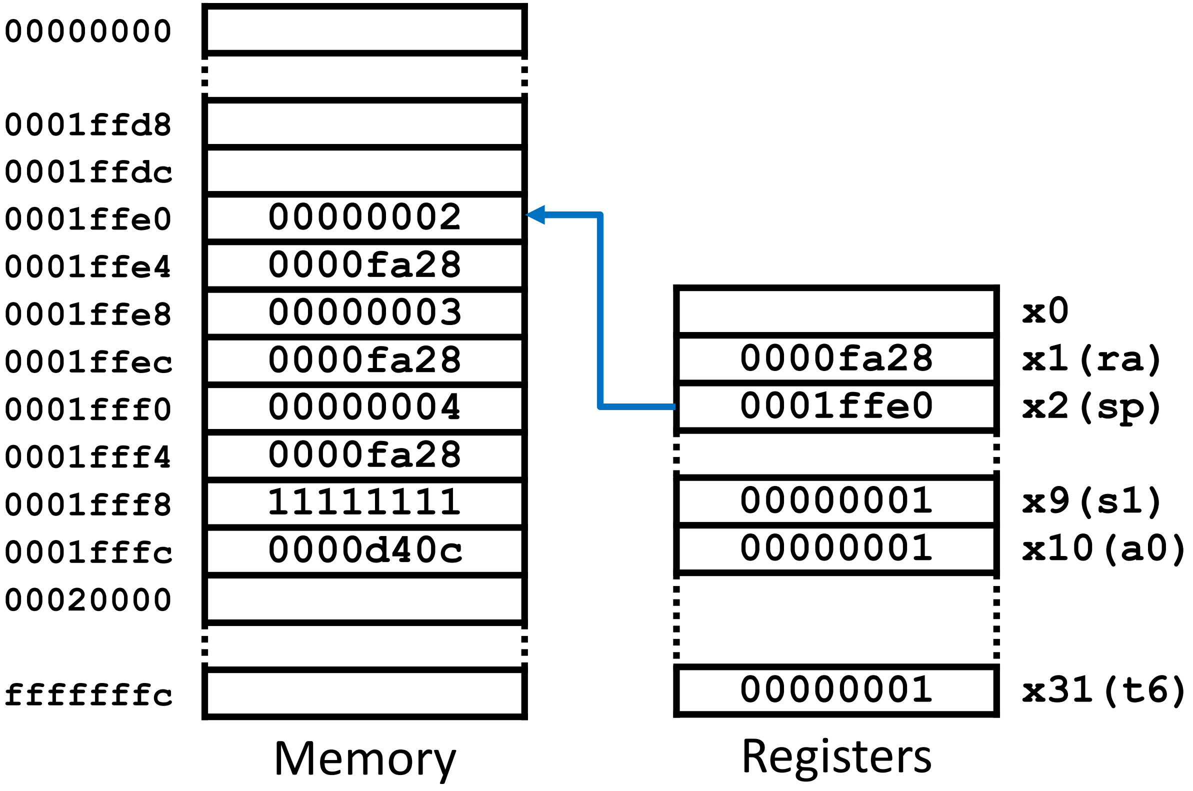
Registers

Functions

Nesting and recursion (iv)

ASM

```
         a:  .space 4
                 ...
                 li   sp, 0x20000
                 li   a0, 4
                 call fact
0000d40c         la   t6, a
                 sw   a0, 0(t6)
                 ...
         fact:
                 add  sp, sp, -8
                 sw   ra, 4(sp)
                 sw   s1, 0(sp)
                 mv   s1, a0
                 li   t6, 1
      ⇨          bgt  s1, t6, else
                 li   a0, 1
                 j    eif
         else:
                 add  a0, s1, -1
                 call fact
0000fa28         mul  a0, s1, a0
         eif:
                 lw   ra, 4(sp)
                 lw   s1, 0(sp)
                 add  sp, sp, 8
                 ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

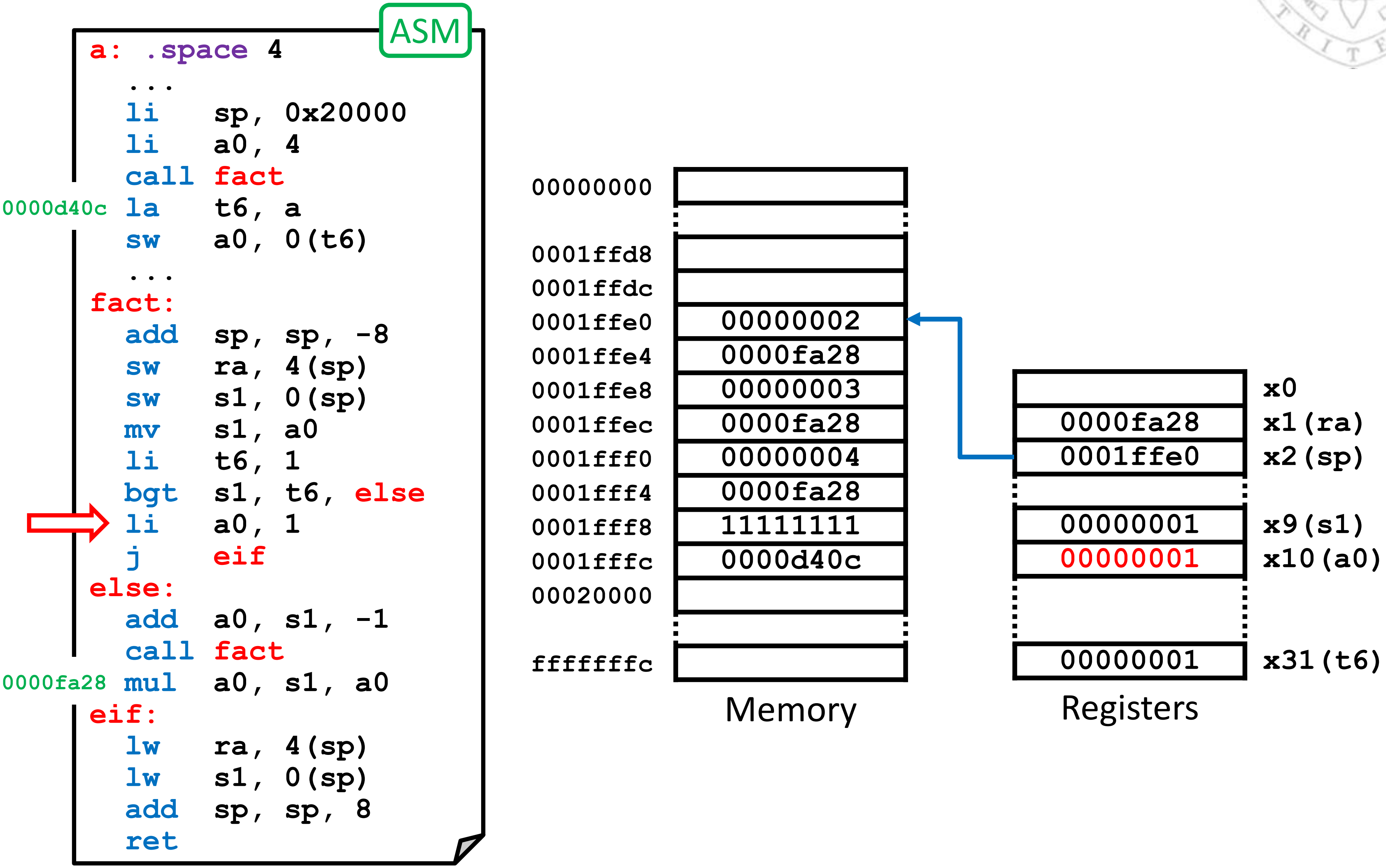
Memory

	x0
0000fa28	x1(ra)
0001ffe0	x2(sp)
00000001	x9(s1)
00000001	x10(a0)
00000001	x31(t6)

Registers

Functions

Nesting and recursion (iv)



ASM

```
a: .space 4
   ...
   li    sp, 0x20000
   li    a0, 4
   call  fact
0000d40c la    t6, a
   sw    a0, 0(t6)
   ...
fact:
   add   sp, sp, -8
   sw    ra, 4(sp)
   sw    s1, 0(sp)
   mv    s1, a0
   li    t6, 1
   bgt   s1, t6, else
   li    a0, 1
   j     eif
else:
   add   a0, s1, -1
   call  fact
0000fa28 mul   a0, s1, a0
eif:
   lw    ra, 4(sp)
   lw    s1, 0(sp)
   add   sp, sp, 8
   ret
```

00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe0	x2(sp)
00000001	x9(s1)
00000001	x10(a0)
00000001	x31(t6)

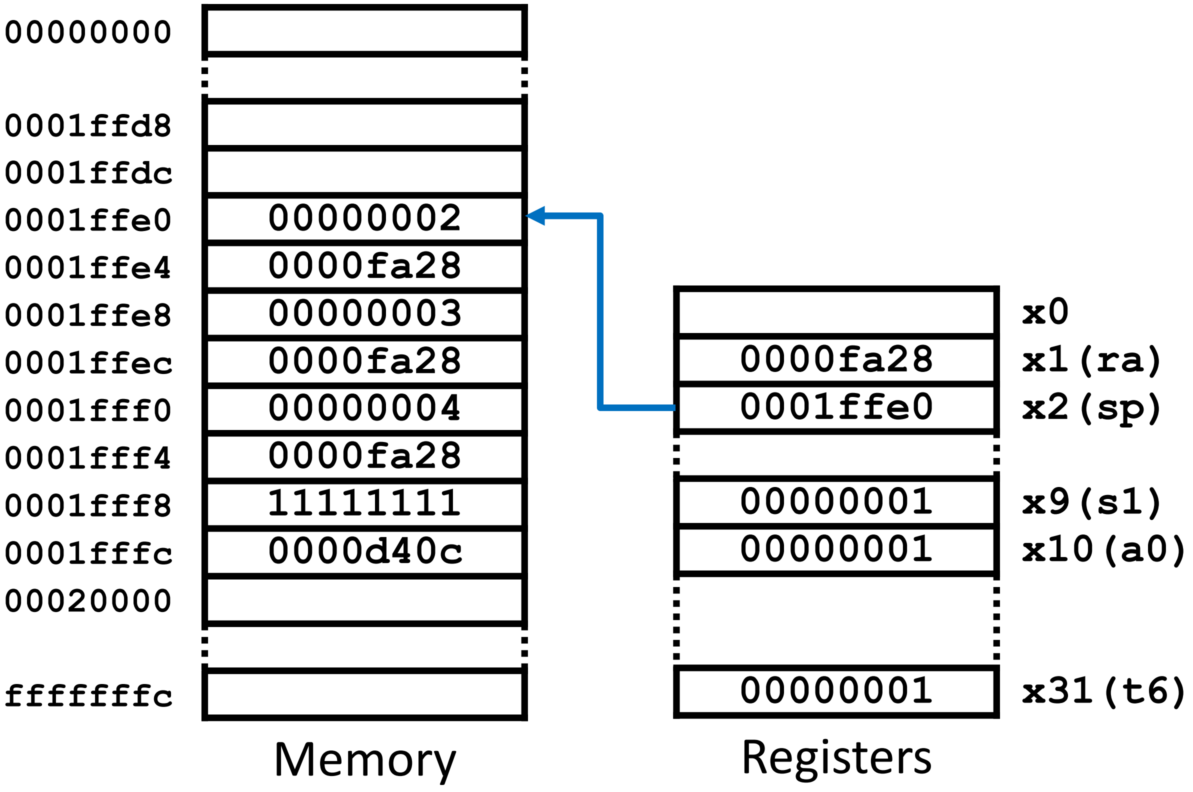
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
    ...
    li   sp, 0x20000
    li   a0, 4
    call fact
0000d40c la   t6, a
    sw   a0, 0(t6)
    ...
fact:
    add  sp, sp, -8
    sw   ra, 4(sp)
    sw   s1, 0(sp)
    mv   s1, a0
    li   t6, 1
    bgt  s1, t6, else
    li   a0, 1
    j    eif
else:
    add  a0, s1, -1
    call fact
0000fa28 mul  a0, s1, a0
eif:
    lw   ra, 4(sp)
    lw   s1, 0(sp)
    add  sp, sp, 8
    ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe0	x2(sp)
00000001	x9(s1)
00000001	x10(a0)
00000001	x31(t6)

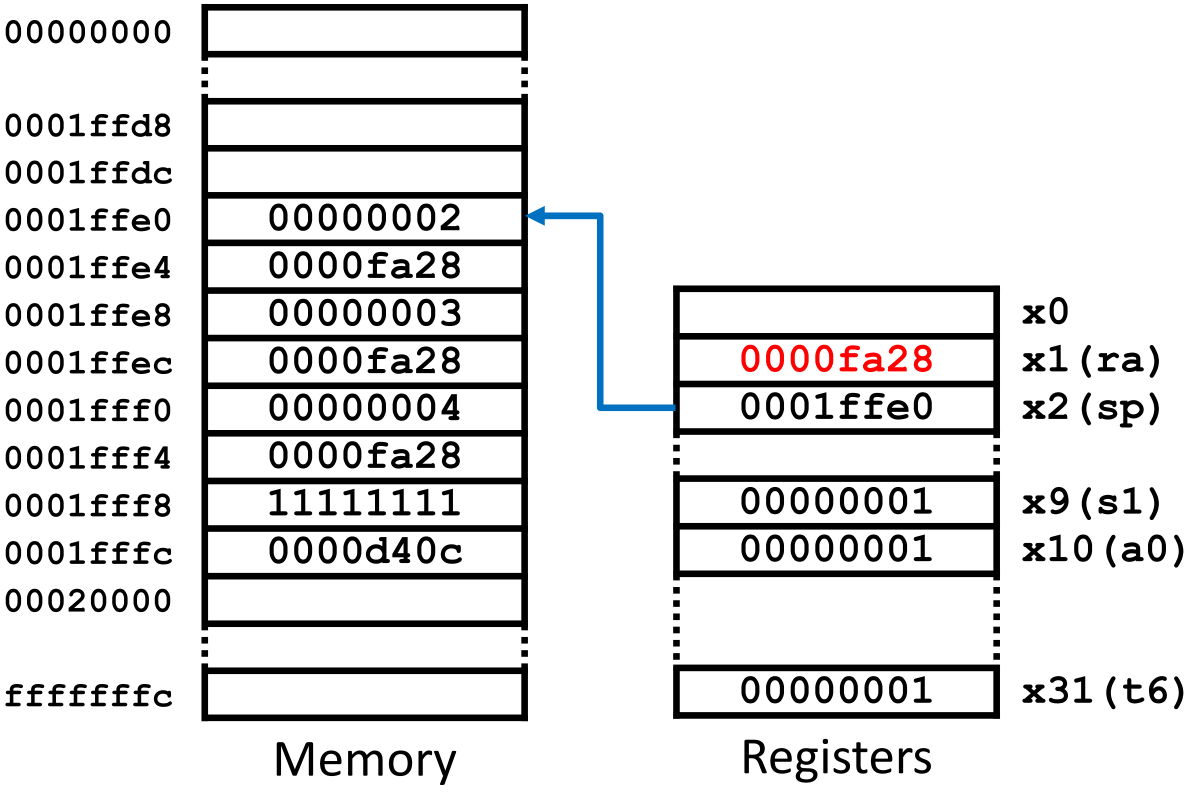
Registers

Functions

Nesting and recursion (iv)

ASM

```
         a:  .space 4
             ...
             li    sp, 0x20000
             li    a0, 4
             call  fact
0000d40c     la    t6, a
             sw    a0, 0(t6)
             ...
         fact:
             add   sp, sp, -8
             sw    ra, 4(sp)
             sw    s1, 0(sp)
             mv    s1, a0
             li    t6, 1
             bgt   s1, t6, else
             li    a0, 1
             j     eif
         else:
             add   a0, s1, -1
             call  fact
0000fa28     mul   a0, s1, a0
         eif:
      ⇒      lw    ra, 4(sp)
             lw    s1, 0(sp)
             add   sp, sp, 8
             ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe0	x2(sp)
00000001	x9(s1)
00000001	x10(a0)
00000001	x31(t6)

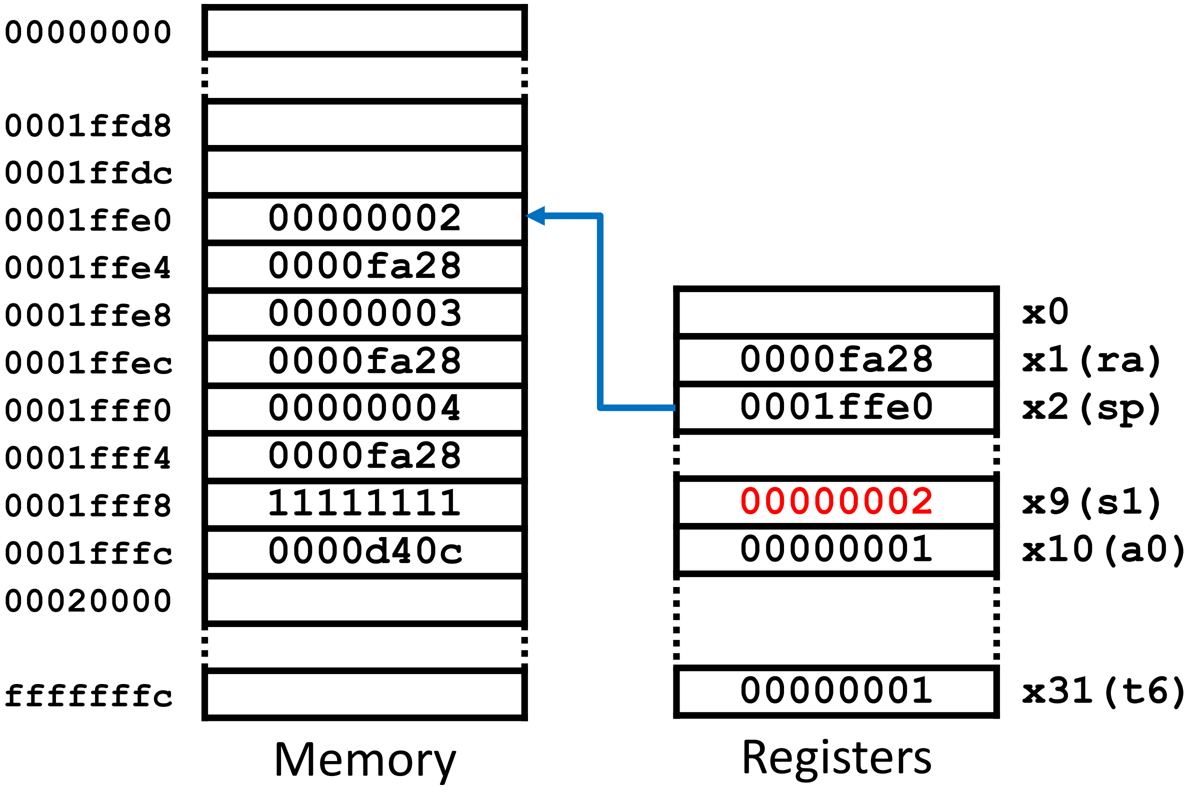
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
    ...
    li    sp, 0x20000
    li    a0, 4
    call  fact
0000d40c la    t6, a
    sw    a0, 0(t6)
    ...
fact:
    add   sp, sp, -8
    sw    ra, 4(sp)
    sw    s1, 0(sp)
    mv    s1, a0
    li    t6, 1
    bgt   s1, t6, else
    li    a0, 1
    j     eif
else:
    add   a0, s1, -1
    call  fact
0000fa28 mul   a0, s1, a0
eif:
    lw    ra, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 8
    ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe0	x2(sp)
00000002	x9(s1)
00000001	x10(a0)
00000001	x31(t6)

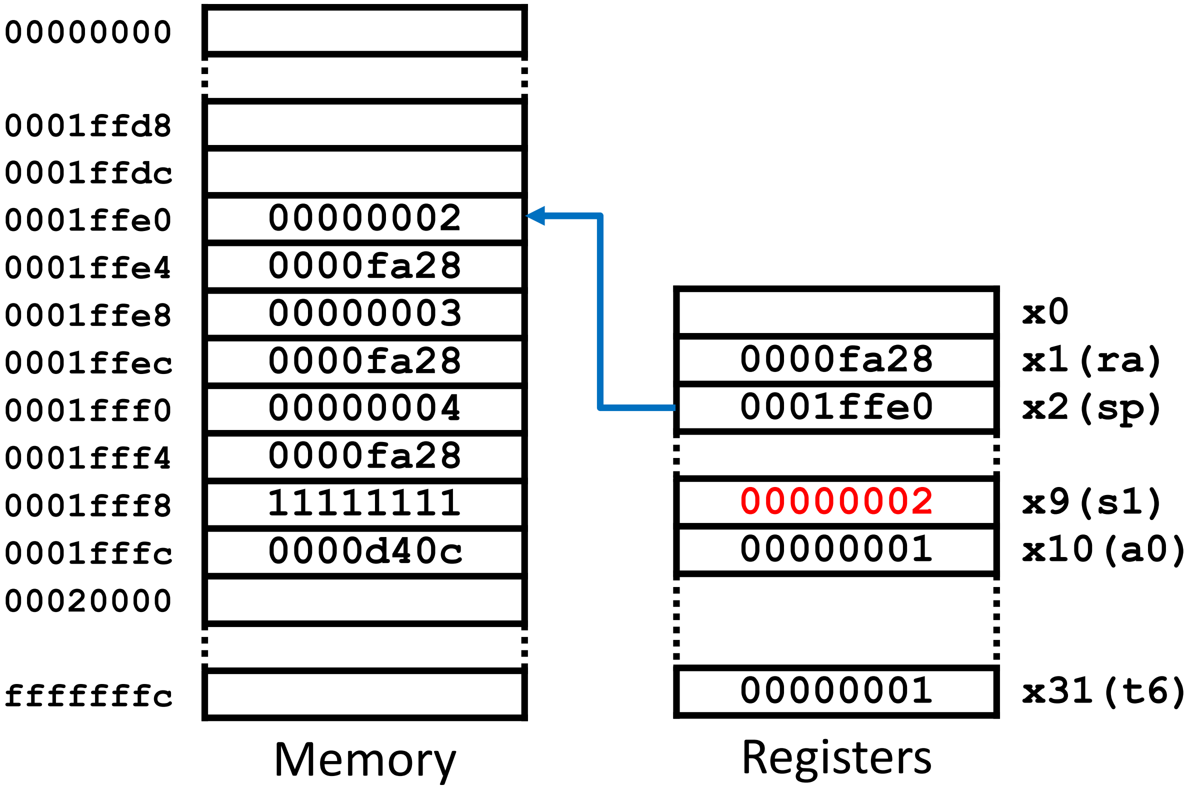
Registers

Functions

Nesting and recursion (iv)

ASM

```
         a:  .space 4
               ...
               li   sp, 0x20000
               li   a0, 4
               call fact
0000d40c       la   t6, a
               sw   a0, 0(t6)
               ...
         fact:
               add  sp, sp, -8
               sw   ra, 4(sp)
               sw   s1, 0(sp)
               mv   s1, a0
               li   t6, 1
               bgt  s1, t6, else
               li   a0, 1
               j    eif
         else:
               add  a0, s1, -1
               call fact
0000fa28       mul  a0, s1, a0
         eif:
               lw   ra, 4(sp)
        ⇨      lw   s1, 0(sp)
               add  sp, sp, 8
               ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe0	x2(sp)
00000002	x9(s1)
00000001	x10(a0)
00000001	x31(t6)

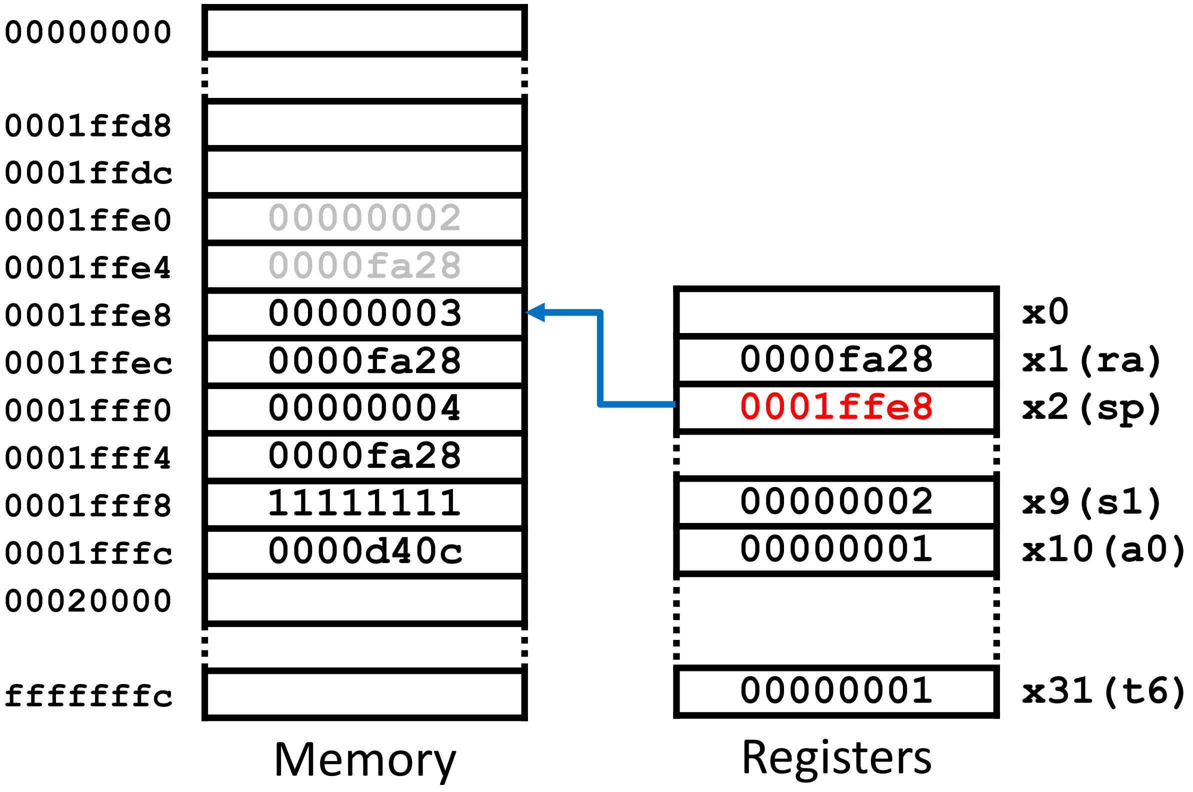
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
       ...
       li    sp, 0x20000
       li    a0, 4
       call  fact
0000d40c la    t6, a
       sw    a0, 0(t6)
       ...
fact:
       add   sp, sp, -8
       sw    ra, 4(sp)
       sw    s1, 0(sp)
       mv    s1, a0
       li    t6, 1
       bgt   s1, t6, else
       li    a0, 1
       j     eif
else:
       add   a0, s1, -1
       call  fact
0000fa28 mul   a0, s1, a0
eif:
       lw    ra, 4(sp)
       lw    s1, 0(sp)
    => add   sp, sp, 8
       ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe8	x2(sp)
00000002	x9(s1)
00000001	x10(a0)
00000001	x31(t6)

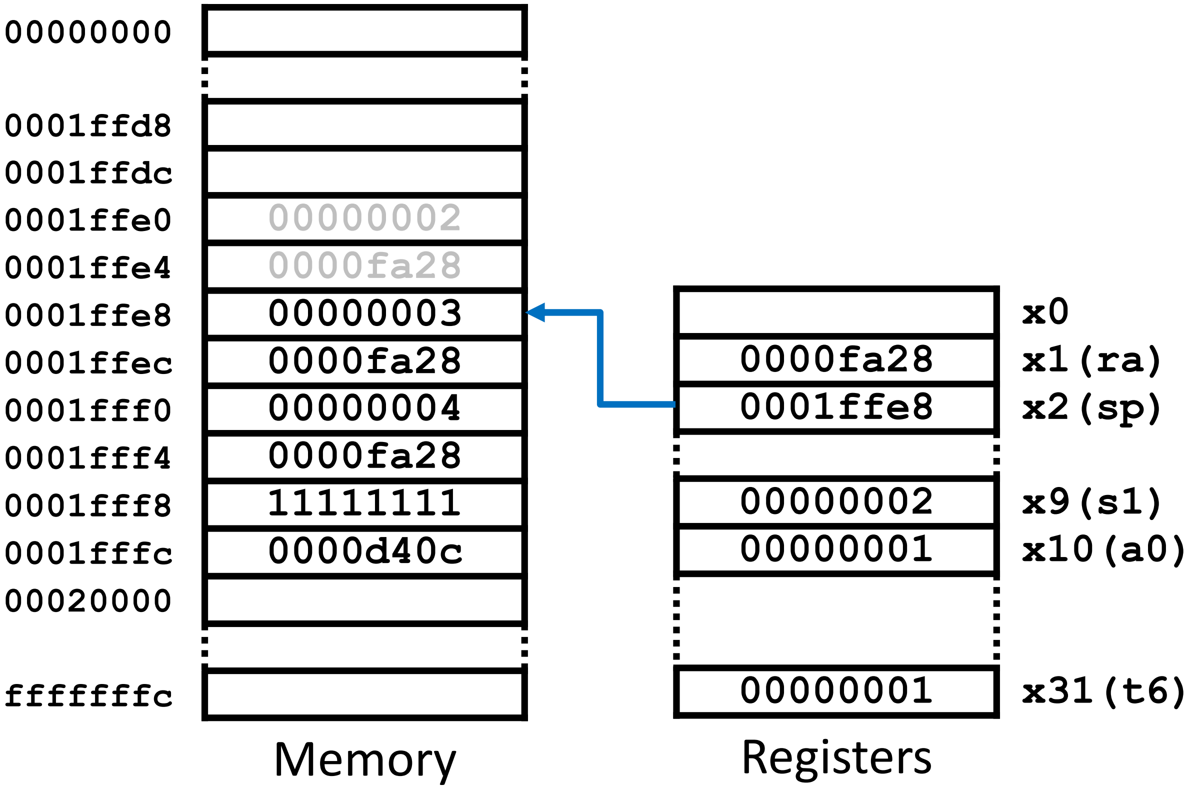
Registers

Functions

Nesting and recursion (iv)

ASM

```
a: .space 4
         ...
         li    sp, 0x20000
         li    a0, 4
         call  fact
0000d40c la    t6, a
         sw    a0, 0(t6)
         ...
fact:
         add   sp, sp, -8
         sw    ra, 4(sp)
         sw    s1, 0(sp)
         mv    s1, a0
         li    t6, 1
         bgt   s1, t6, else
         li    a0, 1
         j     eif
else:
         add   a0, s1, -1
         call  fact
0000fa28 mul   a0, s1, a0
eif:
         lw    ra, 4(sp)
         lw    s1, 0(sp)
         add   sp, sp, 8
         ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe8	x2(sp)
00000002	x9(s1)
00000001	x10(a0)
00000001	x31(t6)

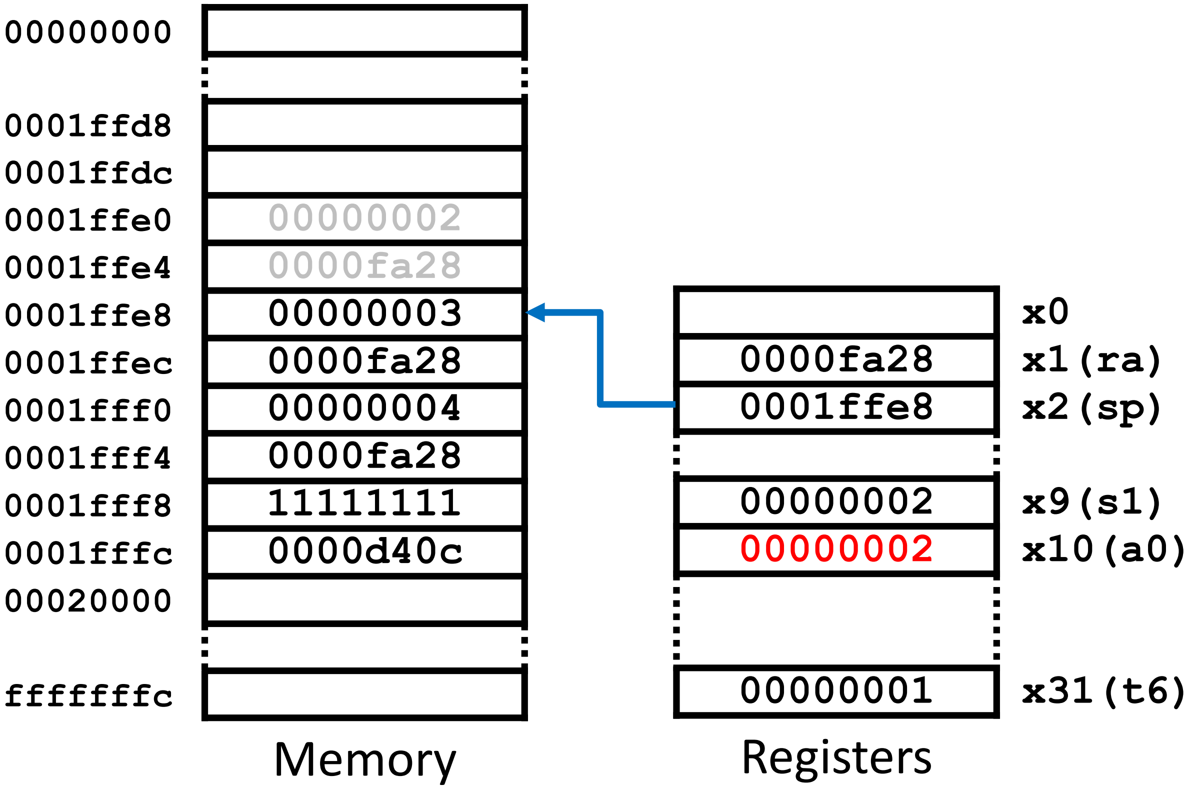
Registers

Functions

Nesting and recursion (iv)

ASM

```
a: .space 4
   ...
   li   sp, 0x20000
   li   a0, 4
   call fact
0000d40c la   t6, a
   sw   a0, 0(t6)
   ...
fact:
   add  sp, sp, -8
   sw   ra, 4(sp)
   sw   s1, 0(sp)
   mv   s1, a0
   li   t6, 1
   bgt  s1, t6, else
   li   a0, 1
   j    eif
else:
   add  a0, s1, -1
   call fact
⇨  mul  a0, s1, a0
eif:
   lw   ra, 4(sp)
   lw   s1, 0(sp)
   add  sp, sp, 8
   ret
```



Memory

00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Registers

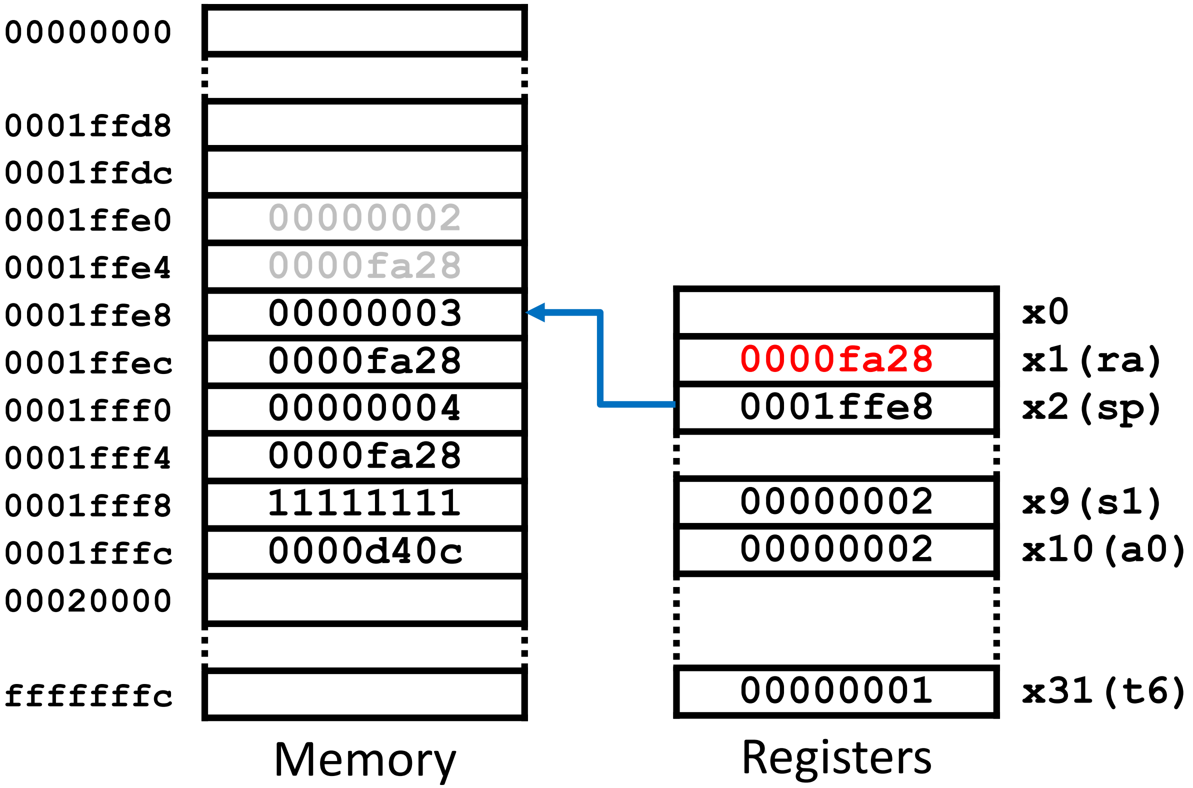
	x0
0000fa28	x1(ra)
0001ffe8	x2(sp)
00000002	x9(s1)
00000002	x10(a0)
00000001	x31(t6)

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
       ...
       li    sp, 0x20000
       li    a0, 4
       call  fact
0000d40c la    t6, a
       sw    a0, 0(t6)
       ...
fact:
       add   sp, sp, -8
       sw    ra, 4(sp)
       sw    s1, 0(sp)
       mv    s1, a0
       li    t6, 1
       bgt   s1, t6, else
       li    a0, 1
       j     eif
else:
       add   a0, s1, -1
       call  fact
0000fa28 mul   a0, s1, a0
eif:
       lw    ra, 4(sp)
       lw    s1, 0(sp)
       add   sp, sp, 8
       ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe8	x2(sp)
00000002	x9(s1)
00000002	x10(a0)
00000001	x31(t6)

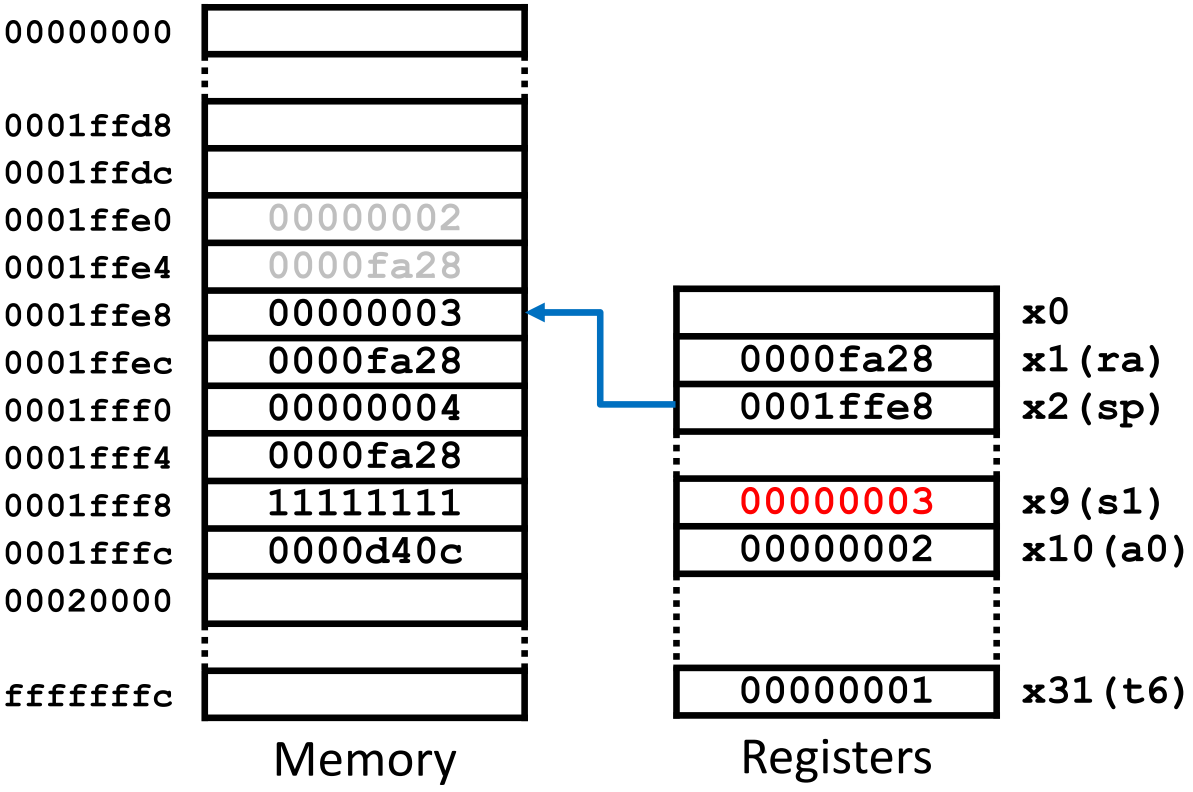
Registers

Functions

Nesting and recursion (iv)

ASM

```
a: .space 4
    ...
    li   sp, 0x20000
    li   a0, 4
    call fact
0000d40c la   t6, a
    sw   a0, 0(t6)
    ...
fact:
    add  sp, sp, -8
    sw   ra, 4(sp)
    sw   s1, 0(sp)
    mv   s1, a0
    li   t6, 1
    bgt  s1, t6, else
    li   a0, 1
    j    eif
else:
    add  a0, s1, -1
    call fact
0000fa28 mul  a0, s1, a0
eif:
    lw   ra, 4(sp)
⇨   lw   s1, 0(sp)
    add  sp, sp, 8
    ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001ffe8	x2(sp)
00000003	x9(s1)
00000002	x10(a0)
00000001	x31(t6)

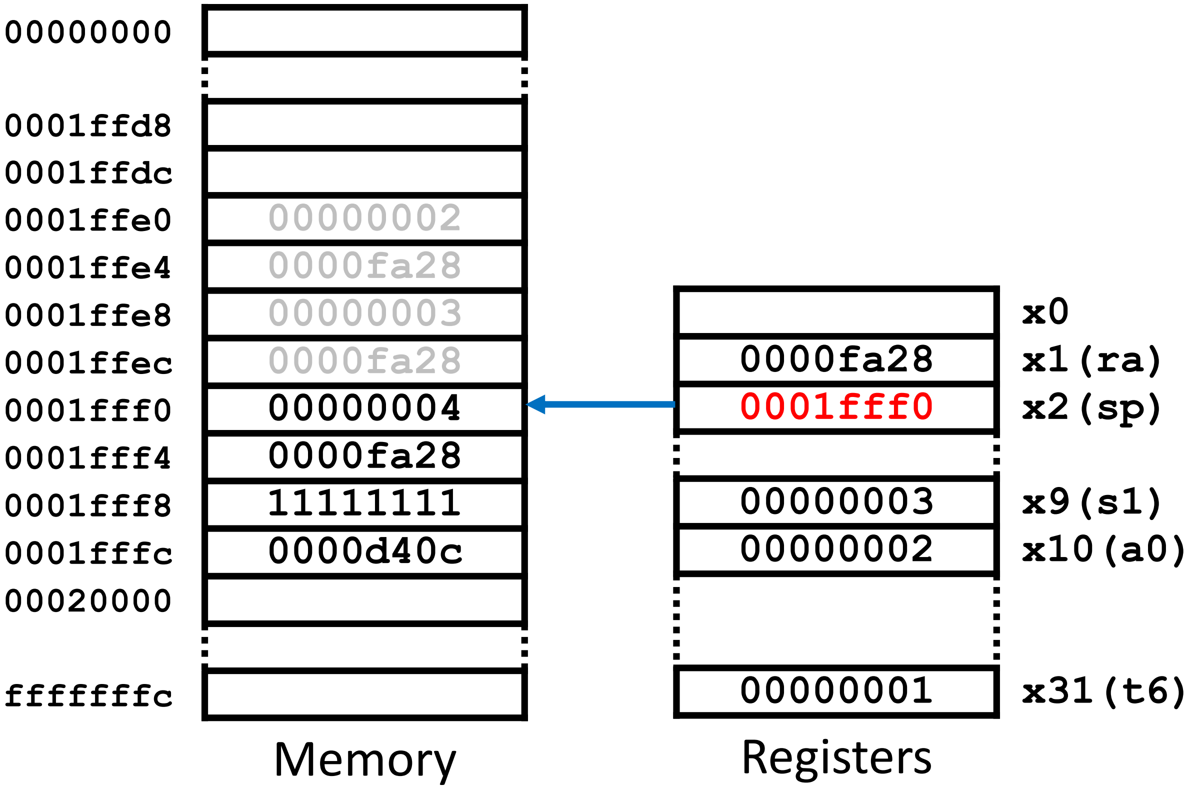
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
     ...
     li    sp, 0x20000
     li    a0, 4
     call  fact
0000d40c la    t6, a
     sw    a0, 0(t6)
     ...
fact:
     add   sp, sp, -8
     sw    ra, 4(sp)
     sw    s1, 0(sp)
     mv    s1, a0
     li    t6, 1
     bgt   s1, t6, else
     li    a0, 1
     j     eif
else:
     add   a0, s1, -1
     call  fact
0000fa28 mul   a0, s1, a0
eif:
     lw    ra, 4(sp)
     lw    s1, 0(sp)
     add   sp, sp, 8
     ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001fff0	x2(sp)
00000003	x9(s1)
00000002	x10(a0)
00000001	x31(t6)

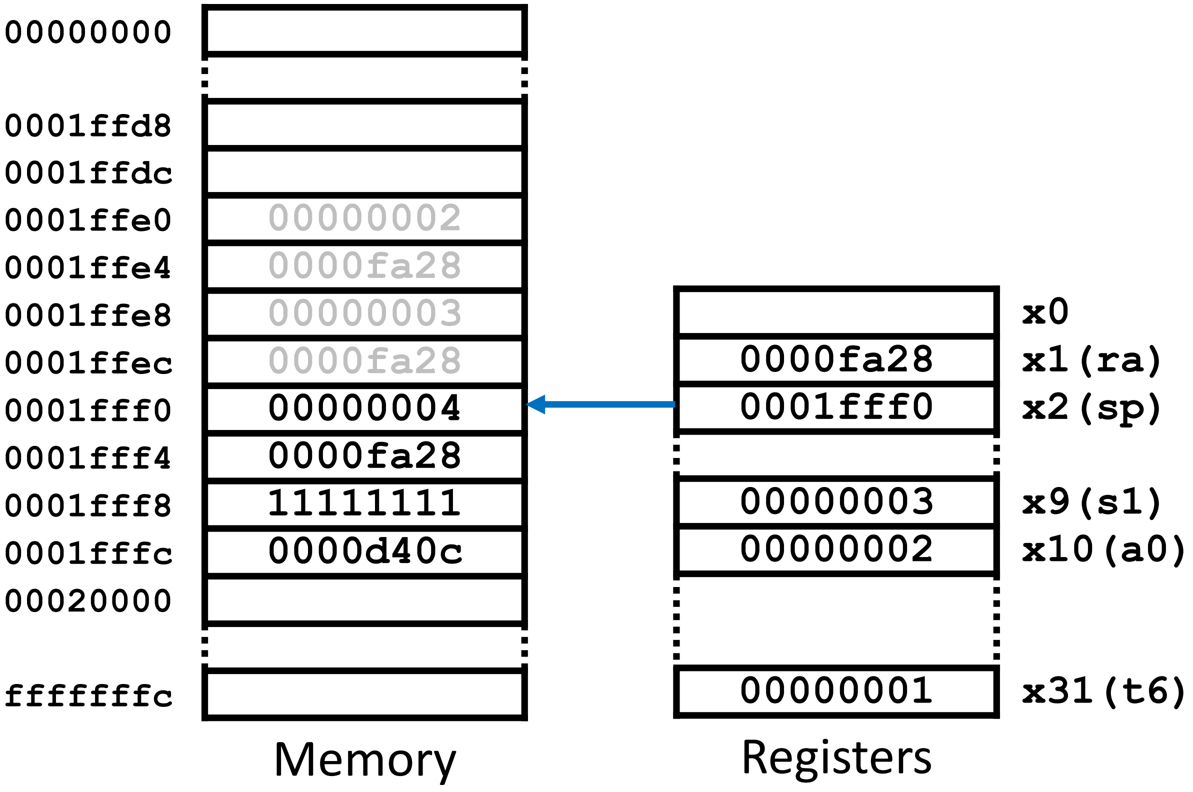
Registers

Functions

Nesting and recursion (iv)

ASM

```
         a:  .space 4
             ...
             li   sp, 0x20000
             li   a0, 4
             call fact
0000d40c     la   t6, a
             sw   a0, 0(t6)
             ...
         fact:
             add  sp, sp, -8
             sw   ra, 4(sp)
             sw   s1, 0(sp)
             mv   s1, a0
             li   t6, 1
             bgt  s1, t6, else
             li   a0, 1
             j    eif
         else:
             add  a0, s1, -1
             call fact
0000fa28     mul  a0, s1, a0
         eif:
             lw   ra, 4(sp)
             lw   s1, 0(sp)
             add  sp, sp, 8
             ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001fff0	x2(sp)
00000003	x9(s1)
00000002	x10(a0)
00000001	x31(t6)

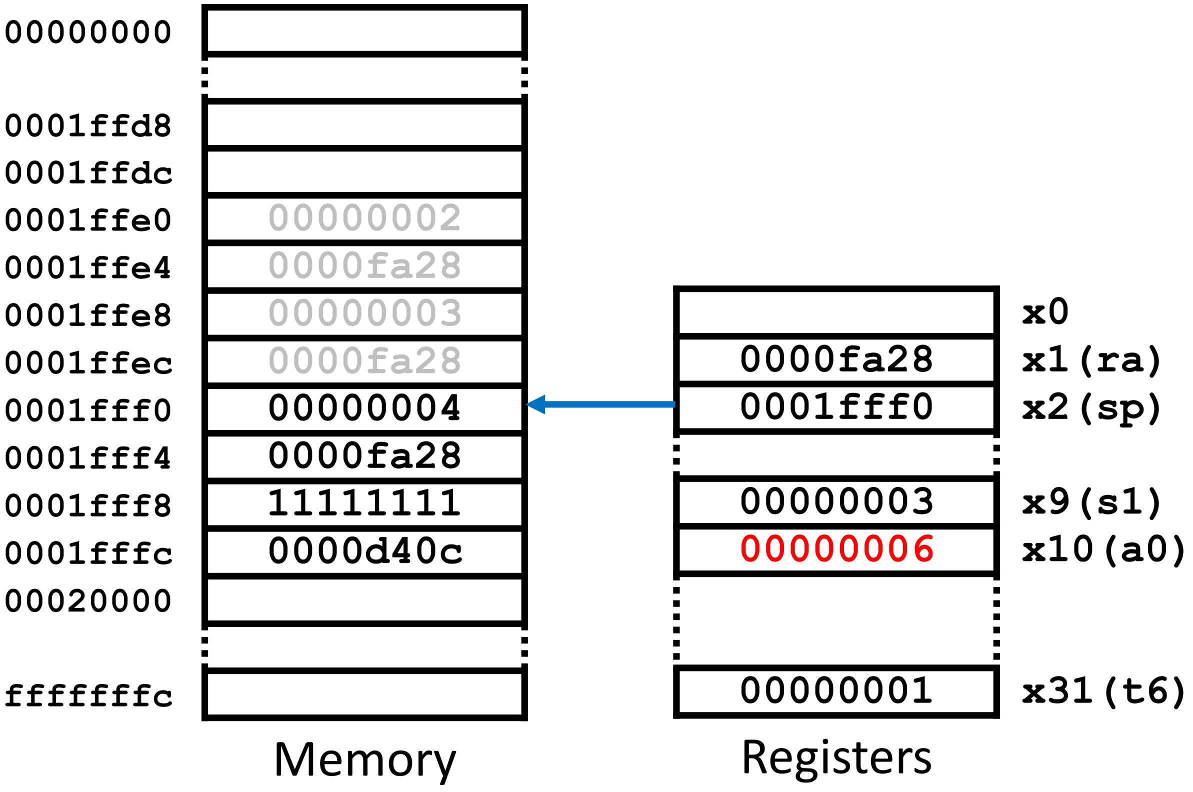
Registers

Functions

Nesting and recursion (iv)

ASM

```
a: .space 4
   ...
   li    sp, 0x20000
   li    a0, 4
   call  fact
0000d40c la    t6, a
   sw    a0, 0(t6)
   ...
fact:
   add   sp, sp, -8
   sw    ra, 4(sp)
   sw    s1, 0(sp)
   mv    s1, a0
   li    t6, 1
   bgt   s1, t6, else
   li    a0, 1
   j     eif
else:
   add   a0, s1, -1
   call  fact
   mul   a0, s1, a0
eif:
   lw    ra, 4(sp)
   lw    s1, 0(sp)
   add   sp, sp, 8
   ret
```



Memory

00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Registers

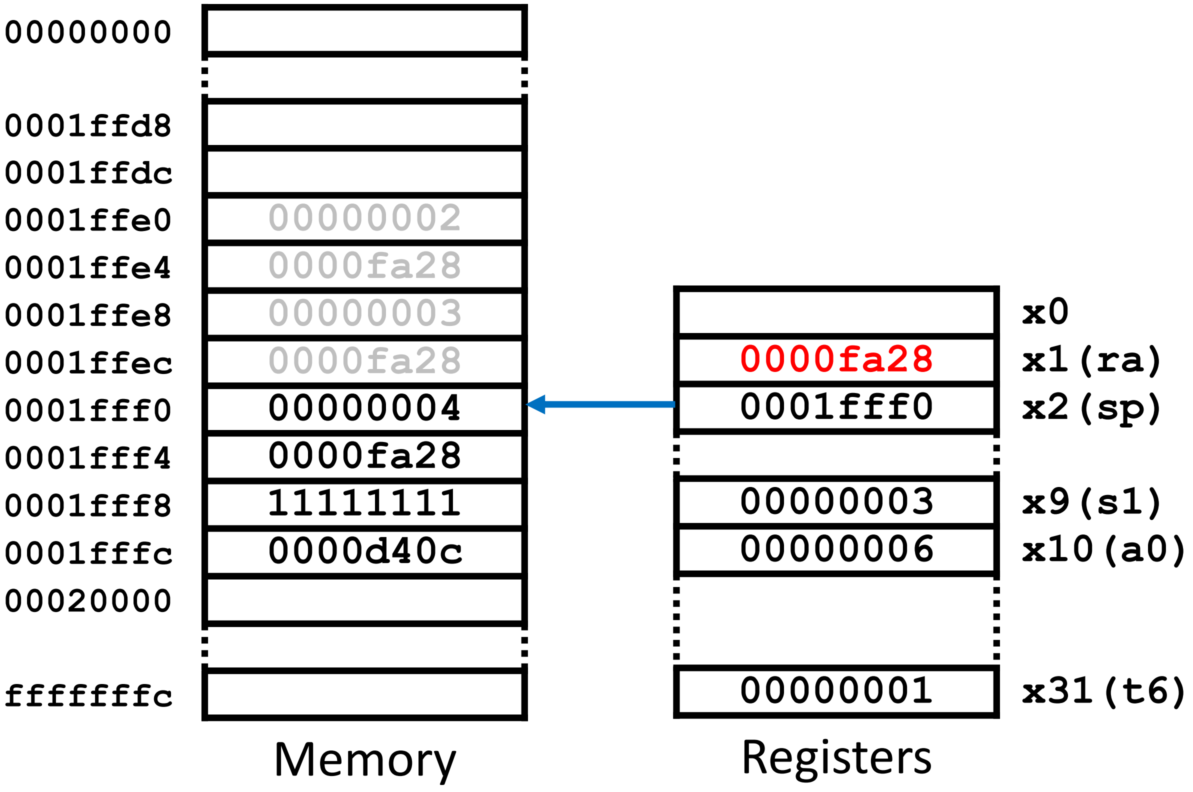
	x0
0000fa28	x1(ra)
0001fff0	x2(sp)
00000003	x9(s1)
00000006	x10(a0)
00000001	x31(t6)

Functions

Nesting and recursion (iv)

ASM

```
         a:  .space 4
               ...
               li    sp, 0x20000
               li    a0, 4
               call  fact
0000d40c       la    t6, a
               sw    a0, 0(t6)
               ...
         fact:
               add   sp, sp, -8
               sw    ra, 4(sp)
               sw    s1, 0(sp)
               mv    s1, a0
               li    t6, 1
               bgt   s1, t6, else
               li    a0, 1
               j     eif
         else:
               add   a0, s1, -1
               call  fact
0000fa28       mul   a0, s1, a0
         eif:
               lw    ra, 4(sp)
               lw    s1, 0(sp)
               add   sp, sp, 8
               ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1 (ra)
0001fff0	x2 (sp)
00000003	x9 (s1)
00000006	x10 (a0)
00000001	x31 (t6)

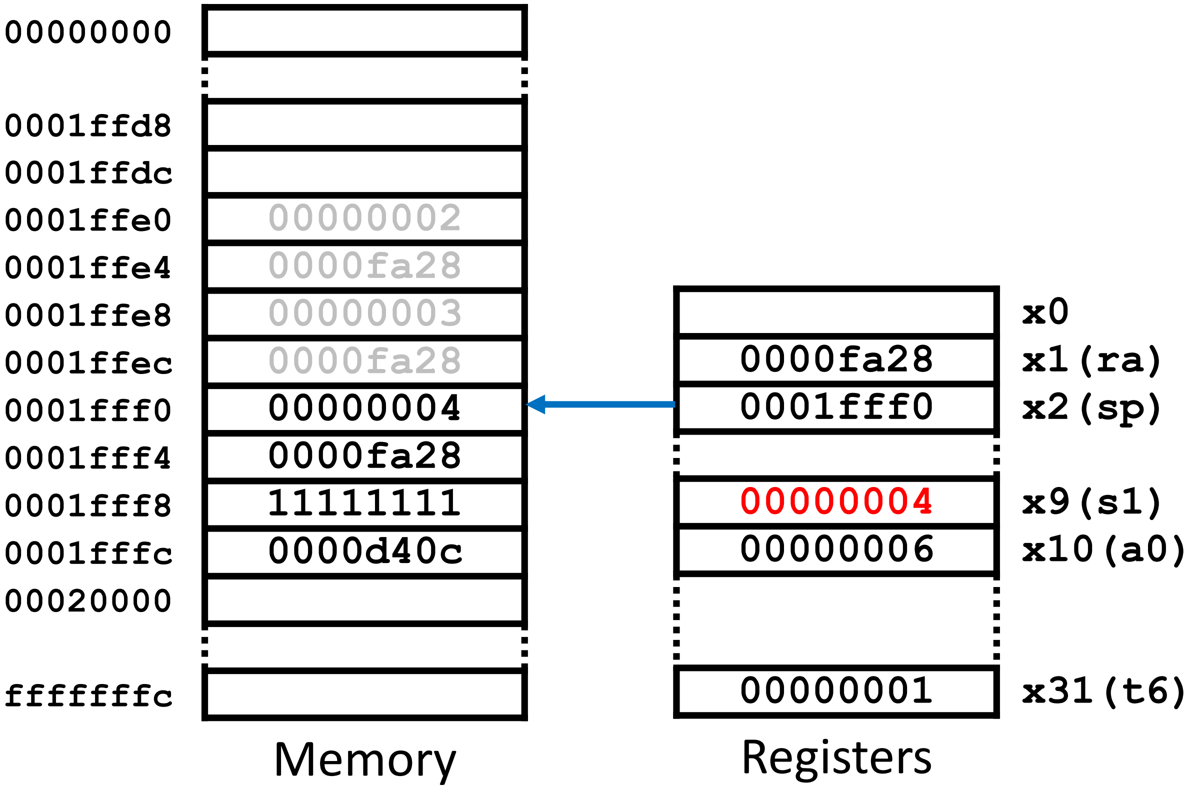
Registers

Functions

Nesting and recursion (iv)

ASM

```
         a:  .space 4
               ...
               li   sp, 0x20000
               li   a0, 4
               call fact
0000d40c       la   t6, a
               sw   a0, 0(t6)
               ...
         fact:
               add  sp, sp, -8
               sw   ra, 4(sp)
               sw   s1, 0(sp)
               mv   s1, a0
               li   t6, 1
               bgt  s1, t6, else
               li   a0, 1
               j    eif
         else:
               add  a0, s1, -1
               call fact
0000fa28       mul  a0, s1, a0
         eif:
               lw   ra, 4(sp)
    ⇨          lw   s1, 0(sp)
               add  sp, sp, 8
               ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001fff0	x2(sp)
00000004	x9(s1)
00000006	x10(a0)
00000001	x31(t6)

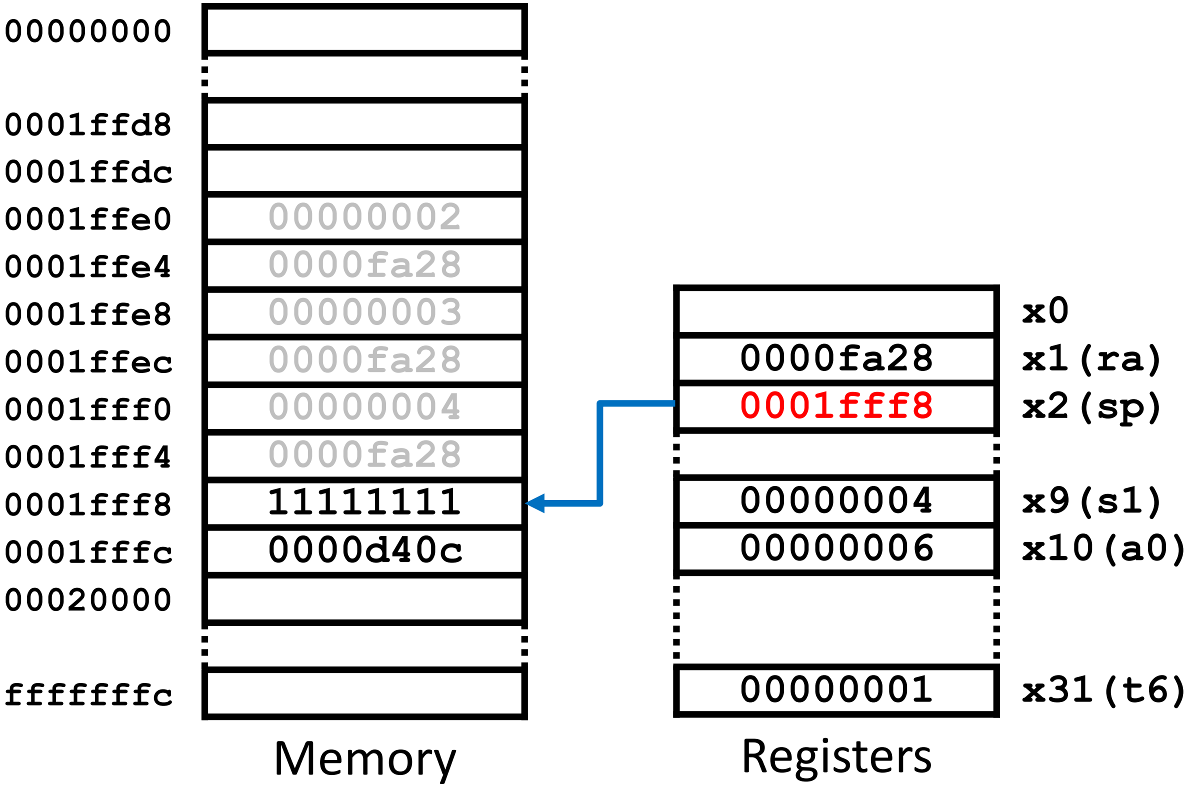
Registers

Functions

Nesting and recursion (iv)

ASM

```
         a:  .space 4
             ...
             li   sp, 0x20000
             li   a0, 4
             call fact
0000d40c     la   t6, a
             sw   a0, 0(t6)
             ...
         fact:
             add  sp, sp, -8
             sw   ra, 4(sp)
             sw   s1, 0(sp)
             mv   s1, a0
             li   t6, 1
             bgt  s1, t6, else
             li   a0, 1
             j    eif
         else:
             add  a0, s1, -1
             call fact
0000fa28     mul  a0, s1, a0
         eif:
             lw   ra, 4(sp)
             lw   s1, 0(sp)
             add  sp, sp, 8
             ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001fff8	x2(sp)
00000004	x9(s1)
00000006	x10(a0)
00000001	x31(t6)

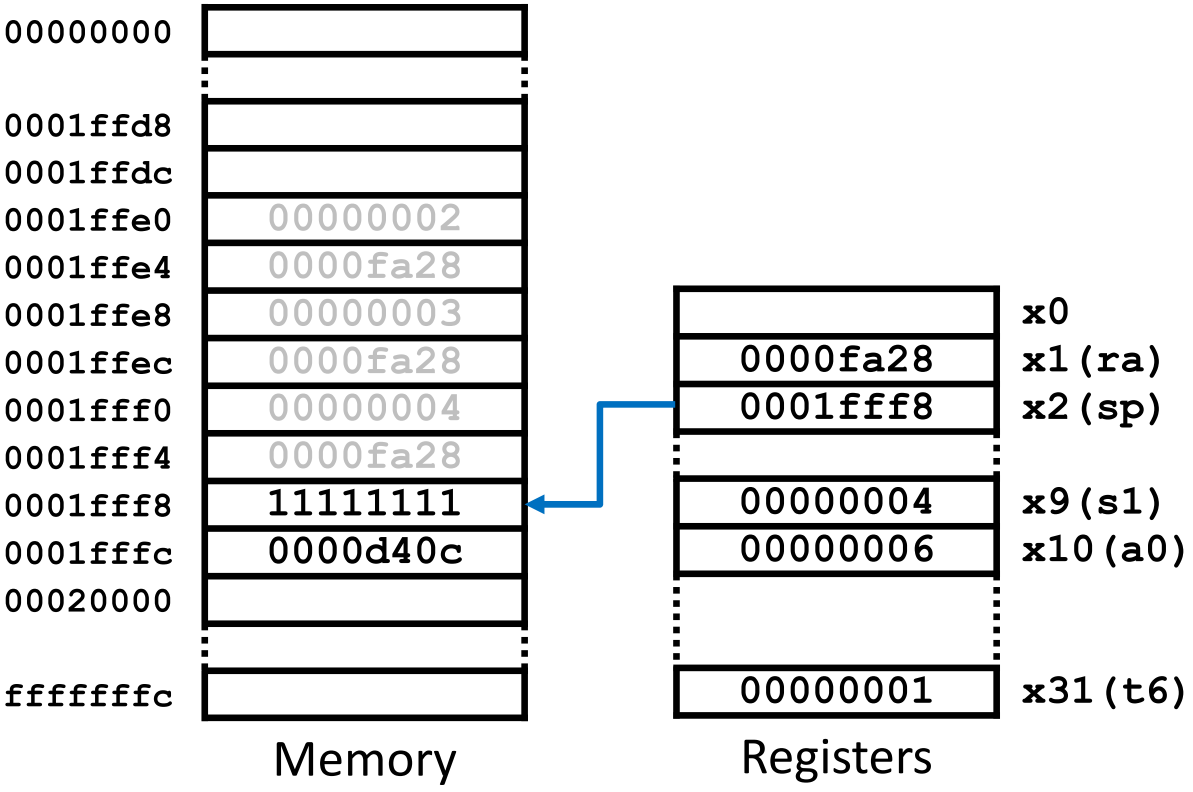
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
    ...
    li   sp, 0x20000
    li   a0, 4
    call fact
0000d40c la   t6, a
    sw   a0, 0(t6)
    ...
fact:
    add  sp, sp, -8
    sw   ra, 4(sp)
    sw   s1, 0(sp)
    mv   s1, a0
    li   t6, 1
    bgt  s1, t6, else
    li   a0, 1
    j    eif
else:
    add  a0, s1, -1
    call fact
0000fa28 mul  a0, s1, a0
eif:
    lw   ra, 4(sp)
    lw   s1, 0(sp)
    add  sp, sp, 8
    ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000fa28	x1(ra)
0001fff8	x2(sp)
00000004	x9(s1)
00000006	x10(a0)
00000001	x31(t6)

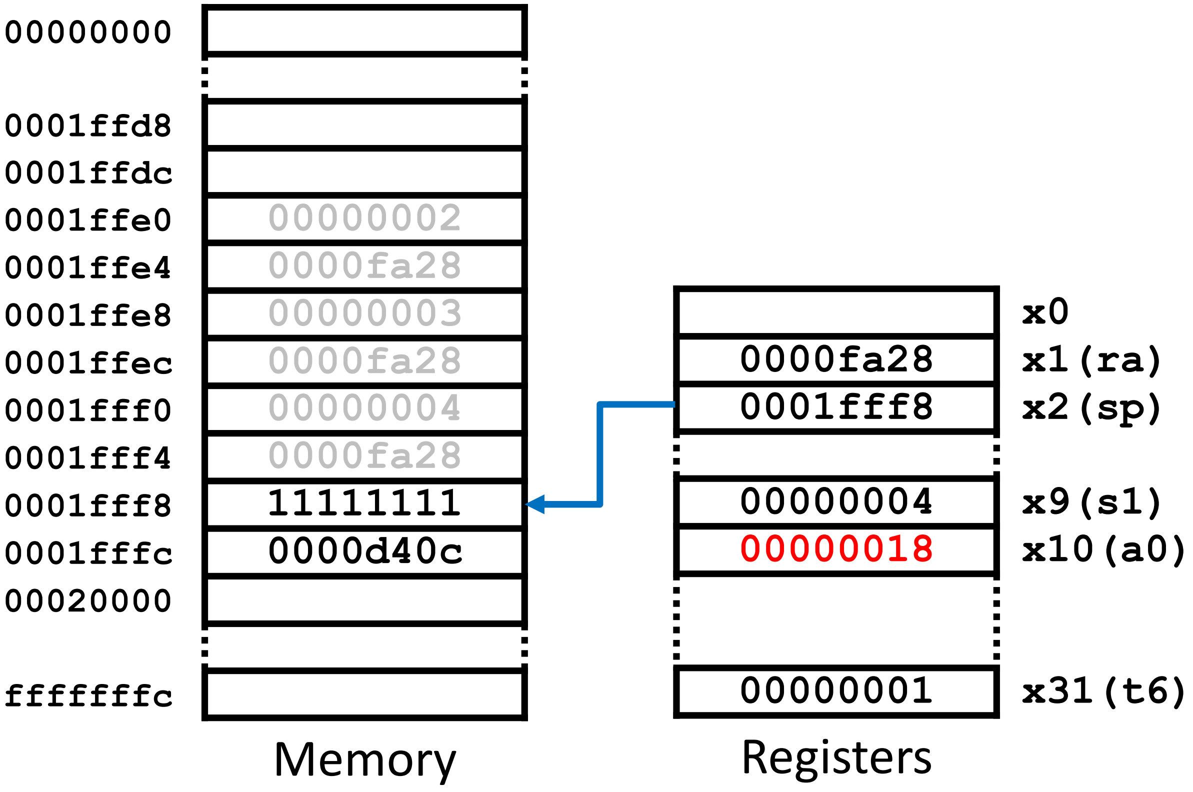
Registers

Functions

Nesting and recursion (iv)

ASM

```
          a:  .space 4
                ...
                li    sp, 0x20000
                li    a0, 4
                call  fact
0000d40c        la    t6, a
                sw    a0, 0(t6)
                ...
          fact:
                add   sp, sp, -8
                sw    ra, 4(sp)
                sw    s1, 0(sp)
                mv    s1, a0
                li    t6, 1
                bgt   s1, t6, else
                li    a0, 1
                j     eif
          else:
                add   a0, s1, -1
                call  fact
        ⇨       mul   a0, s1, a0
          eif:
                lw    ra, 4(sp)
                lw    s1, 0(sp)
                add   sp, sp, 8
                ret
```



Memory

00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Registers

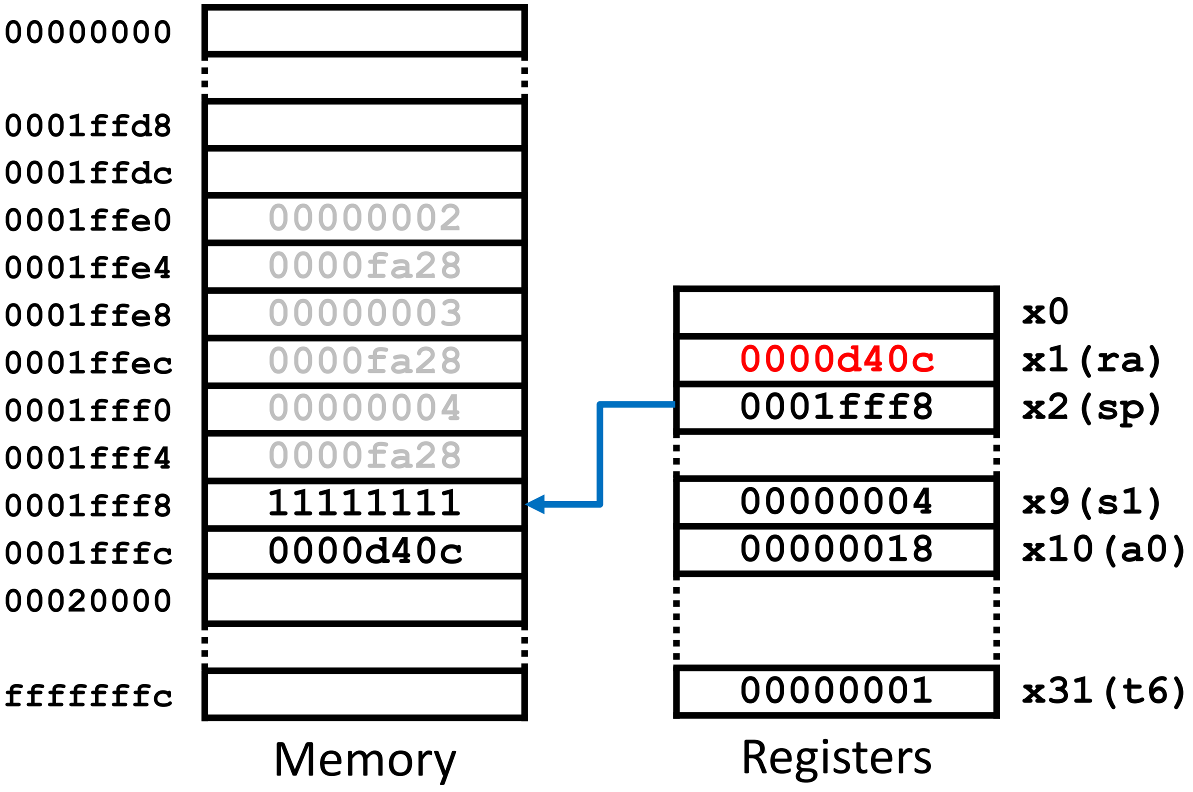
	x0
0000fa28	x1(ra)
0001fff8	x2(sp)
00000004	x9(s1)
00000018	x10(a0)
00000001	x31(t6)

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
    ...
    li    sp, 0x20000
    li    a0, 4
    call  fact
0000d40c la    t6, a
    sw    a0, 0(t6)
    ...
fact:
    add   sp, sp, -8
    sw    ra, 4(sp)
    sw    s1, 0(sp)
    mv    s1, a0
    li    t6, 1
    bgt   s1, t6, else
    li    a0, 1
    j     eif
else:
    add   a0, s1, -1
    call  fact
0000fa28 mul   a0, s1, a0
eif:
    lw    ra, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 8
    ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001fff8	x2(sp)
00000004	x9(s1)
00000018	x10(a0)
00000001	x31(t6)

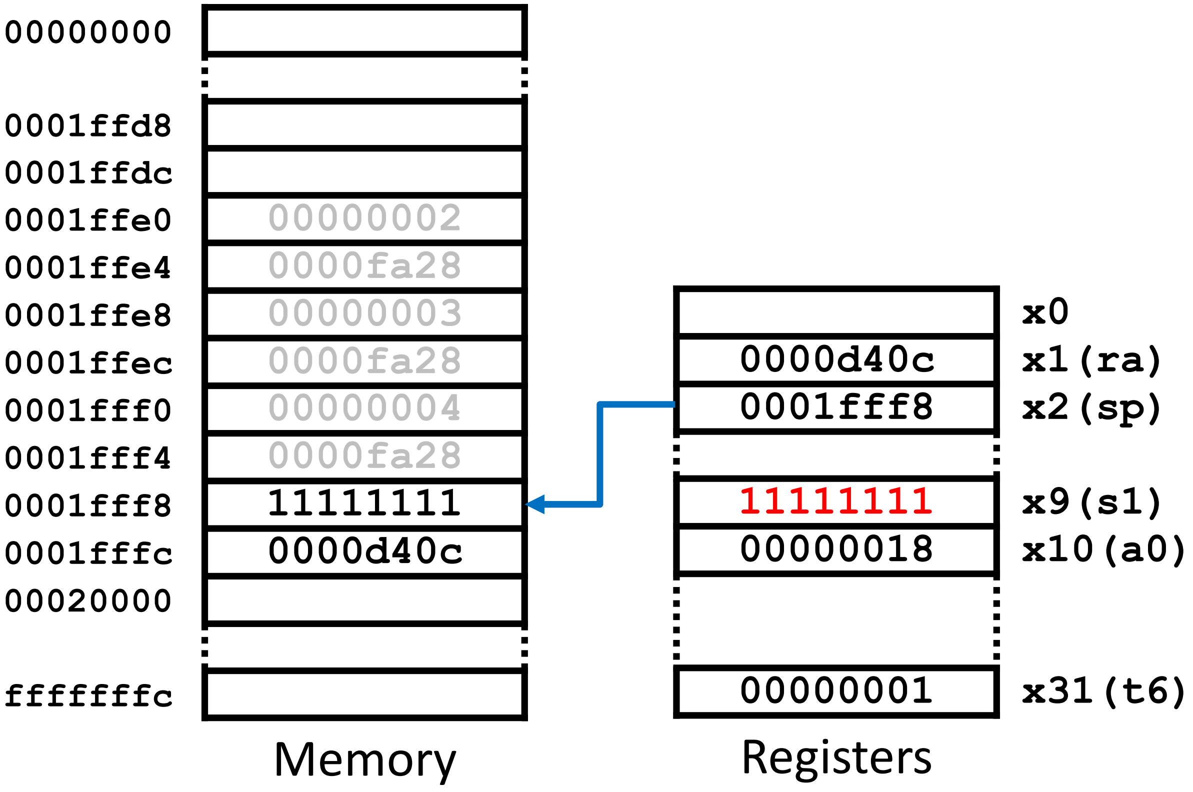
Registers

Functions

Nesting and recursion (iv)

ASM

```
         a:  .space 4
             ...
             li   sp, 0x20000
             li   a0, 4
             call fact
0000d40c     la   t6, a
             sw   a0, 0(t6)
             ...
         fact:
             add  sp, sp, -8
             sw   ra, 4(sp)
             sw   s1, 0(sp)
             mv   s1, a0
             li   t6, 1
             bgt  s1, t6, else
             li   a0, 1
             j    eif
         else:
             add  a0, s1, -1
             call fact
0000fa28     mul  a0, s1, a0
         eif:
             lw   ra, 4(sp)
        ⇨    lw   s1, 0(sp)
             add  sp, sp, 8
             ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

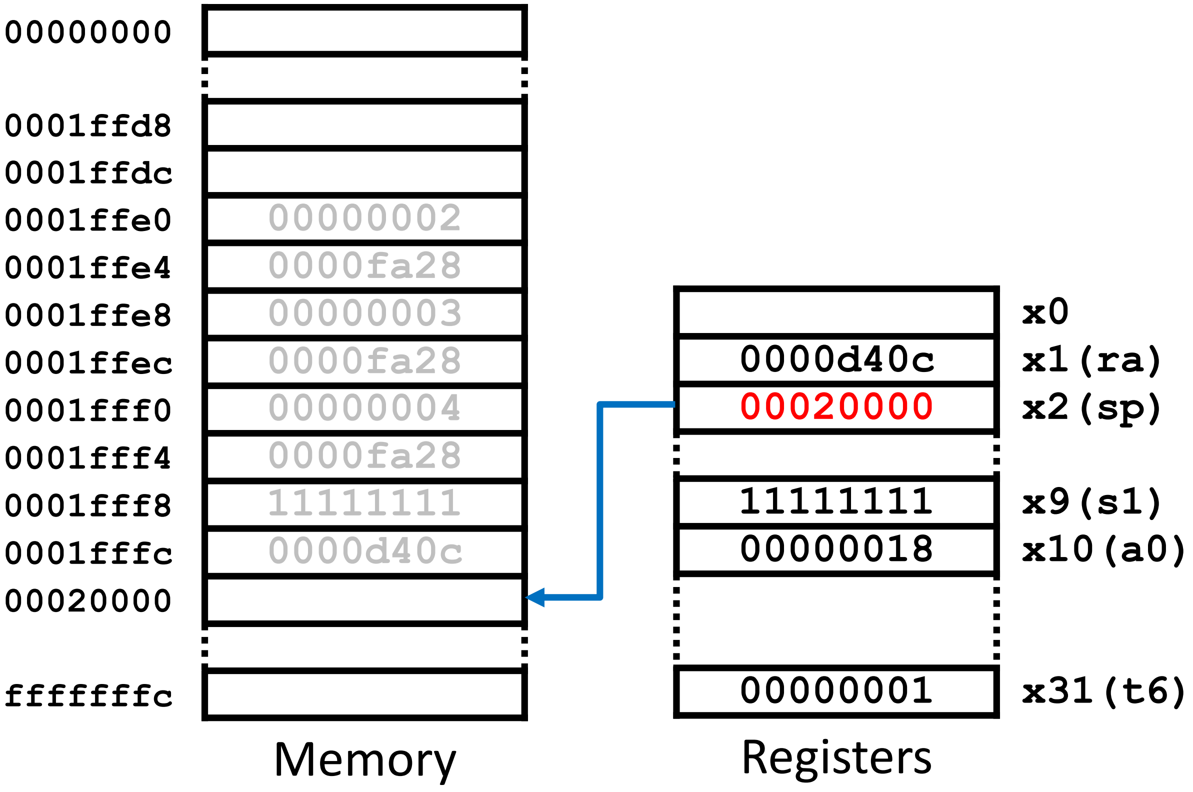
	x0
0000d40c	x1 (ra)
0001fff8	x2 (sp)
11111111	x9 (s1)
00000018	x10 (a0)
00000001	x31 (t6)

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
    ...
    li   sp, 0x20000
    li   a0, 4
    call fact
0000d40c la   t6, a
    sw   a0, 0(t6)
    ...
fact:
    add  sp, sp, -8
    sw   ra, 4(sp)
    sw   s1, 0(sp)
    mv   s1, a0
    li   t6, 1
    bgt  s1, t6, else
    li   a0, 1
    j    eif
else:
    add  a0, s1, -1
    call fact
0000fa28 mul  a0, s1, a0
eif:
    lw   ra, 4(sp)
    lw   s1, 0(sp)
⇨   add  sp, sp, 8
    ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
00020000	x2(sp)
11111111	x9(s1)
00000018	x10(a0)
00000001	x31(t6)

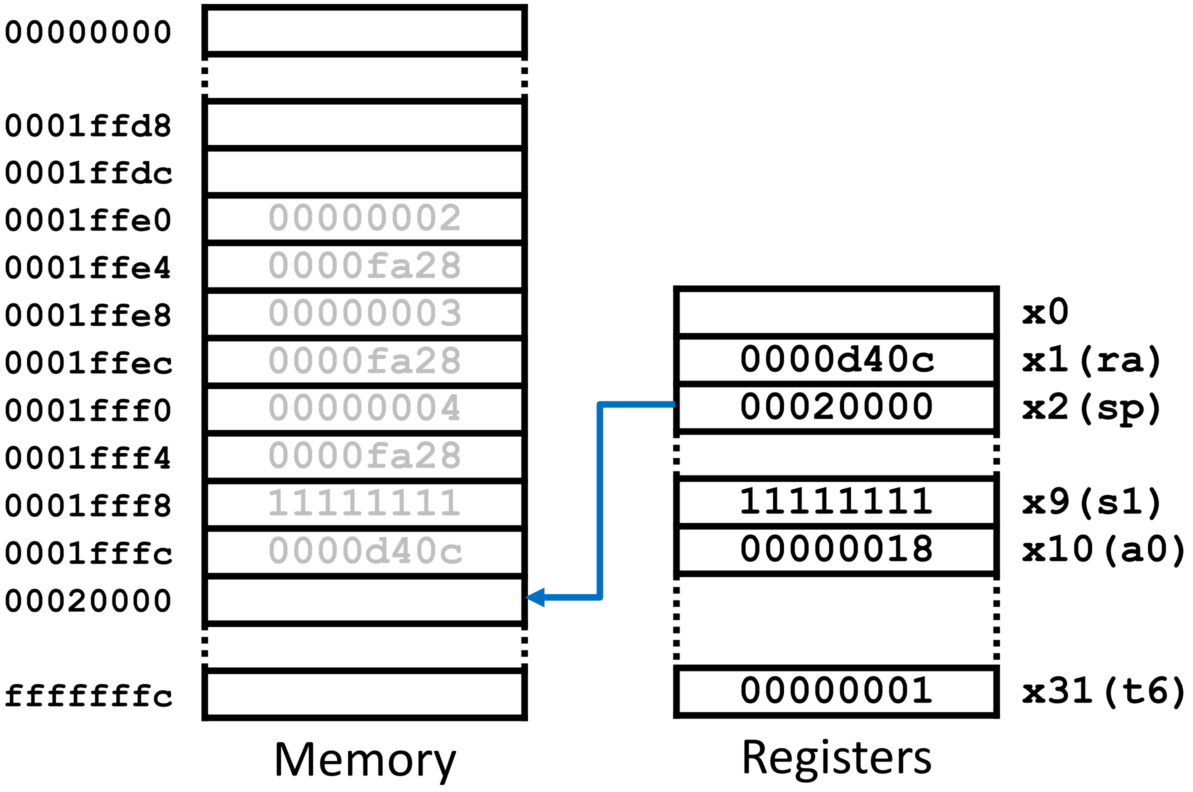
Registers

Functions

Nesting and recursion (iv)

ASM

```
          a:  .space 4
                ...
                li    sp, 0x20000
                li    a0, 4
                call  fact
0000d40c        la    t6, a
                sw    a0, 0(t6)
                ...
          fact:
                add   sp, sp, -8
                sw    ra, 4(sp)
                sw    s1, 0(sp)
                mv    s1, a0
                li    t6, 1
                bgt   s1, t6, else
                li    a0, 1
                j     eif
          else:
                add   a0, s1, -1
                call  fact
0000fa28        mul   a0, s1, a0
          eif:
                lw    ra, 4(sp)
                lw    s1, 0(sp)
                add   sp, sp, 8
                ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
00020000	x2(sp)
11111111	x9(s1)
00000018	x10(a0)
00000001	x31(t6)

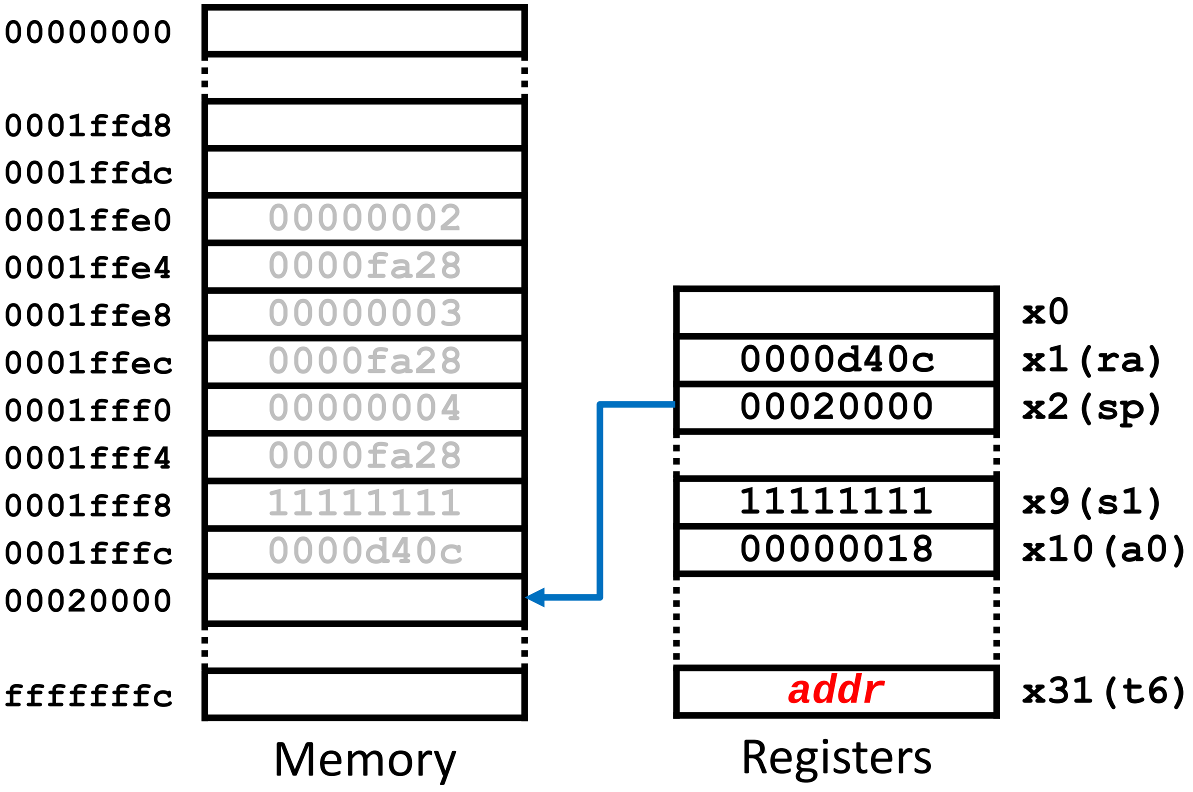
Registers

Functions

Nesting and recursion (iv)

ASM

```
a:  .space 4
    ...
    li   sp, 0x20000
    li   a0, 4
    call fact
    la   t6, a
    sw   a0, 0(t6)
    ...
fact:
    add  sp, sp, -8
    sw   ra, 4(sp)
    sw   s1, 0(sp)
    mv   s1, a0
    li   t6, 1
    bgt  s1, t6, else
    li   a0, 1
    j    eif
else:
    add  a0, s1, -1
    call fact
0000fa28 mul  a0, s1, a0
eif:
    lw   ra, 4(sp)
    lw   s1, 0(sp)
    add  sp, sp, 8
    ret
```



00000000	
0001ffd8	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

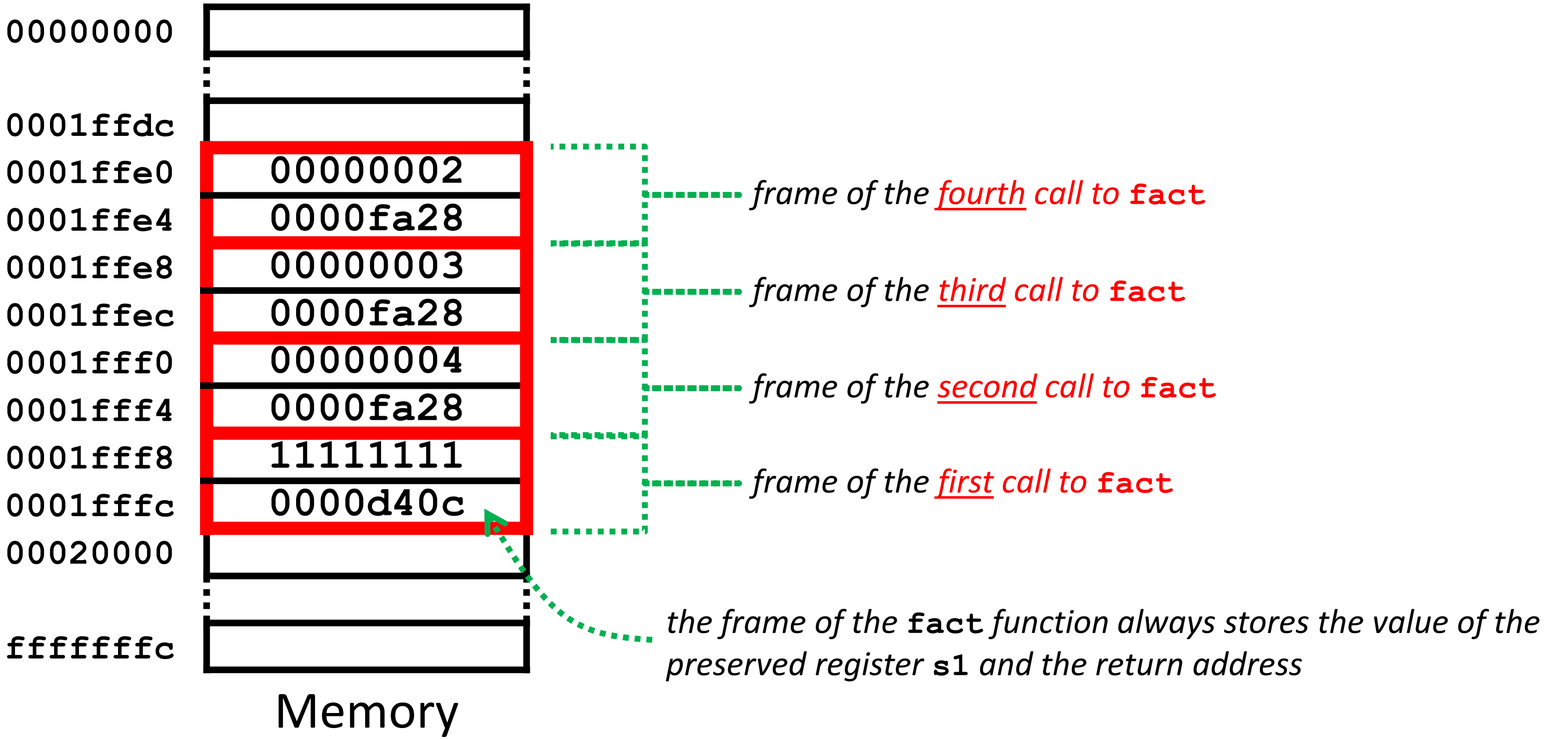
	x0
0000d40c	x1 (ra)
00020000	x2 (sp)
11111111	x9 (s1)
00000018	x10 (a0)
addr	x31 (t6)

Registers

Functions

Frames

- A frame is a stack region where all data related to a function activation are placed.
  - All frames have a fixed structure
    - They contain the same type of information and in the same relative position.
  - But in each activation of the function, its frame may be placed in different effective memory positions.



00000000	
0001ffdc	
0001ffe0	00000002
0001ffe4	0000fa28
0001ffe8	00000003
0001ffec	0000fa28
0001fff0	00000004
0001fff4	0000fa28
0001fff8	11111111
0001fffc	0000d40c
00020000	
fffffffc	

Memory

frame of the fourth call to `fact`

frame of the third call to `fact`

frame of the second call to `fact`

frame of the first call to `fact`

the frame of the `fact` *function always stores the value of the preserved register* `s1` *and the return address*

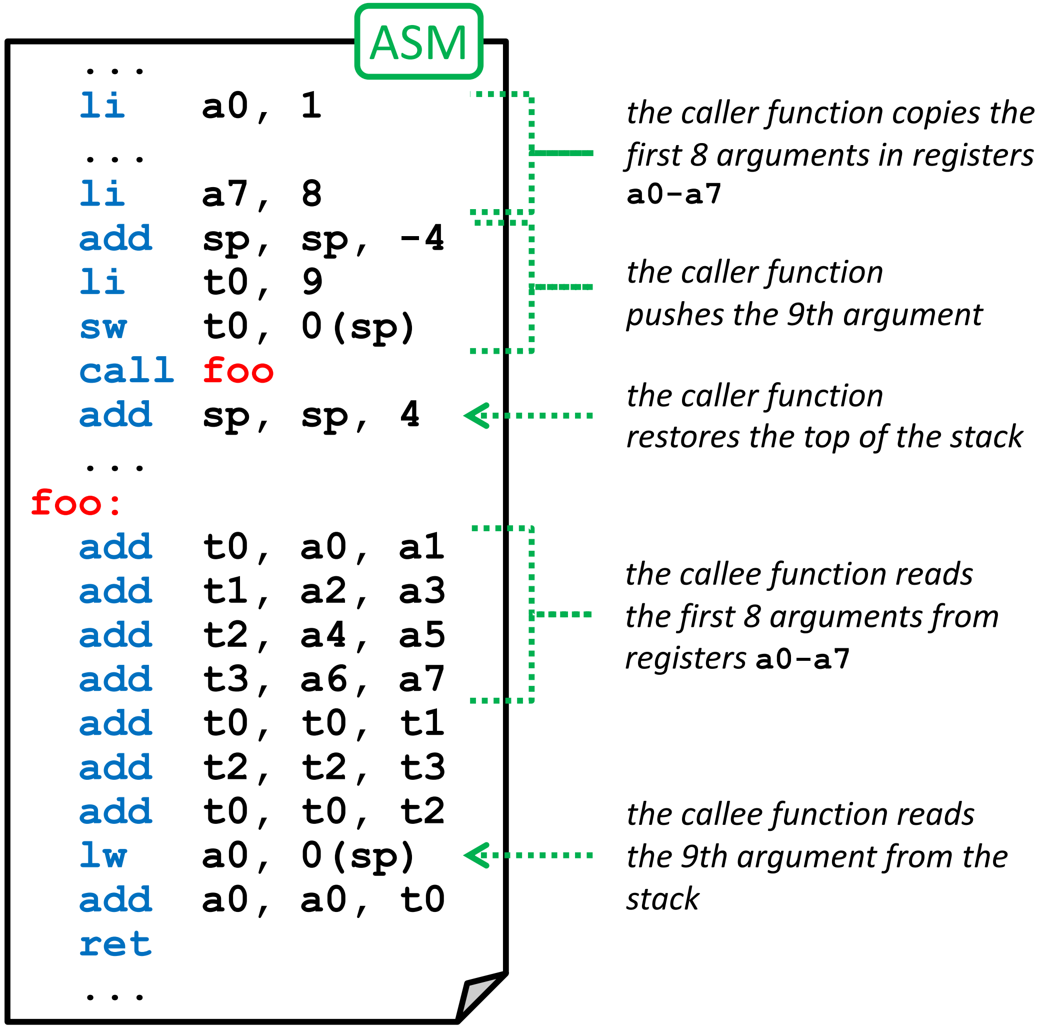
Functions

Passing a large number of arguments

- If the number of parameters of a function is larger than 8, the caller function must push the 9th argument (and successive) before branching.

C/C++

```
...
y = foo( 1, 2, 3,
         4, 5, 6,
         7, 8, 9 );
...
int foo( int a, int b, int c,
         int d, int e, int f,
         int g, int h, int i )
{
  return a + b + c + d +
         e + f + g + h + i;
}
...
```



ASM

```
    ...
    li    a0, 1
    ...
    li    a7, 8
    add   sp, sp, -4
    li    t0, 9
    sw    t0, 0(sp)
    call  foo
    add   sp, sp, 4
    ...
foo:
    add   t0, a0, a1
    add   t1, a2, a3
    add   t2, a4, a5
    add   t3, a6, a7
    add   t0, t0, t1
    add   t2, t2, t3
    add   t0, t0, t2
    lw    a0, 0(sp)
    add   a0, a0, t0
    ret
    ...
```

the caller function copies the first 8 arguments in registers `a0-a7`

the caller function pushes the 9th argument

the caller function restores the top of the stack

the callee function reads the first 8 arguments from registers `a0-a7`

the callee function reads the 9th argument from the stack

Functions

Local variables (i)

- When there are not enough available registers, the local variables of a function are placed in the stack (in the function frame):
  - They are only alive during the execution of the function.
  - Since they do not have fixed effective addresses, labels cannot be used to address them, and therefore immediate offsets relative to a register are used.

C/C++

```
int baz( int a, int b,
         int c, int d )
{
  int sum1, sum2;

  sum1 = (a + b);
  sum2 = (c + d);
  ...
  return sum1 - sum2;
}
```

sum1 → 4(sp)   sum2 → 0(sp)

ASM

```
baz:
  add  sp, sp, -8
  add  t6, a0, a1
  sw   t6, 4(sp)
  add  t6, a2, a3
  sw   t6, 0(sp)
  ...
  lw   t5, 4(sp)
  lw   t6, 0(sp)
  sub  a0, t5, t6
  add  sp, sp, 8
  ret
```

← *reserves stack space for 2 local variables*

← *stores* **a+b** *in* **sum1**

← *stores* **c+d** *in* **sum2**

← *loads* **sum1**

← *loads* **sum2**

← *calculates the return value*

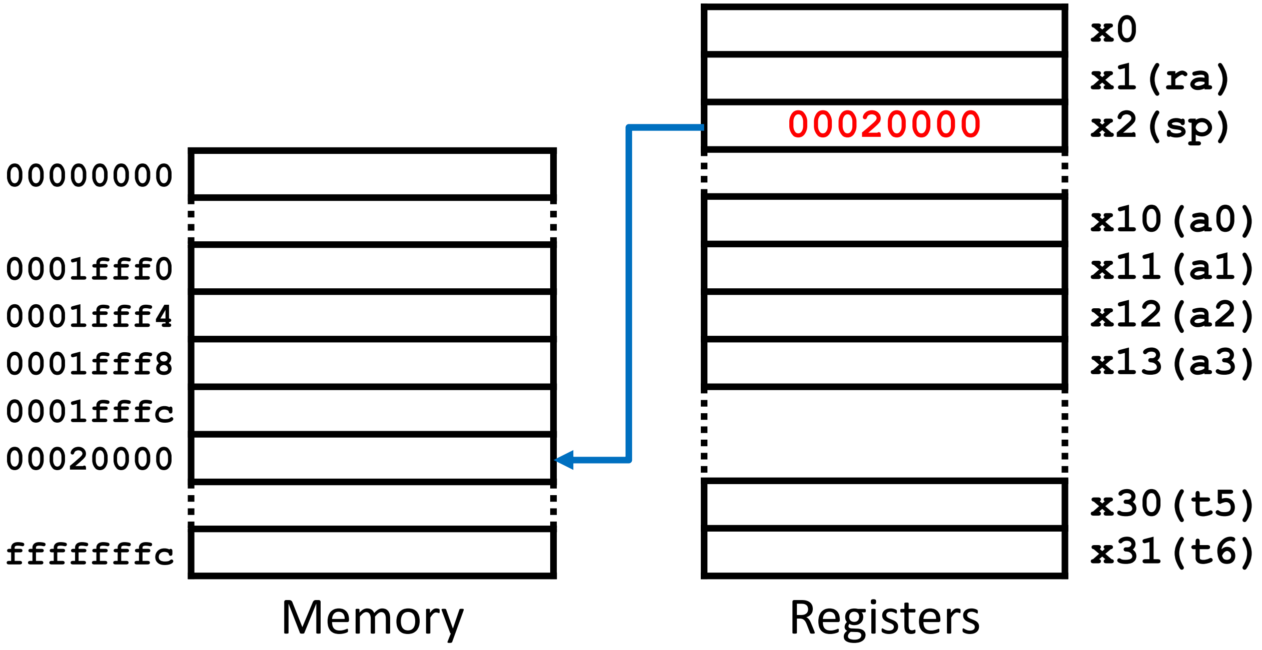
← *frees up stack space*

Functions

Local variables (ii)

ASM

```
           ...
           li    sp, 0x20000
           li    a0, 10
           li    a1, 20
           li    a2, 30
           li    a3, 40
           call  baz
0000d40c   ...
        baz:
           add   sp, sp, -8
           add   t6, a0, a1
           sw    t6, 4(sp)
           add   t6, a2, a3
           sw    t6, 0(sp)
           ...
           lw    t5, 4(sp)
           lw    t6, 0(sp)
           sub   a0, t5, t6
           add   sp, sp, 8
           ret
           ...
```

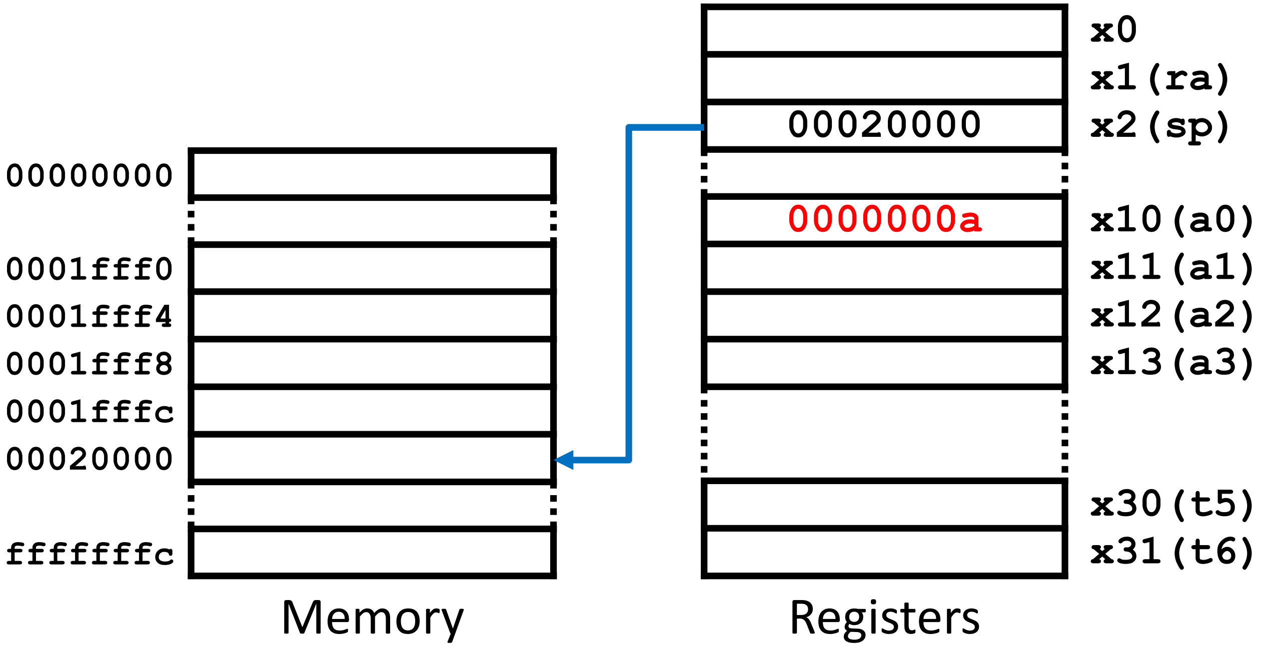


Functions

Local variables (ii)

ASM

```
         ...
         li   sp, 0x20000
         li   a0, 10
         li   a1, 20
         li   a2, 30
         li   a3, 40
         call baz
0000d40c ...
       baz:
         add  sp, sp, -8
         add  t6, a0, a1
         sw   t6, 4(sp)
         add  t6, a2, a3
         sw   t6, 0(sp)
         ...
         lw   t5, 4(sp)
         lw   t6, 0(sp)
         sub  a0, t5, t6
         add  sp, sp, 8
         ret
         ...
```

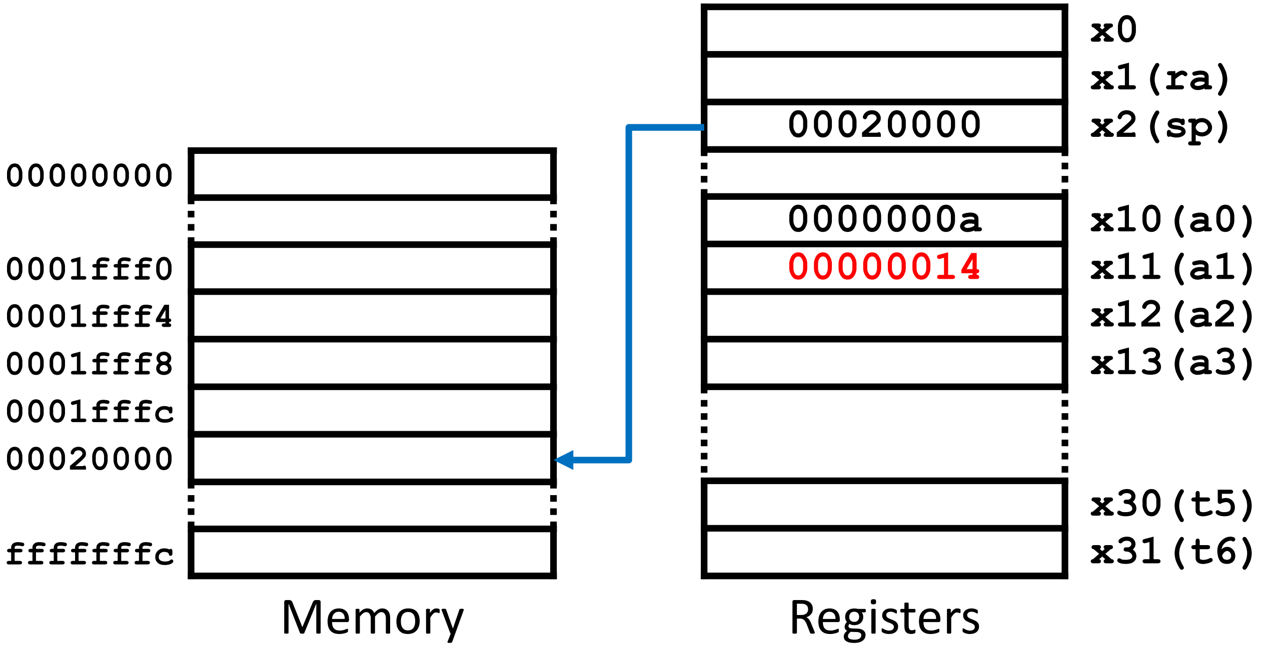


Functions

Local variables (ii)

ASM

```
         ...
         li    sp, 0x20000
         li    a0, 10
         li    a1, 20
         li    a2, 30
         li    a3, 40
         call  baz
0000d40c ...
      baz:
         add   sp, sp, -8
         add   t6, a0, a1
         sw    t6, 4(sp)
         add   t6, a2, a3
         sw    t6, 0(sp)
         ...
         lw    t5, 4(sp)
         lw    t6, 0(sp)
         sub   a0, t5, t6
         add   sp, sp, 8
         ret
         ...
```

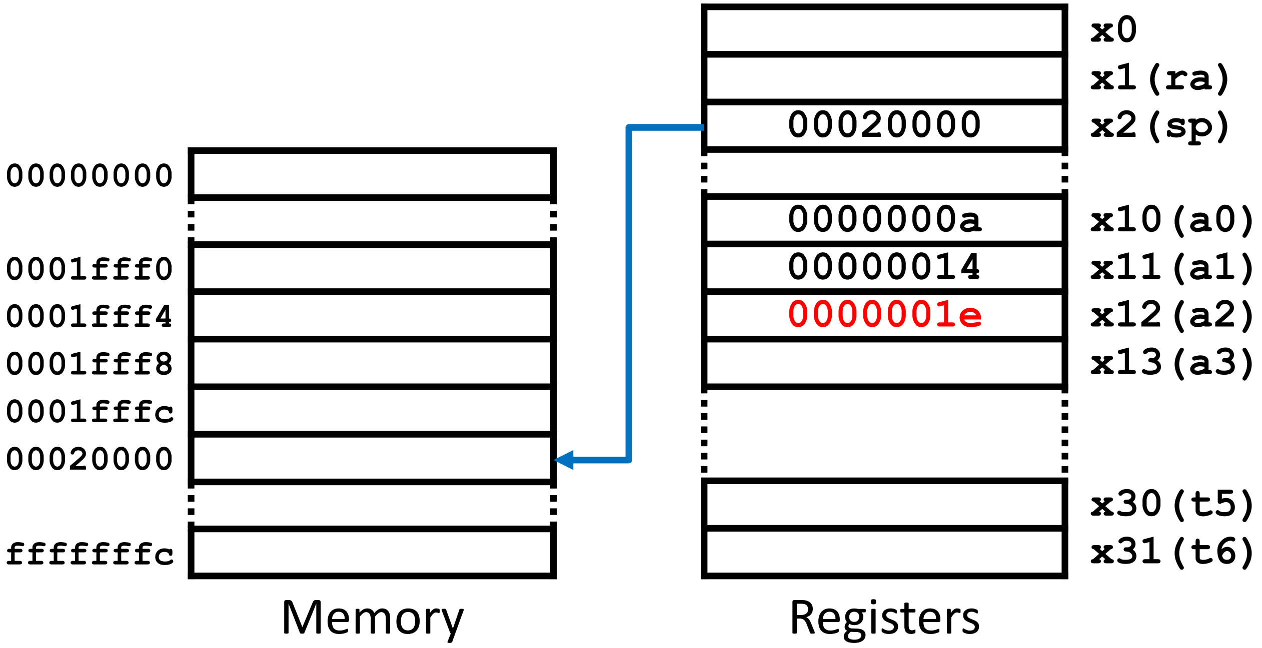


Functions

Local variables (ii)

ASM

```
          ...
          li   sp, 0x20000
          li   a0, 10
          li   a1, 20
          li   a2, 30
          li   a3, 40
          call baz
0000d40c  ...
        baz:
          add  sp, sp, -8
          add  t6, a0, a1
          sw   t6, 4(sp)
          add  t6, a2, a3
          sw   t6, 0(sp)
          ...
          lw   t5, 4(sp)
          lw   t6, 0(sp)
          sub  a0, t5, t6
          add  sp, sp, 8
          ret
          ...
```



00000000	
0001fff0	
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
	x1(ra)
00020000	x2(sp)
0000000a	x10(a0)
00000014	x11(a1)
0000001e	x12(a2)
	x13(a3)
	x30(t5)
	x31(t6)

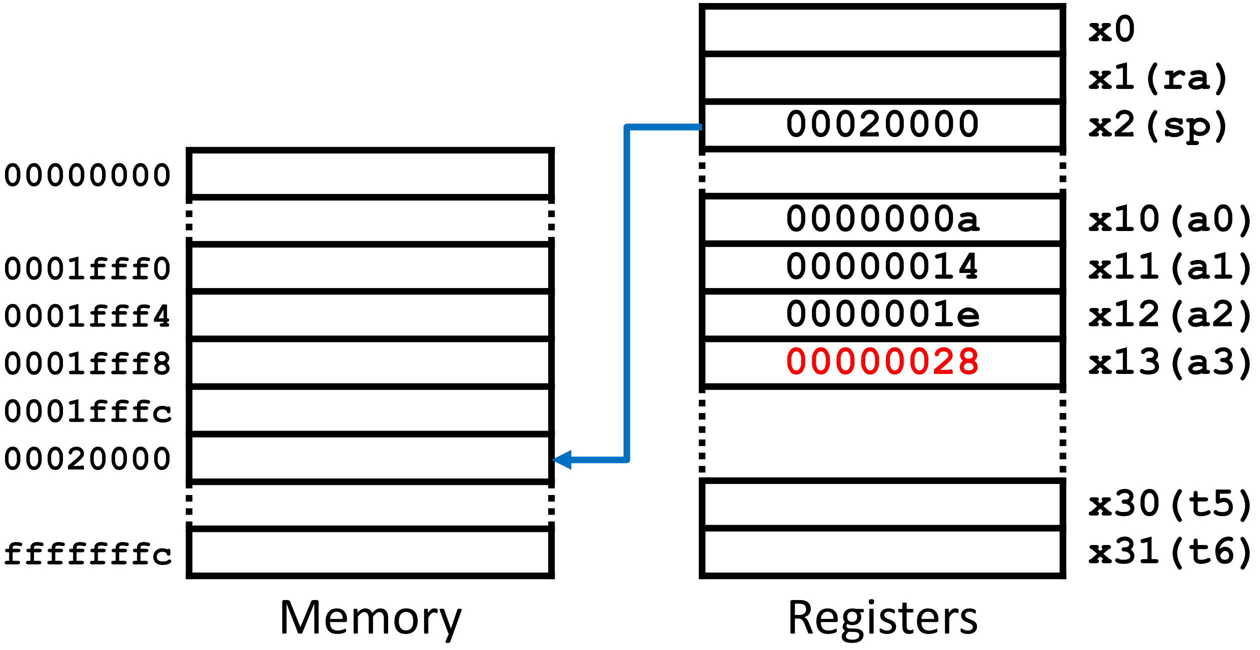
Registers

Functions

Local variables (ii)

ASM

```
          ...
          li    sp, 0x20000
          li    a0, 10
          li    a1, 20
          li    a2, 30
          li    a3, 40
          call  baz
0000d40c  ...
      baz:
          add   sp, sp, -8
          add   t6, a0, a1
          sw    t6, 4(sp)
          add   t6, a2, a3
          sw    t6, 0(sp)
          ...
          lw    t5, 4(sp)
          lw    t6, 0(sp)
          sub   a0, t5, t6
          add   sp, sp, 8
          ret
          ...
```



00000000	
0001fff0	
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
	x1(ra)
00020000	x2(sp)
0000000a	x10(a0)
00000014	x11(a1)
0000001e	x12(a2)
00000028	x13(a3)
	x30(t5)
	x31(t6)

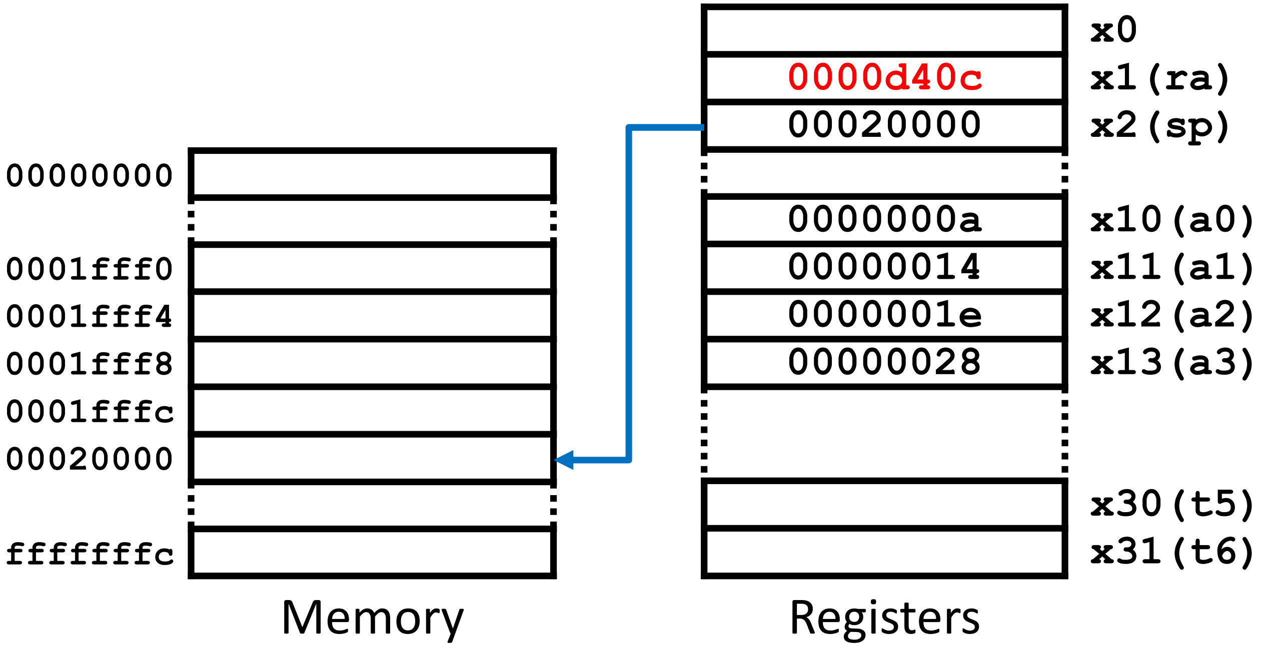
Registers

Functions

Local variables (ii)

ASM

```
          ...
          li    sp, 0x20000
          li    a0, 10
          li    a1, 20
          li    a2, 30
          li    a3, 40
     ⇨    call  baz
0000d40c  ...
        baz:
          add   sp, sp, -8
          add   t6, a0, a1
          sw    t6, 4(sp)
          add   t6, a2, a3
          sw    t6, 0(sp)
          ...
          lw    t5, 4(sp)
          lw    t6, 0(sp)
          sub   a0, t5, t6
          add   sp, sp, 8
          ret
          ...
```

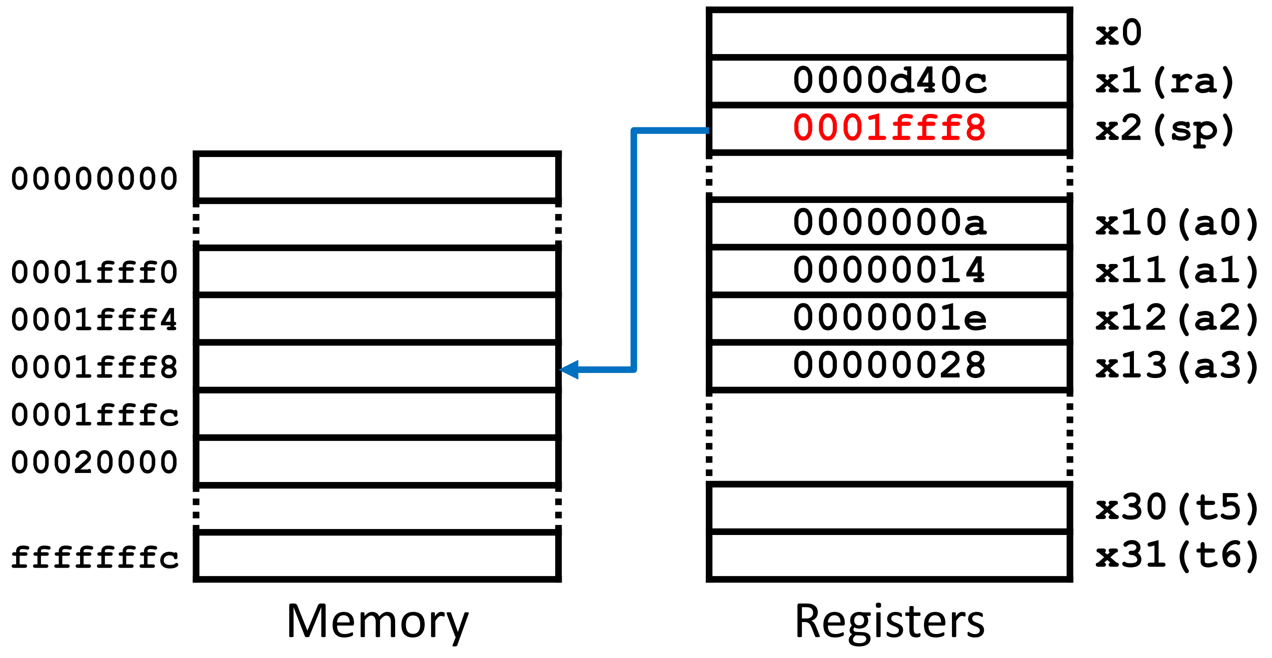


Functions

Local variables (ii)

ASM

```
         ...
         li    sp, 0x20000
         li    a0, 10
         li    a1, 20
         li    a2, 30
         li    a3, 40
         call  baz
0000d40c ...
baz:
         add   sp, sp, -8
         add   t6, a0, a1
         sw    t6, 4(sp)
         add   t6, a2, a3
         sw    t6, 0(sp)
         ...
         lw    t5, 4(sp)
         lw    t6, 0(sp)
         sub   a0, t5, t6
         add   sp, sp, 8
         ret
         ...
```



00000000	
⋮	
0001fff0	
0001fff4	
0001fff8	
0001fffc	
00020000	
⋮	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001fff8	x2(sp)
⋮	
0000000a	x10(a0)
00000014	x11(a1)
0000001e	x12(a2)
00000028	x13(a3)
⋮	
	x30(t5)
	x31(t6)

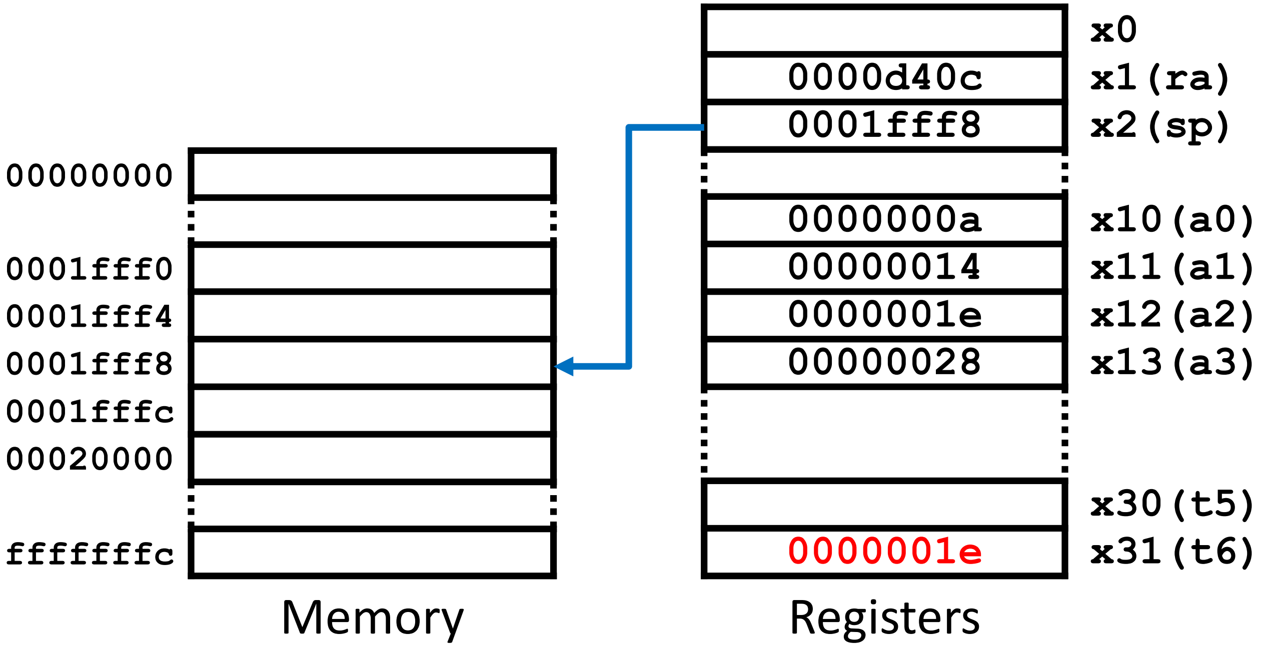
Registers

Functions

Local variables (ii)

ASM

```
         ...
         li   sp, 0x20000
         li   a0, 10
         li   a1, 20
         li   a2, 30
         li   a3, 40
         call baz
0000d40c ...
      baz:
         add  sp, sp, -8
         add  t6, a0, a1
         sw   t6, 4(sp)
         add  t6, a2, a3
         sw   t6, 0(sp)
         ...
         lw   t5, 4(sp)
         lw   t6, 0(sp)
         sub  a0, t5, t6
         add  sp, sp, 8
         ret
         ...
```



00000000	
0001fff0	
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001fff8	x2(sp)
0000000a	x10(a0)
00000014	x11(a1)
0000001e	x12(a2)
00000028	x13(a3)
	x30(t5)
0000001e	x31(t6)

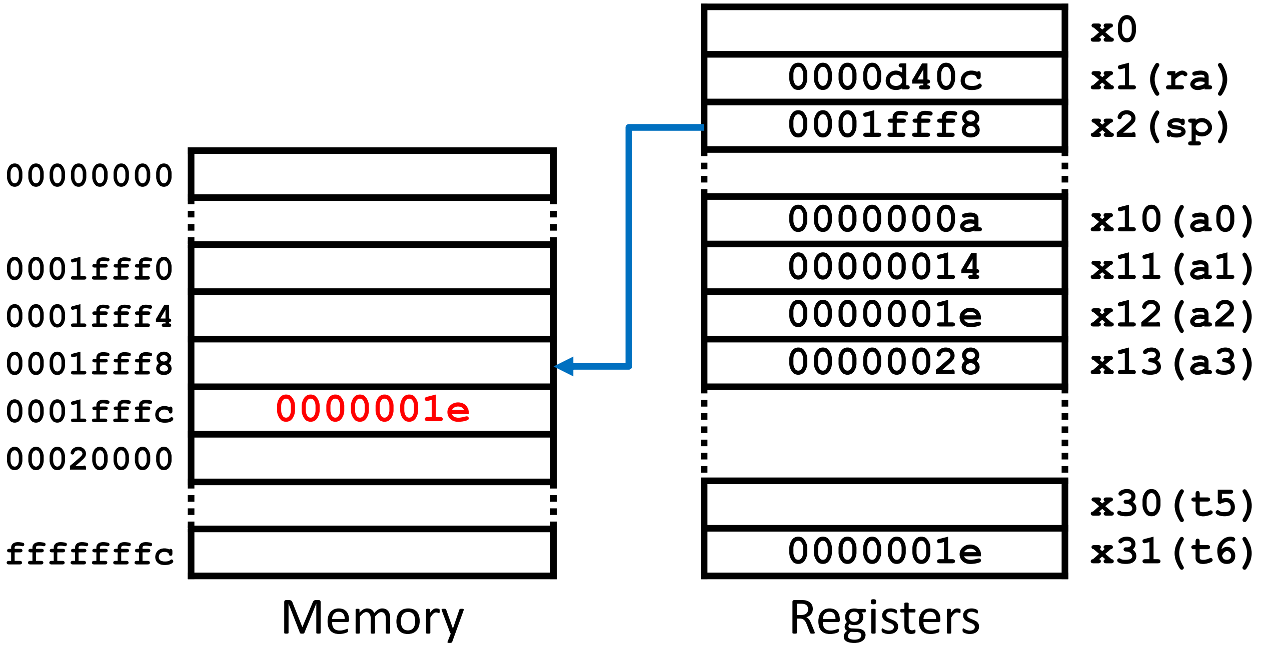
Registers

Functions

Local variables (ii)

ASM

```
         ...
         li    sp, 0x20000
         li    a0, 10
         li    a1, 20
         li    a2, 30
         li    a3, 40
         call  baz
0000d40c ...
       baz:
         add   sp, sp, -8
         add   t6, a0, a1
         sw    t6, 4(sp)
         add   t6, a2, a3
         sw    t6, 0(sp)
         ...
         lw    t5, 4(sp)
         lw    t6, 0(sp)
         sub   a0, t5, t6
         add   sp, sp, 8
         ret
         ...
```

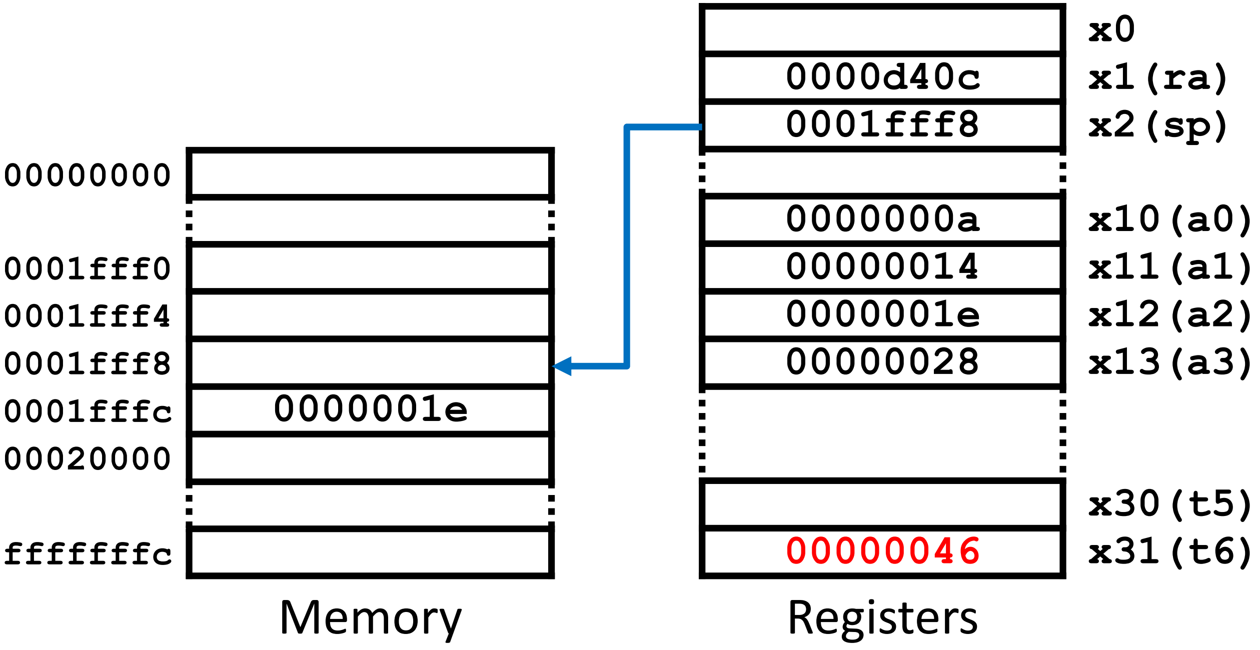


Functions

Local variables (ii)

ASM

```
         ...
         li    sp, 0x20000
         li    a0, 10
         li    a1, 20
         li    a2, 30
         li    a3, 40
         call  baz
0000d40c ...
       baz:
         add   sp, sp, -8
         add   t6, a0, a1
         sw    t6, 4(sp)
  ⇨      add   t6, a2, a3
         sw    t6, 0(sp)
         ...
         lw    t5, 4(sp)
         lw    t6, 0(sp)
         sub   a0, t5, t6
         add   sp, sp, 8
         ret
         ...
```



00000000	
⋮	⋮
0001fff0	
0001fff4	
0001fff8	
0001fffc	0000001e
00020000	
⋮	⋮
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001fff8	x2(sp)
⋮	
0000000a	x10(a0)
00000014	x11(a1)
0000001e	x12(a2)
00000028	x13(a3)
⋮	
	x30(t5)
00000046	x31(t6)

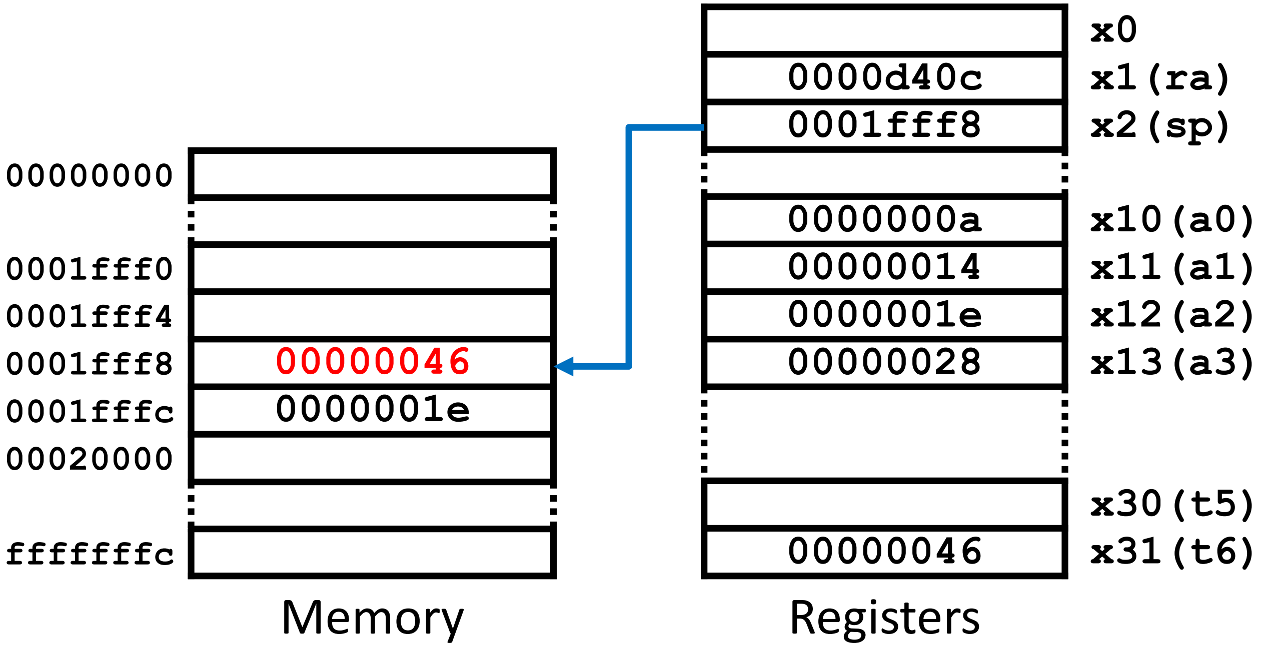
Registers

Functions

Local variables (ii)

ASM

```
          ...
          li    sp, 0x20000
          li    a0, 10
          li    a1, 20
          li    a2, 30
          li    a3, 40
          call  baz
0000d40c  ...
        baz:
          add   sp, sp, -8
          add   t6, a0, a1
          sw    t6, 4(sp)
          add   t6, a2, a3
      ⇨   sw    t6, 0(sp)
          ...
          lw    t5, 4(sp)
          lw    t6, 0(sp)
          sub   a0, t5, t6
          add   sp, sp, 8
          ret
          ...
```



00000000	
0001fff0	
0001fff4	
0001fff8	00000046
0001fffc	0000001e
00020000	
fffffffc	

Memory

	x0
0000d40c	x1 (ra)
0001fff8	x2 (sp)
0000000a	x10 (a0)
00000014	x11 (a1)
0000001e	x12 (a2)
00000028	x13 (a3)
	x30 (t5)
00000046	x31 (t6)

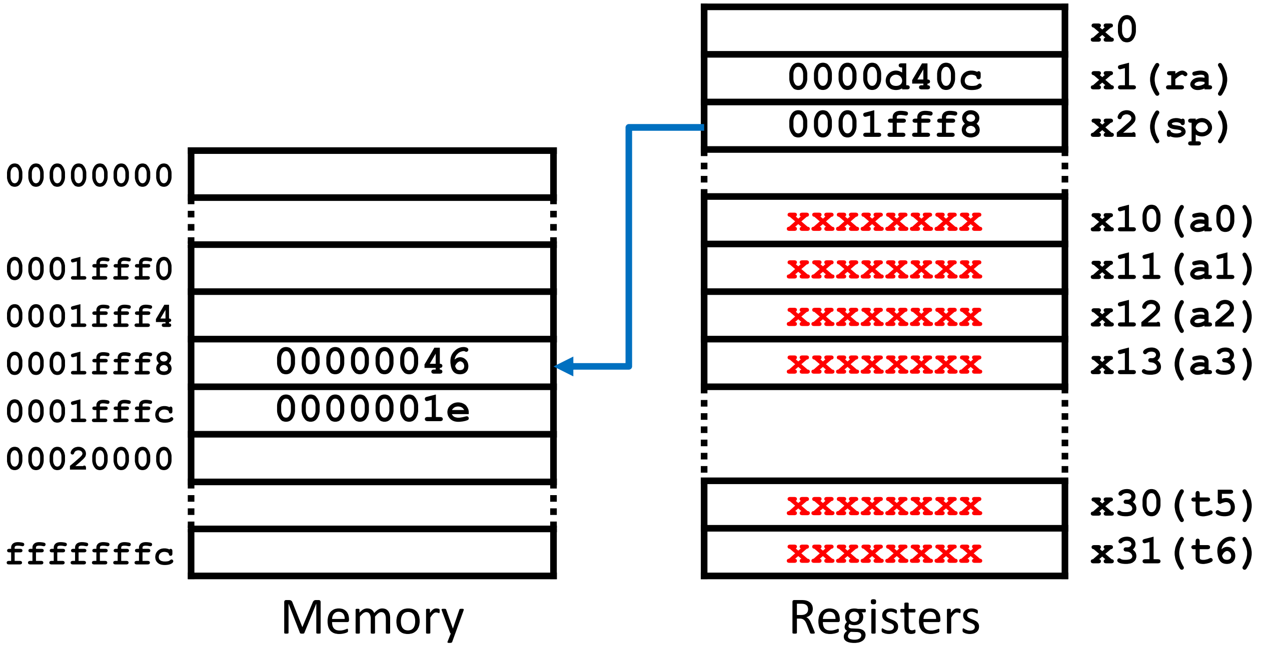
Registers

Functions

Local variables (ii)

ASM

```
         ...
         li    sp, 0x20000
         li    a0, 10
         li    a1, 20
         li    a2, 30
         li    a3, 40
         call  baz
0000d40c ...
      baz:
         add   sp, sp, -8
         add   t6, a0, a1
         sw    t6, 4(sp)
         add   t6, a2, a3
         sw    t6, 0(sp)
         ...
         lw    t5, 4(sp)
         lw    t6, 0(sp)
         sub   a0, t5, t6
         add   sp, sp, 8
         ret
         ...
```



00000000	
0001fff0	
0001fff4	
0001fff8	00000046
0001fffc	0000001e
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001fff8	x2(sp)
xxxxxxxx	x10(a0)
xxxxxxxx	x11(a1)
xxxxxxxx	x12(a2)
xxxxxxxx	x13(a3)
xxxxxxxx	x30(t5)
xxxxxxxx	x31(t6)

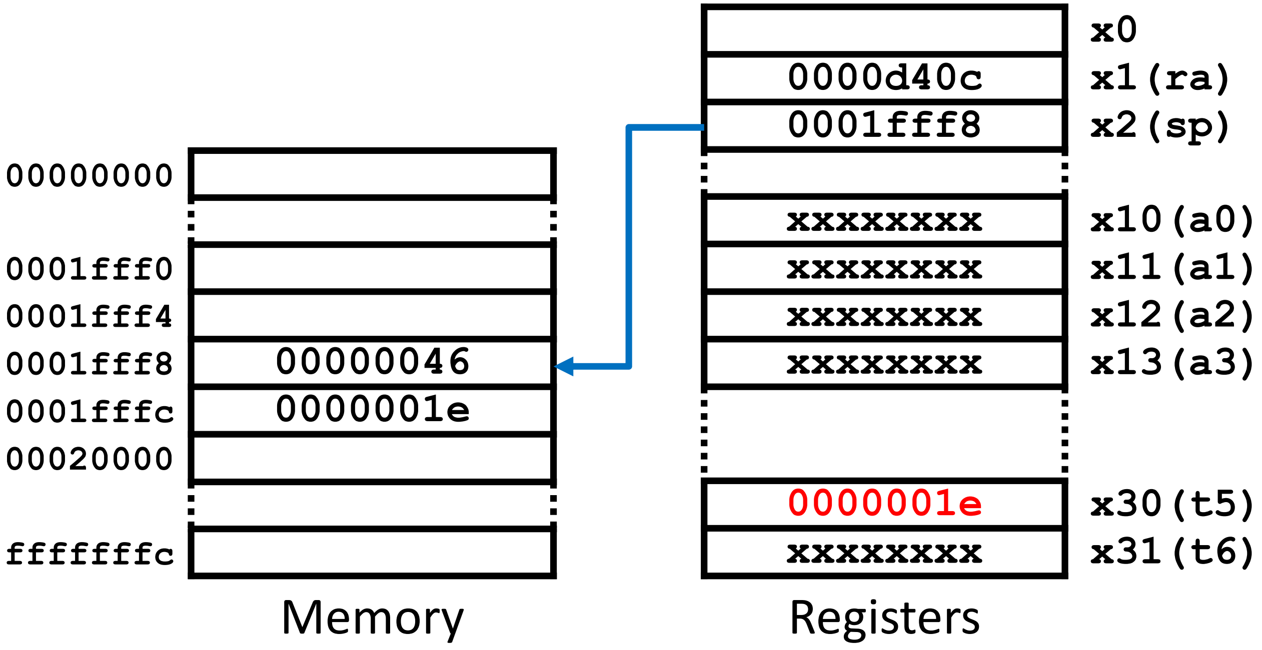
Registers

Functions

Local variables (ii)

ASM

```
         ...
         li   sp, 0x20000
         li   a0, 10
         li   a1, 20
         li   a2, 30
         li   a3, 40
         call baz
0000d40c ...
       baz:
         add  sp, sp, -8
         add  t6, a0, a1
         sw   t6, 4(sp)
         add  t6, a2, a3
         sw   t6, 0(sp)
         ...
  ⇒      lw   t5, 4(sp)
         lw   t6, 0(sp)
         sub  a0, t5, t6
         add  sp, sp, 8
         ret
         ...
```

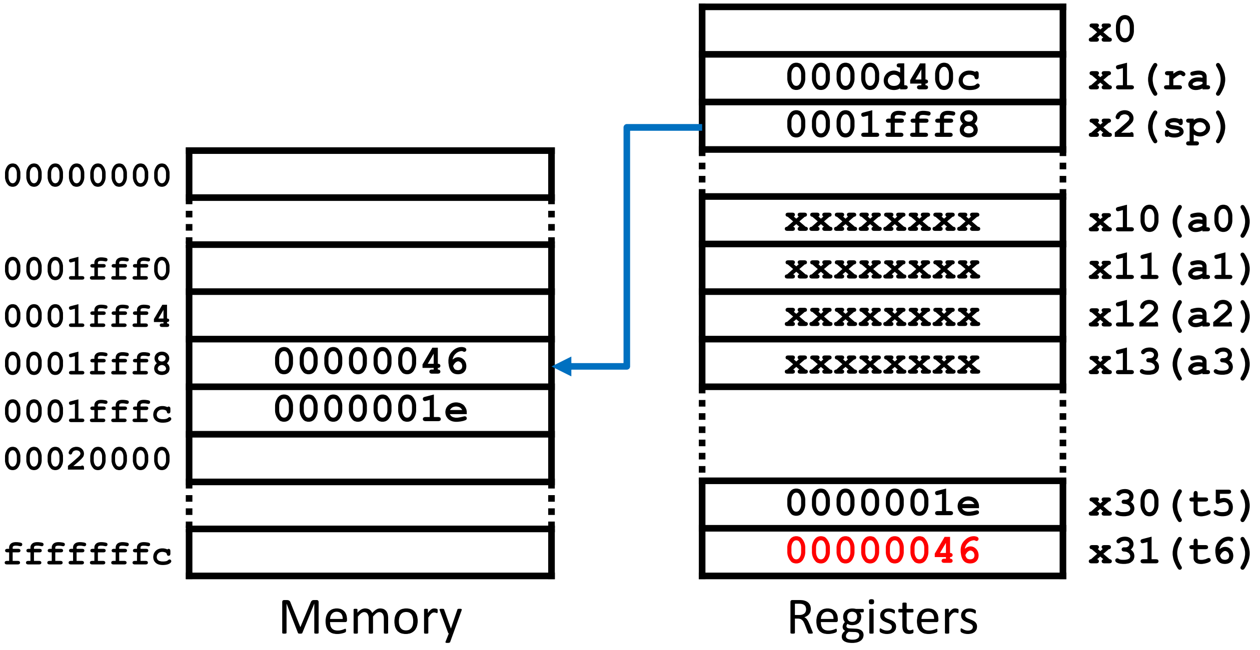


Functions

Local variables (ii)

ASM

```
         ...
         li    sp, 0x20000
         li    a0, 10
         li    a1, 20
         li    a2, 30
         li    a3, 40
         call  baz
0000d40c ...
      baz:
         add   sp, sp, -8
         add   t6, a0, a1
         sw    t6, 4(sp)
         add   t6, a2, a3
         sw    t6, 0(sp)
         ...
         lw    t5, 4(sp)
    ⇨    lw    t6, 0(sp)
         sub   a0, t5, t6
         add   sp, sp, 8
         ret
         ...
```



Memory

00000000	
⋮	⋮
0001fff0	
0001fff4	
0001fff8	00000046
0001fffc	0000001e
00020000	
⋮	⋮
fffffffc	

Registers

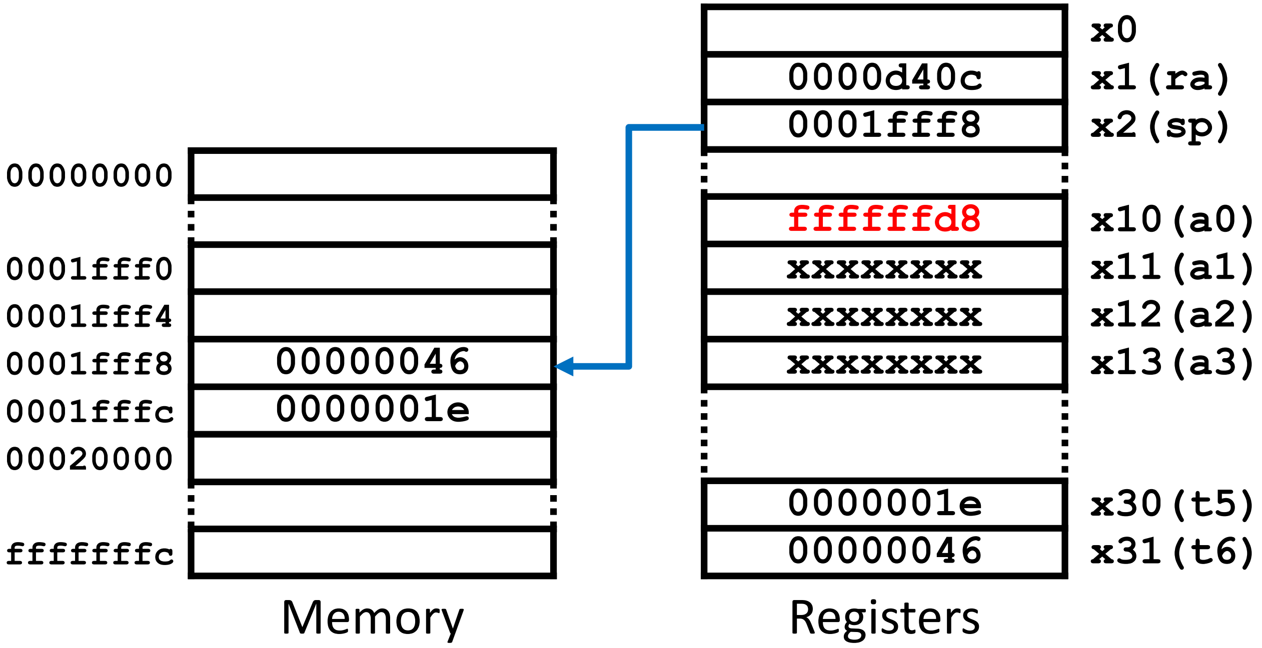
	x0
0000d40c	x1(ra)
0001fff8	x2(sp)
⋮	
xxxxxxxx	x10(a0)
xxxxxxxx	x11(a1)
xxxxxxxx	x12(a2)
xxxxxxxx	x13(a3)
⋮	
0000001e	x30(t5)
00000046	x31(t6)

Functions

Local variables (ii)

ASM

```
         ...
         li    sp, 0x20000
         li    a0, 10
         li    a1, 20
         li    a2, 30
         li    a3, 40
         call  baz
0000d40c ...
       baz:
         add   sp, sp, -8
         add   t6, a0, a1
         sw    t6, 4(sp)
         add   t6, a2, a3
         sw    t6, 0(sp)
         ...
         lw    t5, 4(sp)
         lw    t6, 0(sp)
         sub   a0, t5, t6
         add   sp, sp, 8
         ret
         ...
```



00000000	
0001fff0	
0001fff4	
0001fff8	00000046
0001fffc	0000001e
00020000	
fffffffc	

Memory

	x0
0000d40c	x1 (ra)
0001fff8	x2 (sp)
ffffffd8	x10 (a0)
xxxxxxxx	x11 (a1)
xxxxxxxx	x12 (a2)
xxxxxxxx	x13 (a3)
0000001e	x30 (t5)
00000046	x31 (t6)

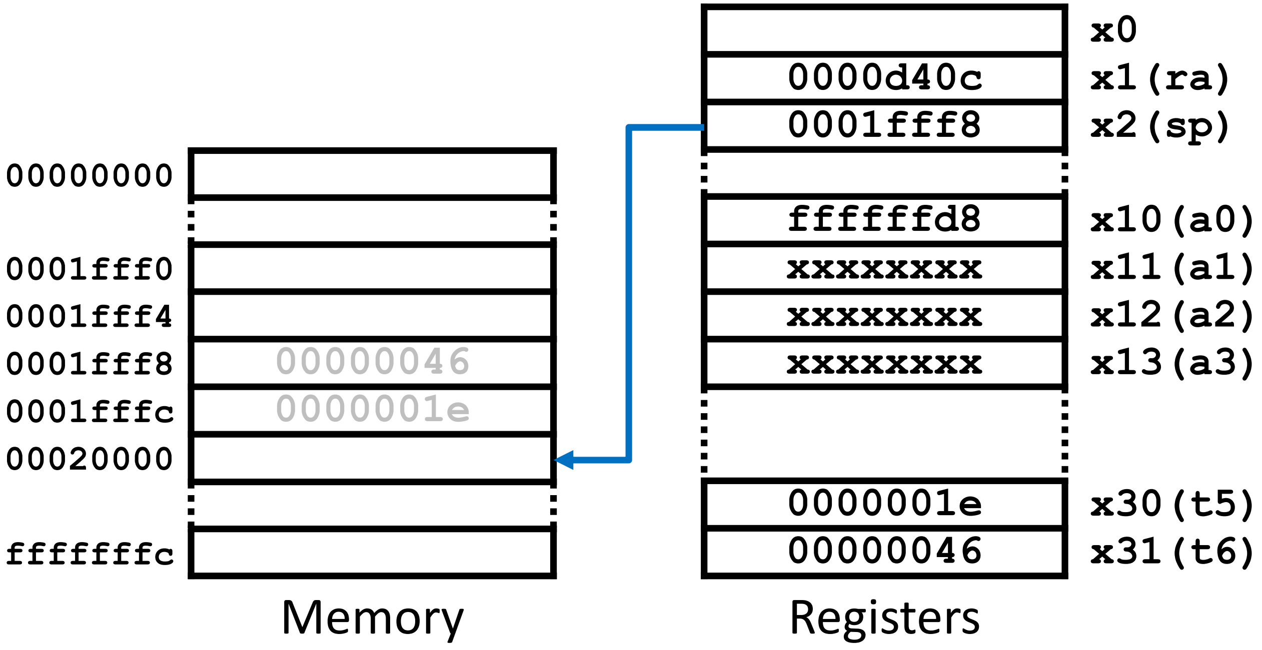
Registers

Functions

Local variables (ii)

ASM

```
          ...
          li    sp, 0x20000
          li    a0, 10
          li    a1, 20
          li    a2, 30
          li    a3, 40
          call  baz
0000d40c  ...
        baz:
          add   sp, sp, -8
          add   t6, a0, a1
          sw    t6, 4(sp)
          add   t6, a2, a3
          sw    t6, 0(sp)
          ...
          lw    t5, 4(sp)
          lw    t6, 0(sp)
          sub   a0, t5, t6
          add   sp, sp, 8
          ret
          ...
```



00000000	
0001fff0	
0001fff4	
0001fff8	00000046
0001fffc	0000001e
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001fff8	x2(sp)
ffffffd8	x10(a0)
xxxxxxxx	x11(a1)
xxxxxxxx	x12(a2)
xxxxxxxx	x13(a3)
0000001e	x30(t5)
00000046	x31(t6)

Registers

Functions

Local variables (iii)

- **`sp`** may be used as the base register, but it is risky if the stack changes during the execution of the function:
  - References to the same local variable in the function code might have different offsets relative to **`sp`**, making it hard to read.

C/C++

```
int baz( int a, int b,
         int c, int d )
{
  int sum1, sum2;

  sum1 = (a + b);
  sum2 = (c + d);
  sum1 = foo( 1, 2, 3,
              4, 5, 6,
              7, 8, sum1 );

  ...
  return sum1 - sum2;
}
```

ASM

```
baz:
   add   sp, sp, -8
   add   t6, a0, a1
   sw    t6, 4(sp)
   add   t6, a2, a3
   sw    t6, 0(sp)
   li    a0, 1
   ...
   li    a7, 8
   add   sp, sp, -4
   lw    t0, 8(sp)
   sw    t0, 0(sp)
   call  foo
   add   sp, sp, 4
   ...
   ret
```

`sum1` *is in* **`4(sp)`**

`sum2` *is in* **`0(sp)`**

the first 8 arguments are copied in **`a0-a7`**

space is reserved for the 9th argument

`sum1` *is pushed, being now in* **`8(sp)`***; if* **`4(sp)`** *was used,* **`sum2`** *would be pushed*

After restoring the top of the stack, **`sum1`** *will be in* **`4(sp)`** *again*

Functions

Local variables (iv)

- In order to avoid risks, **fp** is used as the base register.
    - In this way, all local variables will have a constant and unique offset inside the function.
    - **fp** is a preserved register that is initialized to the top of the stack at the beginning of the function and does not change during its execution.

ASM

```
baz:
   add   sp, sp, -8
   add   t6, a0, a1
   sw    t6, 4(sp)
   add   t6, a2, a3
   sw    t6, 0(sp)
   ...
   lw    t5, 4(sp)
   lw    t6, 0(sp)
   sub   a0, t5, t6
   add   sp, sp, 8
   ret
```

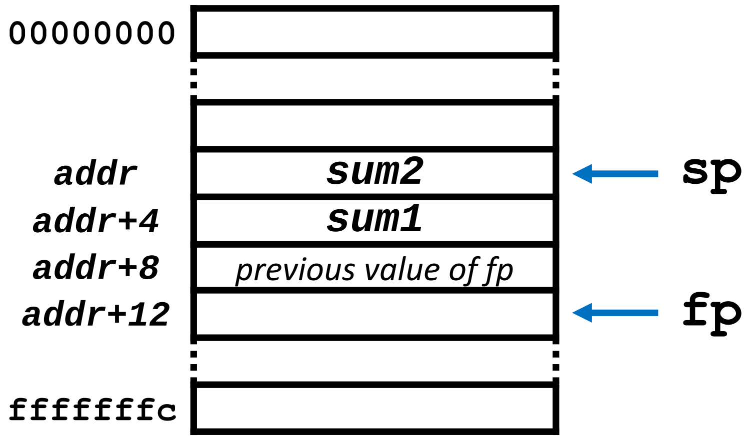
sum1 → 4(sp)
sum2 → 0(sp)

ASM

```
baz:
   add   sp, sp, -12
   sw    fp, 8(sp)
   add   fp, sp, 12
   add   t6, a0, a1
   sw    t6, -8(fp)
   add   t6, a2, a3
   sw    t6, -12(fp)
   ...
   lw    t5, -8(fp)
   lw    t6, -12(fp)
   sub   a0, t5, t6
   add   sp, sp, 12
   ret
```

sum1 → -8(fp)
sum2 → -12(fp)

reserves space for **fp** *and 2 local variables*
pushes **fp** *since it is a preserved register*
initializes **fp**



Memory

Functions

Frame management (i)

- On a function call:

caller function

1. Pushes the temporary registers used by the caller function.
2. Passes the input parameters.
3. Saves the return address.
4. Branches to the initial address of the callee function.

`call` *pseudo-instruction*

callee function

5. Pushes the preserved registers used by the callee function.
6. Reserves space for the local variables of the callee function.
7. Initializes the local variables.

prologue
(frame construction)

8. Processes and updates the output parameters.
9. Saves the return value.

body

10. Frees up the space occupied by the local variables.
11. Pops the preserved registers.

epilogue
(frame destruction)

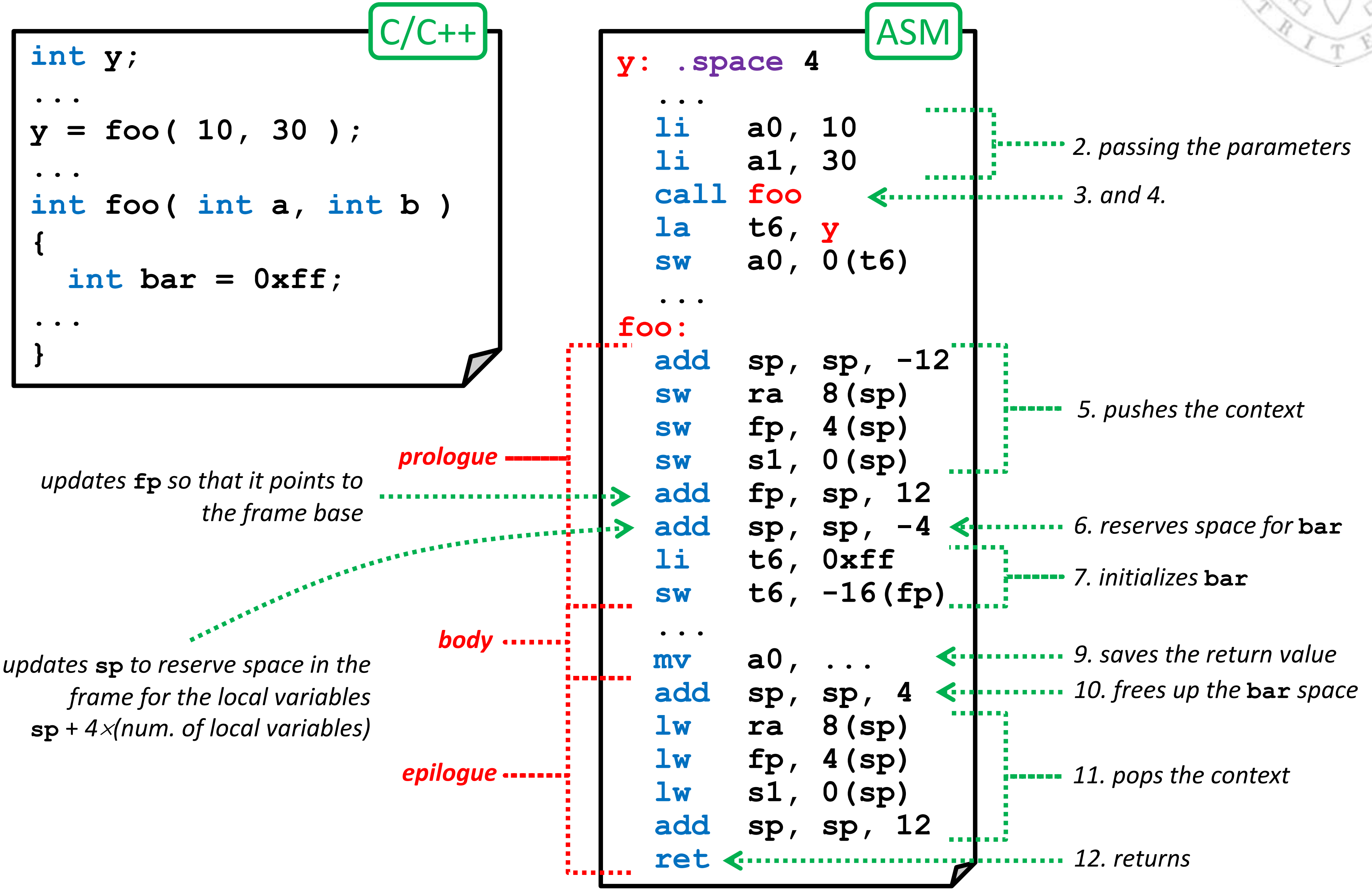
12. Branches to the return address. ← `ret` *pseudo-instruction*

caller function

13. Restores the return value.
14. Pops the temporary registers.

Functions

Frame management (ii)



C/C++

```
int y;
...
y = foo( 10, 30 );
...
int foo( int a, int b )
{
  int bar = 0xff;
...
}
```

ASM

```
y: .space 4
   ...
   li   a0, 10
   li   a1, 30
   call foo
   la   t6, y
   sw   a0, 0(t6)
   ...
foo:
   add  sp, sp, -12
   sw   ra  8(sp)
   sw   fp, 4(sp)
   sw   s1, 0(sp)
   add  fp, sp, 12
   add  sp, sp, -4
   li   t6, 0xff
   sw   t6, -16(fp)
   ...
   mv   a0, ...
   add  sp, sp, 4
   lw   ra  8(sp)
   lw   fp, 4(sp)
   lw   s1, 0(sp)
   add  sp, sp, 12
   ret
```

2. passing the parameters

3. and 4.

5. pushes the context

6. reserves space for **`bar`**

7. initializes **`bar`**

9. saves the return value

10. frees up the **`bar`** space

11. pops the context

12. returns

prologue

body

epilogue

updates **`fp`** *so that it points to the frame base*

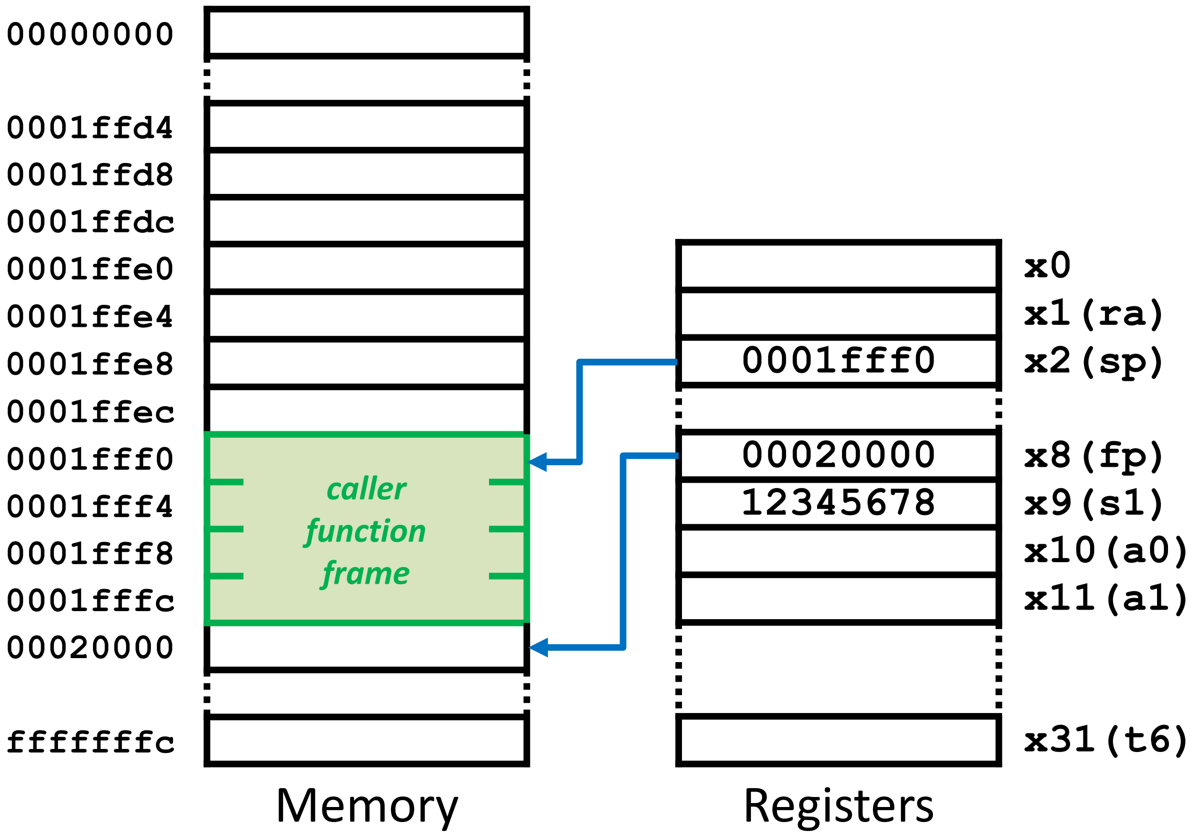
updates **`sp`** *to reserve space in the frame for the local variables* **`sp`** + *4×(num. of local variables)*

Functions

Frame management (iii)

ASM

```
y:  .space 4
    ...
    li    a0, 10
    li    a1, 30
    call  foo
0000d40c la    t6, y
    sw    a0, 0(t6)
    ...
foo:
    add   sp, sp, -12
    sw    ra  8(sp)
    sw    fp, 4(sp)
    sw    s1, 0(sp)
    add   fp, sp, 12
    add   sp, sp, -4
    li    t6, 0xff
    sw    t6, -16(fp)
    ...
    mv    a0, ...
    add   sp, sp, 4
    lw    ra  8(sp)
    lw    fp, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 12
    ret
```

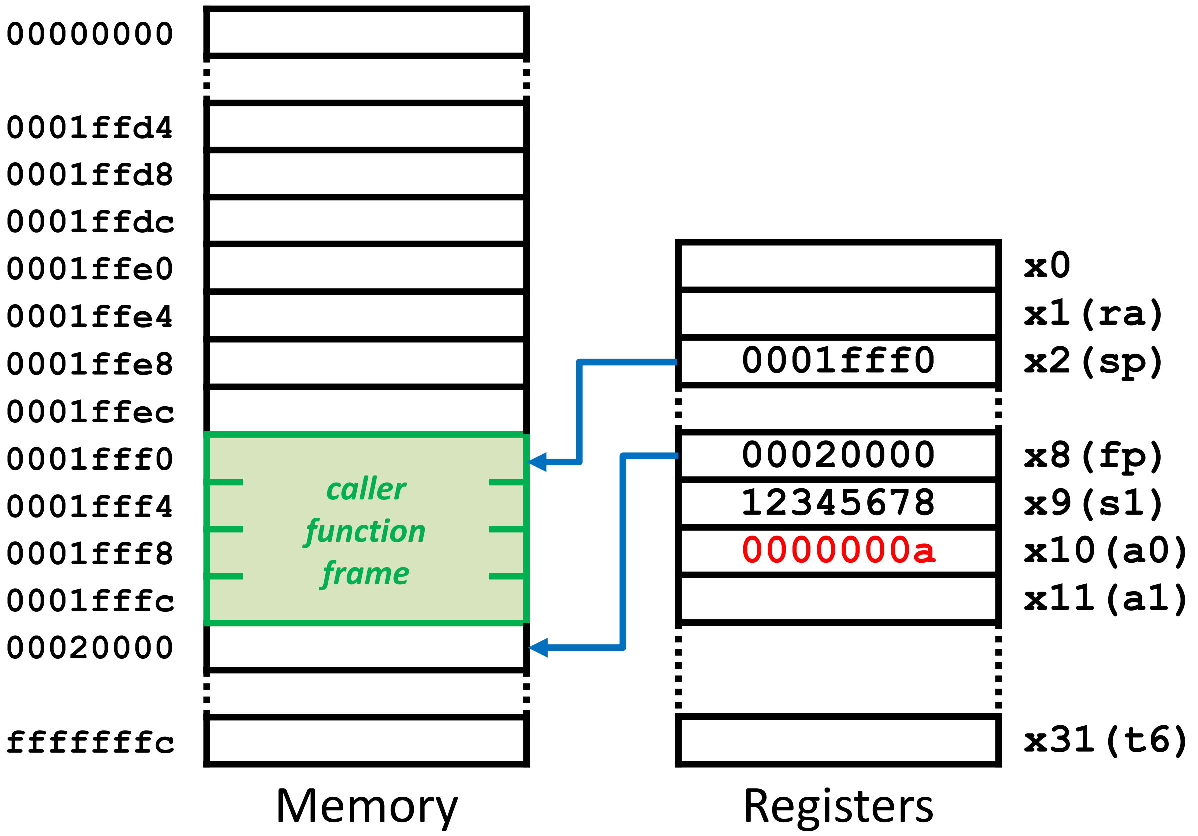


Functions

Frame management (iii)

ASM

```
y:  .space 4
         ...
         li    a0, 10
         li    a1, 30
         call  foo
0000d40c la    t6, y
         sw    a0, 0(t6)
         ...
foo:
         add   sp, sp, -12
         sw    ra  8(sp)
         sw    fp, 4(sp)
         sw    s1, 0(sp)
         add   fp, sp, 12
         add   sp, sp, -4
         li    t6, 0xff
         sw    t6, -16(fp)
         ...
         mv    a0, ...
         add   sp, sp, 4
         lw    ra  8(sp)
         lw    fp, 4(sp)
         lw    s1, 0(sp)
         add   sp, sp, 12
         ret
```



00000000	
0001ffd4	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	caller function frame
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
	x1(ra)
0001fff0	x2(sp)
00020000	x8(fp)
12345678	x9(s1)
0000000a	x10(a0)
	x11(a1)
	x31(t6)

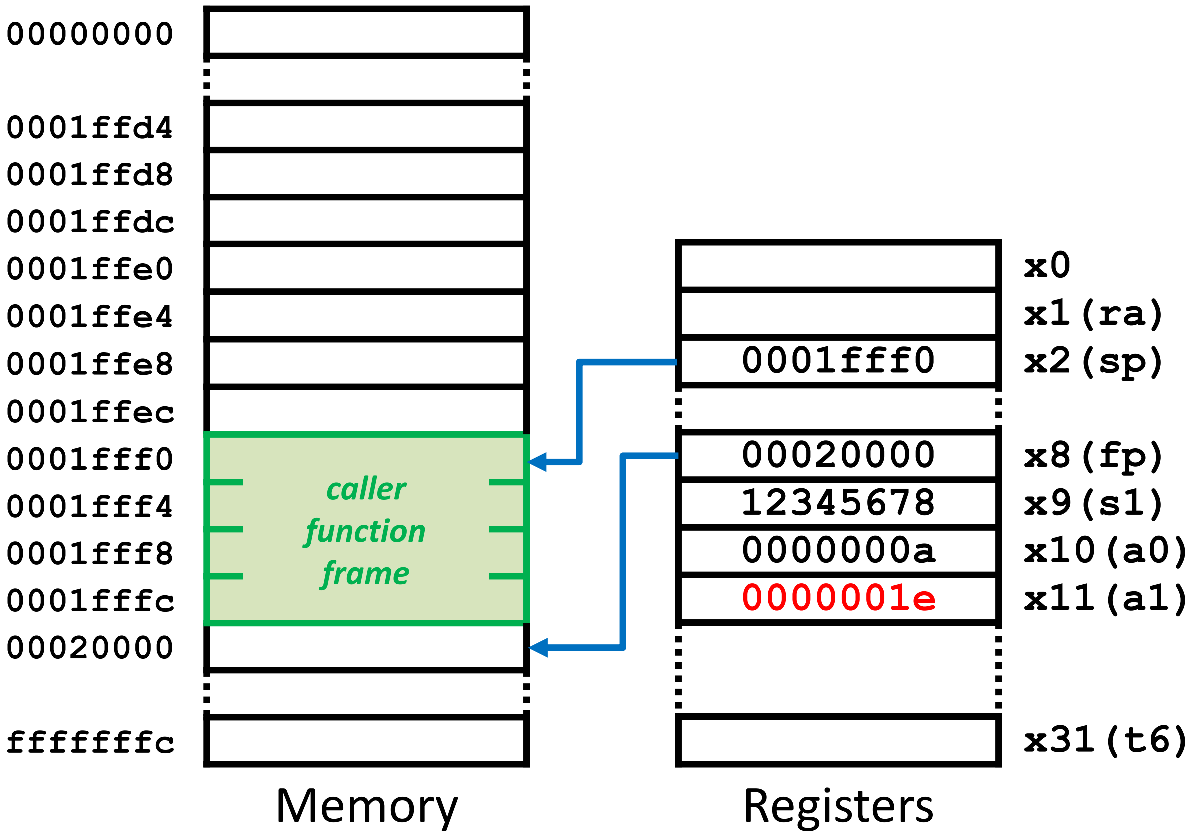
Registers

Functions

Frame management (iii)

ASM

```
y:  .space 4
    ...
    li    a0, 10
    li    a1, 30
    call  foo
0000d40c la    t6, y
    sw    a0, 0(t6)
    ...
foo:
    add   sp, sp, -12
    sw    ra  8(sp)
    sw    fp, 4(sp)
    sw    s1, 0(sp)
    add   fp, sp, 12
    add   sp, sp, -4
    li    t6, 0xff
    sw    t6, -16(fp)
    ...
    mv    a0, ...
    add   sp, sp, 4
    lw    ra  8(sp)
    lw    fp, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 12
    ret
```

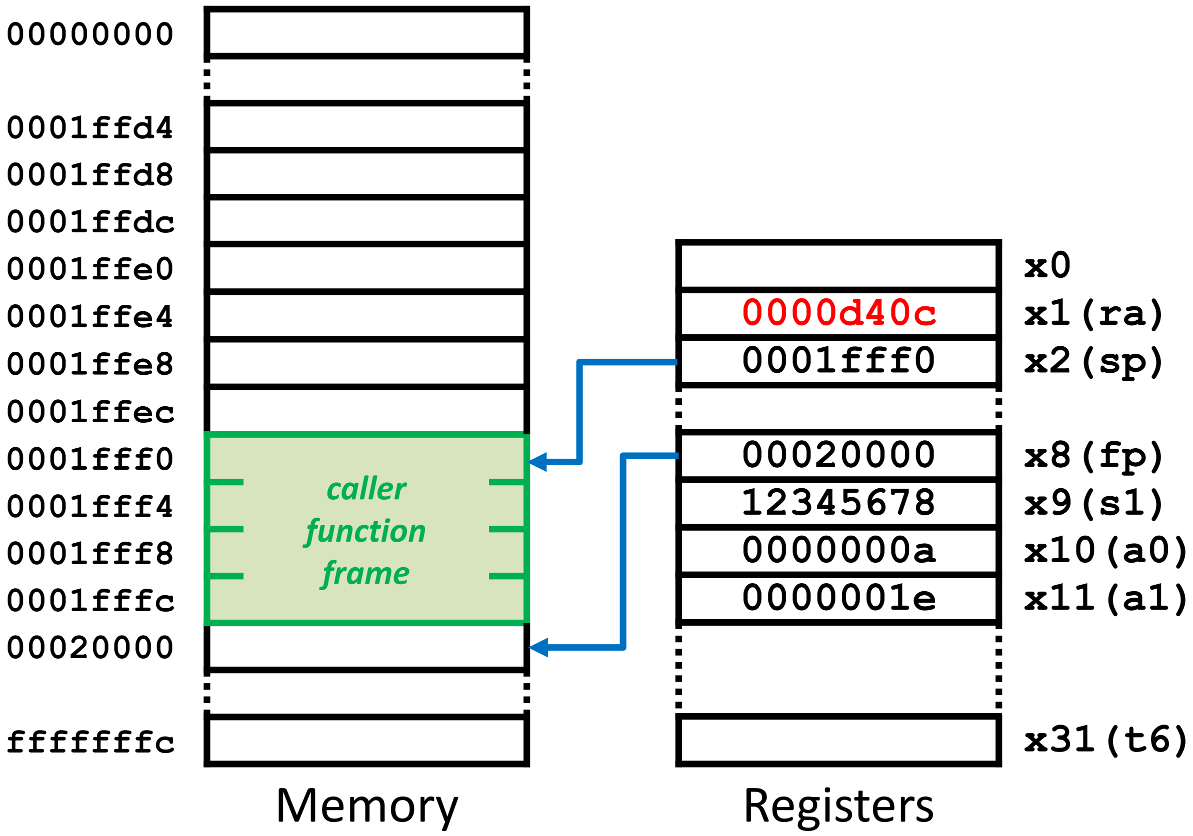


Functions

Frame management (iii)

ASM

```
y:  .space 4
    ...
    li    a0, 10
    li    a1, 30
    call  foo
0000d40c la    t6, y
    sw    a0, 0(t6)
    ...
foo:
    add   sp, sp, -12
    sw    ra  8(sp)
    sw    fp, 4(sp)
    sw    s1, 0(sp)
    add   fp, sp, 12
    add   sp, sp, -4
    li    t6, 0xff
    sw    t6, -16(fp)
    ...
    mv    a0, ...
    add   sp, sp, 4
    lw    ra  8(sp)
    lw    fp, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 12
    ret
```



00000000	
0001ffd4	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	
0001fff0	caller function frame
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001fff0	x2(sp)
00020000	x8(fp)
12345678	x9(s1)
0000000a	x10(a0)
0000001e	x11(a1)
	x31(t6)

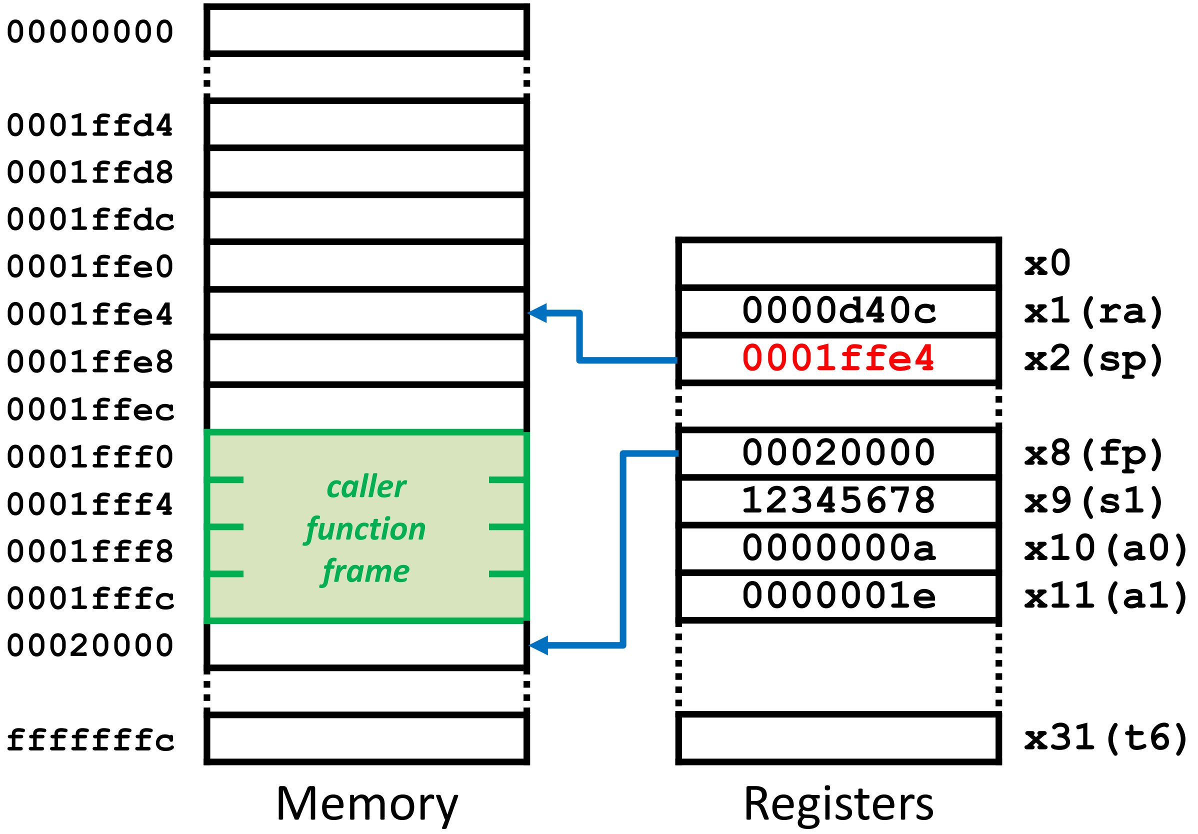
Registers

Functions

Frame management (iii)

ASM

```
y:  .space 4
         ...
         li   a0, 10
         li   a1, 30
         call foo
0000d40c la   t6, y
         sw   a0, 0(t6)
         ...
foo:
⇨        add  sp, sp, -12
         sw   ra  8(sp)
         sw   fp, 4(sp)
         sw   s1, 0(sp)
         add  fp, sp, 12
         add  sp, sp, -4
         li   t6, 0xff
         sw   t6, -16(fp)
         ...
         mv   a0, ...
         add  sp, sp, 4
         lw   ra  8(sp)
         lw   fp, 4(sp)
         lw   s1, 0(sp)
         add  sp, sp, 12
         ret
```

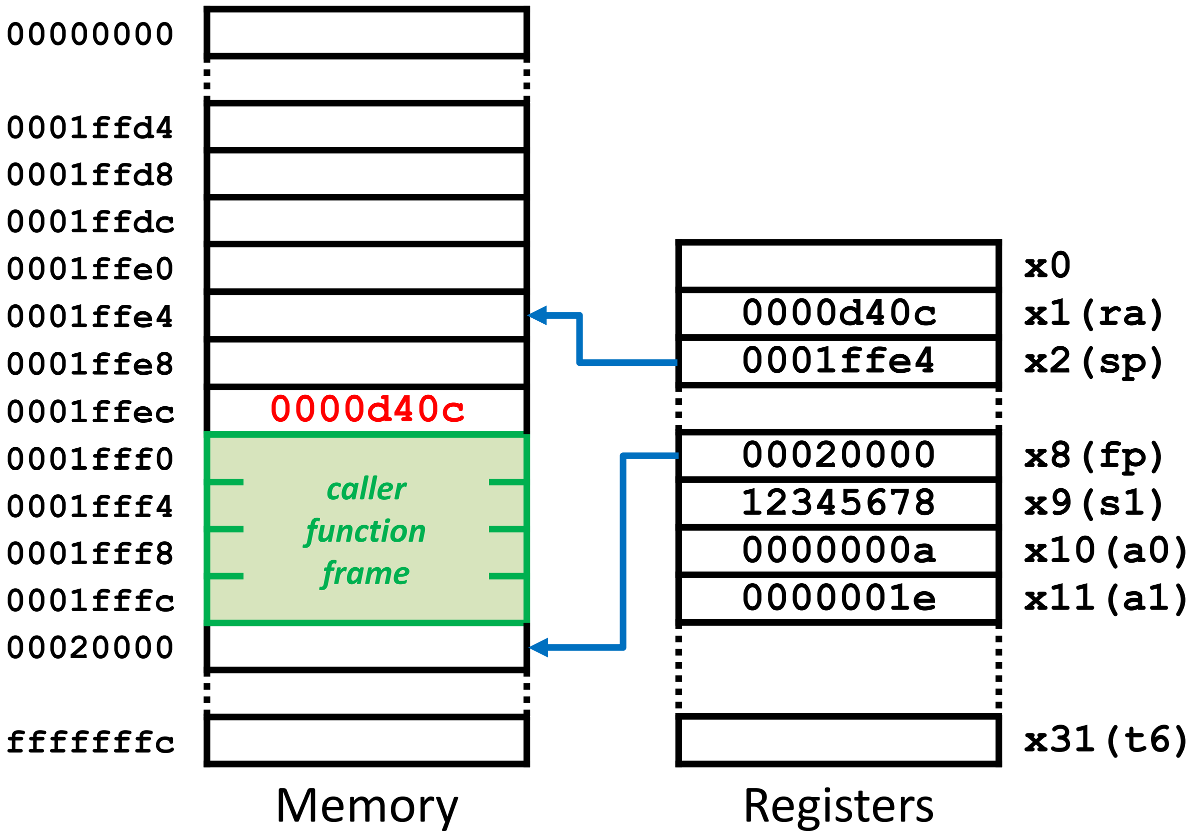


Functions

Frame management (iii)

ASM

```
         y:  .space 4
             ...
             li   a0, 10
             li   a1, 30
             call foo
0000d40c     la   t6, y
             sw   a0, 0(t6)
             ...
         foo:
             add  sp, sp, -12
             sw   ra  8(sp)
             sw   fp, 4(sp)
             sw   s1, 0(sp)
             add  fp, sp, 12
             add  sp, sp, -4
             li   t6, 0xff
             sw   t6, -16(fp)
             ...
             mv   a0, ...
             add  sp, sp, 4
             lw   ra  8(sp)
             lw   fp, 4(sp)
             lw   s1, 0(sp)
             add  sp, sp, 12
             ret
```



00000000	
0001ffd4	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	
0001ffec	0000d40c
0001fff0	caller function frame
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
0000d40c	x1 (ra)
0001ffe4	x2 (sp)
00020000	x8 (fp)
12345678	x9 (s1)
0000000a	x10 (a0)
0000001e	x11 (a1)
	x31 (t6)

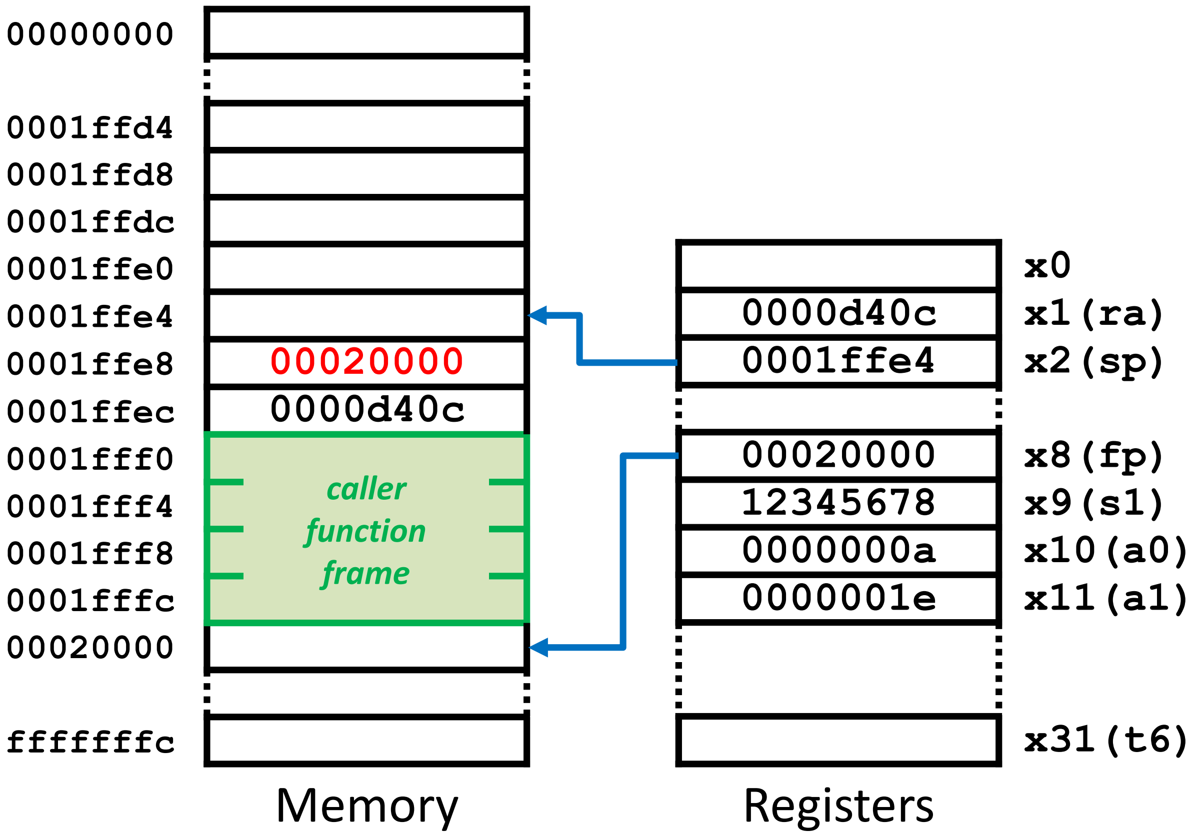
Registers

Functions

Frame management (iii)

ASM

```
         y:  .space 4
             ...
             li    a0, 10
             li    a1, 30
             call  foo
0000d40c     la    t6, y
             sw    a0, 0(t6)
             ...
         foo:
             add   sp, sp, -12
             sw    ra  8(sp)
      ⇨      sw    fp, 4(sp)
             sw    s1, 0(sp)
             add   fp, sp, 12
             add   sp, sp, -4
             li    t6, 0xff
             sw    t6, -16(fp)
             ...
             mv    a0, ...
             add   sp, sp, 4
             lw    ra  8(sp)
             lw    fp, 4(sp)
             lw    s1, 0(sp)
             add   sp, sp, 12
             ret
```



00000000	
0001ffd4	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	
0001ffe8	00020000
0001ffec	0000d40c
0001fff0	caller function frame
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001ffe4	x2(sp)
00020000	x8(fp)
12345678	x9(s1)
0000000a	x10(a0)
0000001e	x11(a1)
	x31(t6)

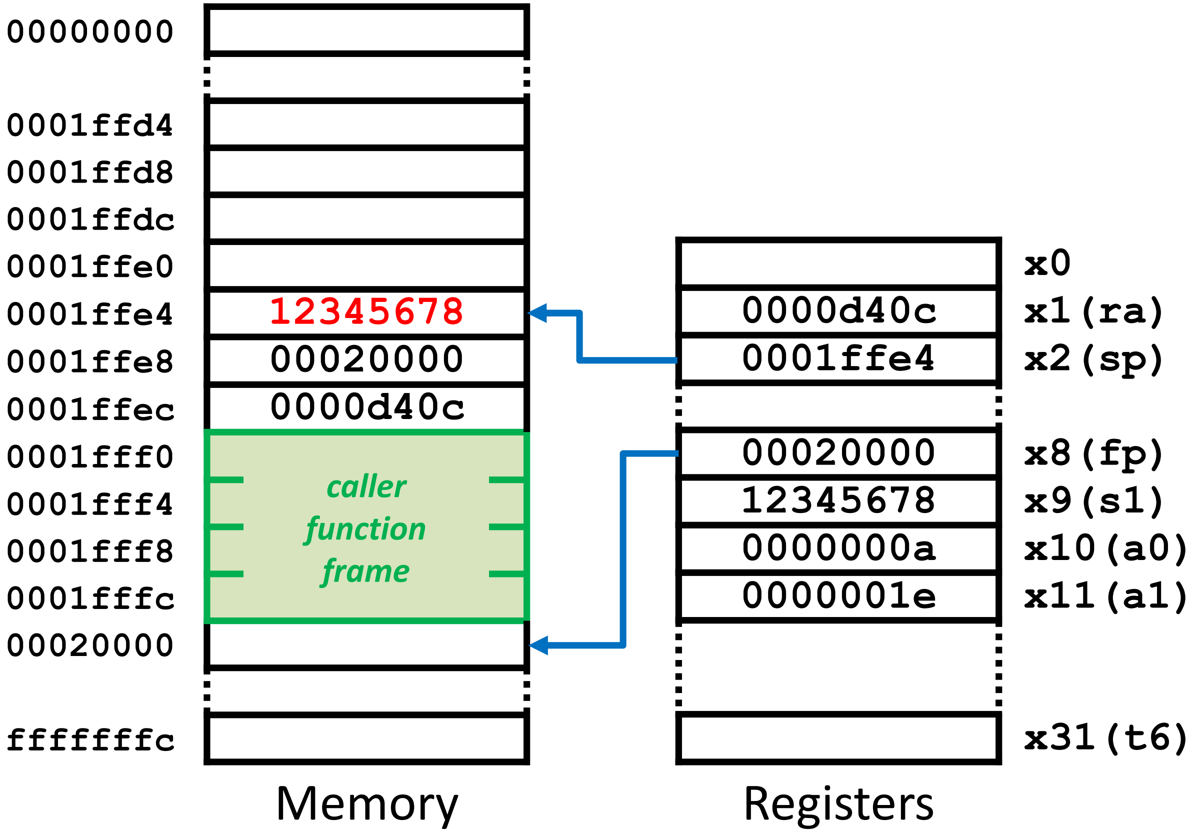
Registers

Functions

Frame management (iii)

ASM

```
         y:  .space 4
                ...
                li    a0, 10
                li    a1, 30
                call  foo
0000d40c        la    t6, y
                sw    a0, 0(t6)
                ...
         foo:
                add   sp, sp, -12
                sw    ra  8(sp)
                sw    fp, 4(sp)
            ⇨   sw    s1, 0(sp)
                add   fp, sp, 12
                add   sp, sp, -4
                li    t6, 0xff
                sw    t6, -16(fp)
                ...
                mv    a0, ...
                add   sp, sp, 4
                lw    ra  8(sp)
                lw    fp, 4(sp)
                lw    s1, 0(sp)
                add   sp, sp, 12
                ret
```



00000000	
0001ffd4	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	12345678
0001ffe8	00020000
0001ffec	0000d40c
0001fff0	caller function frame
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001ffe4	x2(sp)
00020000	x8(fp)
12345678	x9(s1)
0000000a	x10(a0)
0000001e	x11(a1)
	x31(t6)

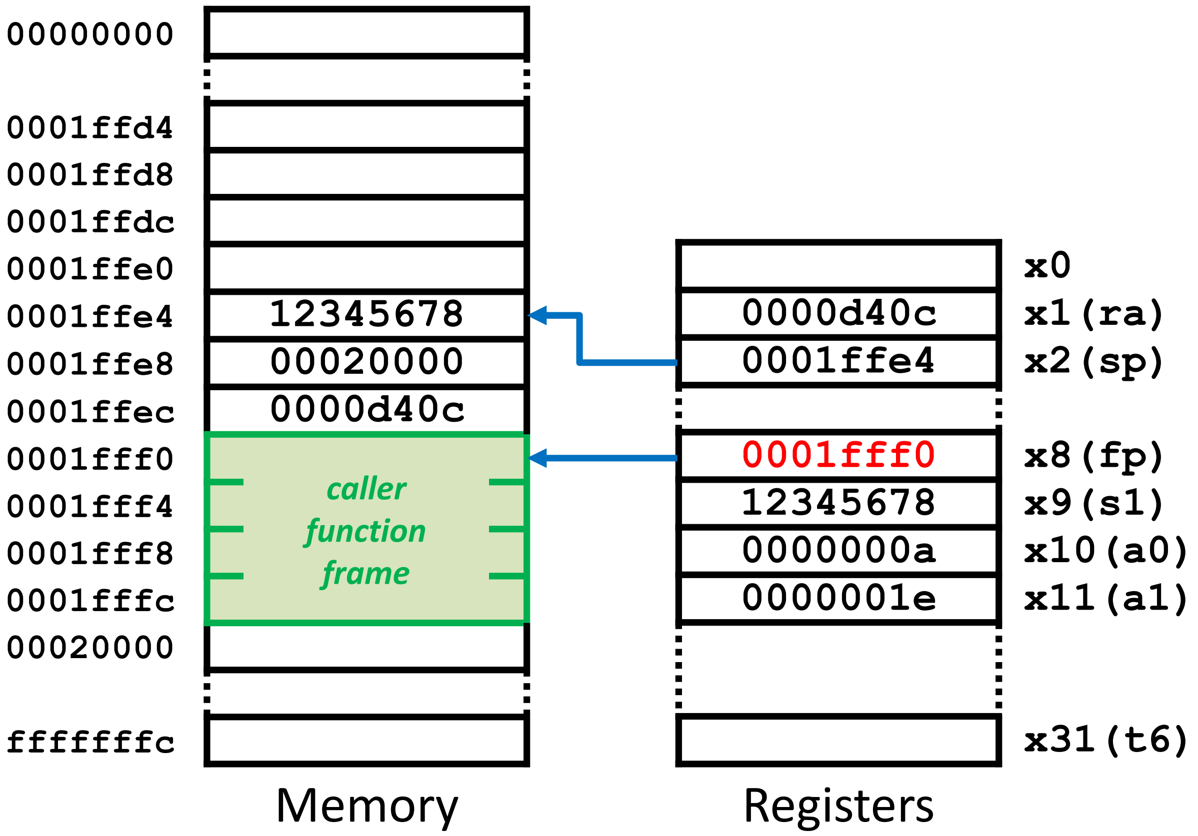
Registers

Functions

Frame management (iii)

ASM

```
         y:  .space 4
             ...
             li    a0, 10
             li    a1, 30
             call  foo
0000d40c     la    t6, y
             sw    a0, 0(t6)
             ...
         foo:
             add   sp, sp, -12
             sw    ra  8(sp)
             sw    fp, 4(sp)
             sw    s1, 0(sp)
             add   fp, sp, 12
             add   sp, sp, -4
             li    t6, 0xff
             sw    t6, -16(fp)
             ...
             mv    a0, ...
             add   sp, sp, 4
             lw    ra  8(sp)
             lw    fp, 4(sp)
             lw    s1, 0(sp)
             add   sp, sp, 12
             ret
```



00000000	
0001ffd4	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	12345678
0001ffe8	00020000
0001ffec	0000d40c
0001fff0	*caller function frame*
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001ffe4	x2(sp)
0001fff0	x8(fp)
12345678	x9(s1)
0000000a	x10(a0)
0000001e	x11(a1)
	x31(t6)

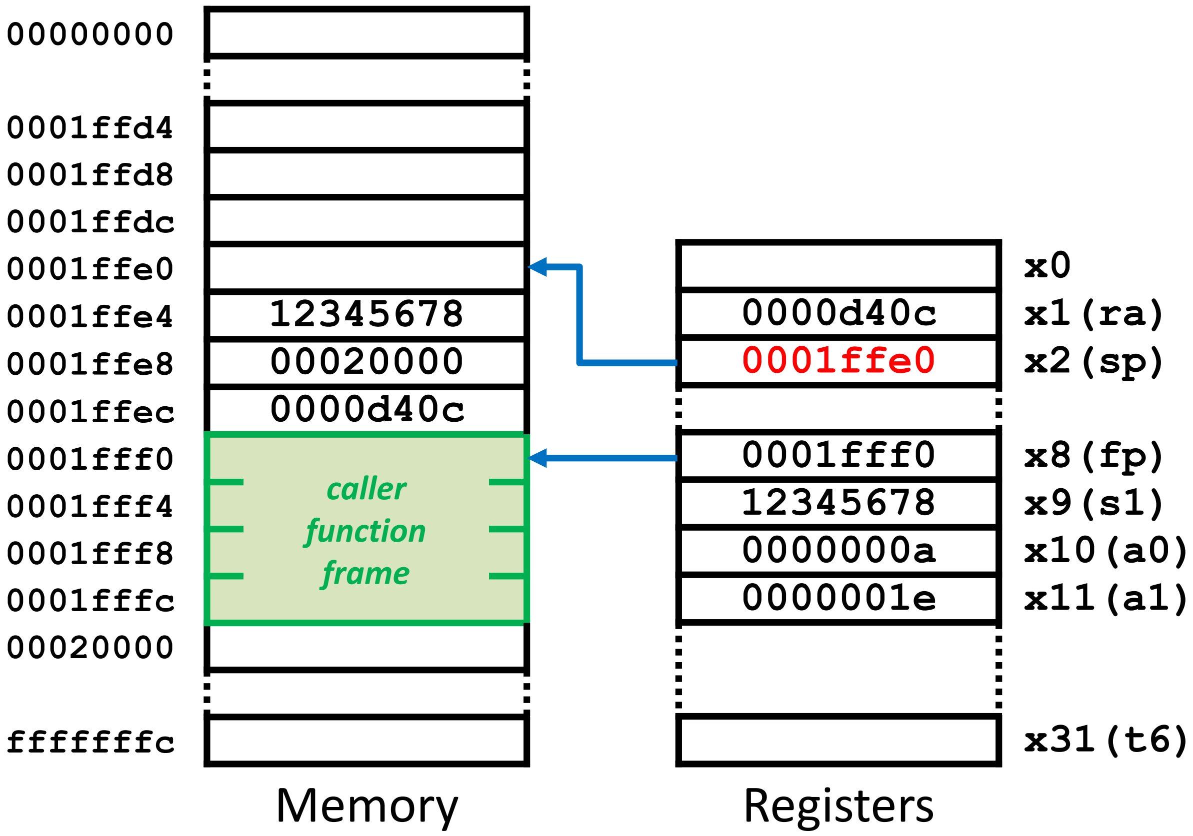
Registers

Functions

Frame management (iii)

ASM

```
y:  .space 4
    ...
    li    a0, 10
    li    a1, 30
    call  foo
0000d40c la    t6, y
    sw    a0, 0(t6)
    ...
foo:
    add   sp, sp, -12
    sw    ra  8(sp)
    sw    fp, 4(sp)
    sw    s1, 0(sp)
    add   fp, sp, 12
    add   sp, sp, -4
    li    t6, 0xff
    sw    t6, -16(fp)
    ...
    mv    a0, ...
    add   sp, sp, 4
    lw    ra  8(sp)
    lw    fp, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 12
    ret
```



00000000	
0001ffd4	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	12345678
0001ffe8	00020000
0001ffec	0000d40c
0001fff0	*caller function frame*
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001ffe0	x2(sp)
0001fff0	x8(fp)
12345678	x9(s1)
0000000a	x10(a0)
0000001e	x11(a1)
	x31(t6)

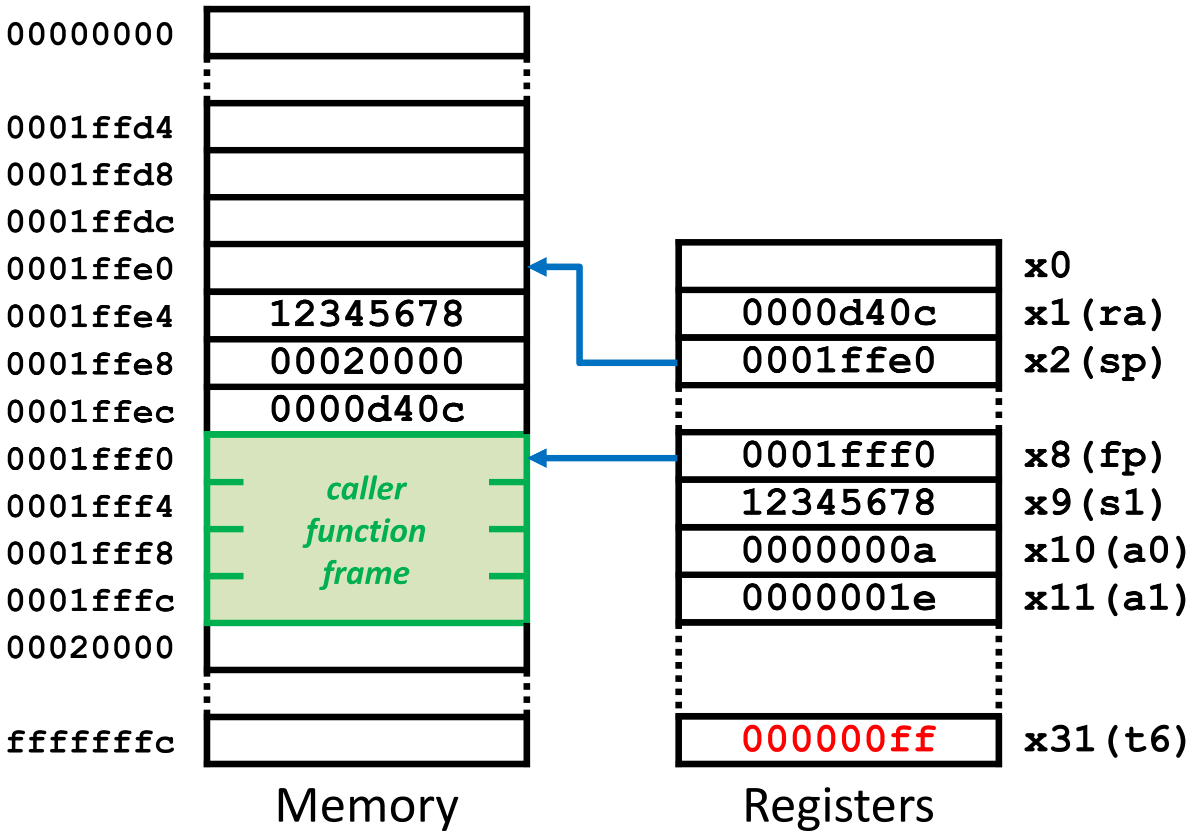
Registers

Functions

Frame management (iii)

ASM

```
y:  .space 4
    ...
    li    a0, 10
    li    a1, 30
    call  foo
0000d40c  la    t6, y
    sw    a0, 0(t6)
    ...
foo:
    add   sp, sp, -12
    sw    ra  8(sp)
    sw    fp, 4(sp)
    sw    s1, 0(sp)
    add   fp, sp, 12
    add   sp, sp, -4
    li    t6, 0xff
    sw    t6, -16(fp)
    ...
    mv    a0, ...
    add   sp, sp, 4
    lw    ra  8(sp)
    lw    fp, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 12
    ret
```



00000000	
0001ffd4	
0001ffd8	
0001ffdc	
0001ffe0	
0001ffe4	12345678
0001ffe8	00020000
0001ffec	0000d40c
0001fff0	caller function frame
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001ffe0	x2(sp)
0001fff0	x8(fp)
12345678	x9(s1)
0000000a	x10(a0)
0000001e	x11(a1)
000000ff	x31(t6)

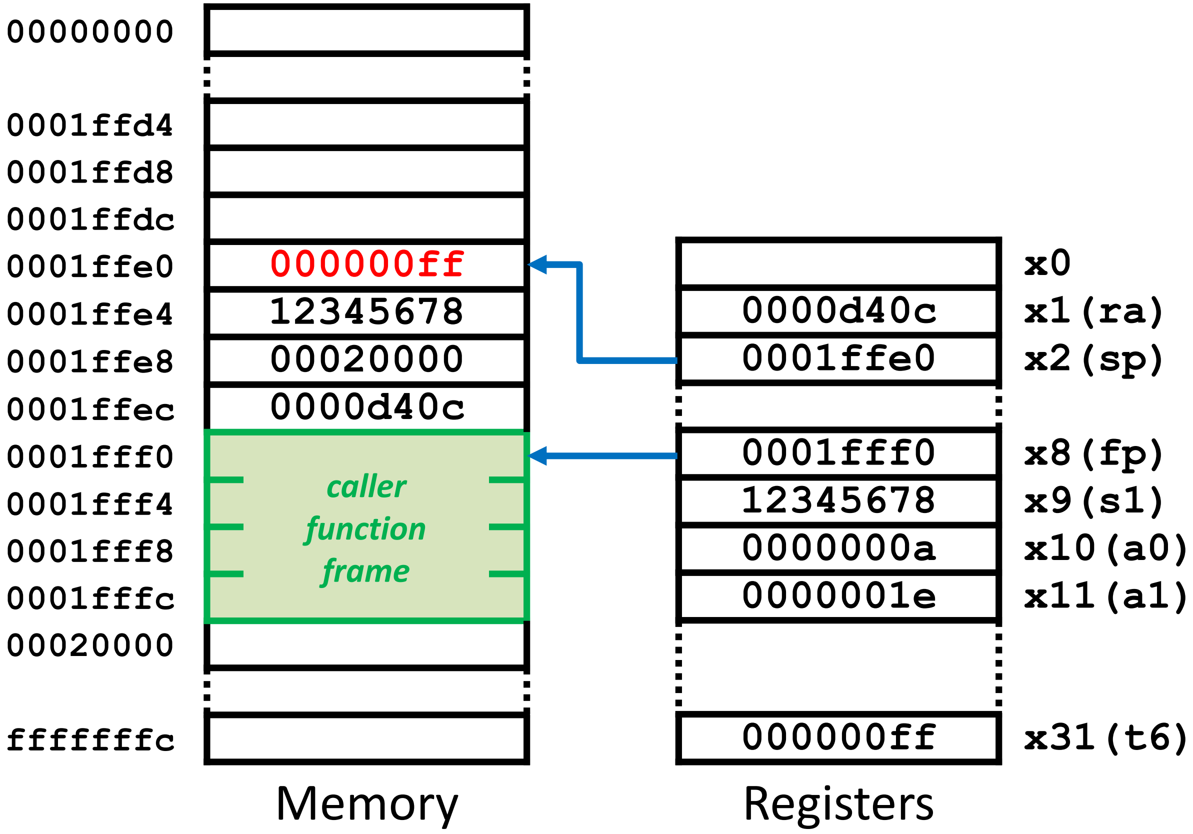
Registers

Functions

Frame management (iii)

ASM

```
y:  .space 4
         ...
         li    a0, 10
         li    a1, 30
         call  foo
0000d40c la    t6, y
         sw    a0, 0(t6)
         ...
foo:
         add   sp, sp, -12
         sw    ra  8(sp)
         sw    fp, 4(sp)
         sw    s1, 0(sp)
         add   fp, sp, 12
         add   sp, sp, -4
         li    t6, 0xff
  ⇨      sw    t6, -16(fp)
         ...
         mv    a0, ...
         add   sp, sp, 4
         lw    ra  8(sp)
         lw    fp, 4(sp)
         lw    s1, 0(sp)
         add   sp, sp, 12
         ret
```



00000000	
0001ffd4	
0001ffd8	
0001ffdc	
0001ffe0	000000ff
0001ffe4	12345678
0001ffe8	00020000
0001ffec	0000d40c
0001fff0	caller function frame
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001ffe0	x2(sp)
0001fff0	x8(fp)
12345678	x9(s1)
0000000a	x10(a0)
0000001e	x11(a1)
000000ff	x31(t6)

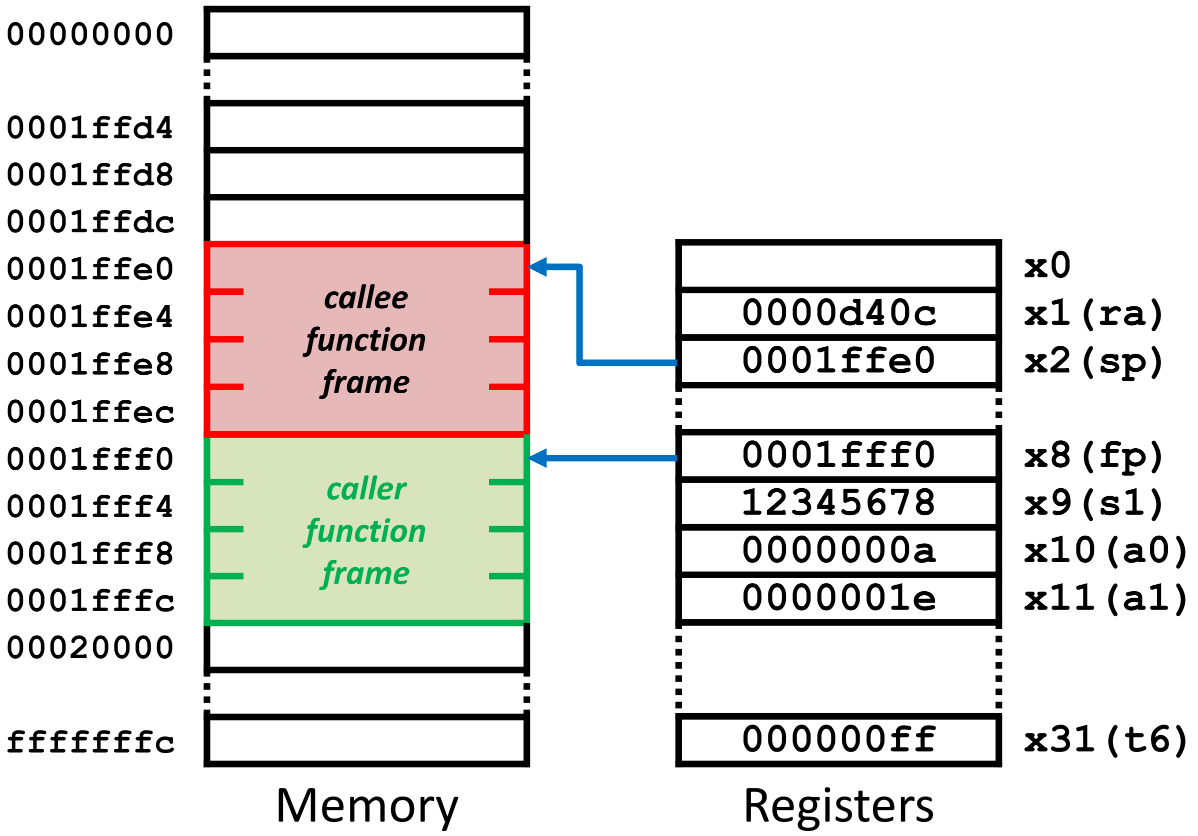
Registers

Functions

Frame management (iii)

ASM

```
         y:  .space 4
               ...
             li    a0, 10
             li    a1, 30
             call  foo
0000d40c     la    t6, y
             sw    a0, 0(t6)
               ...
         foo:
             add   sp, sp, -12
             sw    ra  8(sp)
             sw    fp, 4(sp)
             sw    s1, 0(sp)
             add   fp, sp, 12
             add   sp, sp, -4
             li    t6, 0xff
      ⇨      sw    t6, -16(fp)
               ...
             mv    a0, ...
             add   sp, sp, 4
             lw    ra  8(sp)
             lw    fp, 4(sp)
             lw    s1, 0(sp)
             add   sp, sp, 12
             ret
```

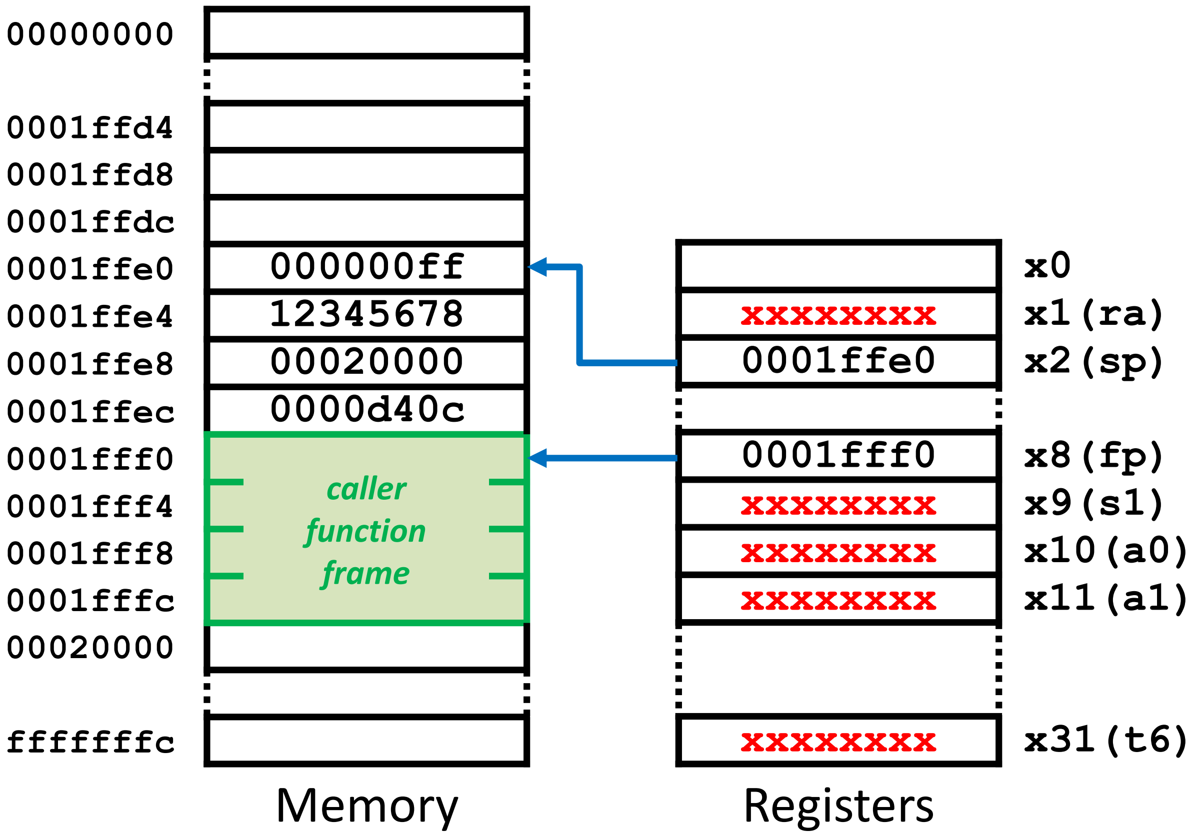


Functions

Frame management (iii)

ASM

```
y:  .space 4
         ...
         li    a0, 10
         li    a1, 30
         call  foo
0000d40c la    t6, y
         sw    a0, 0(t6)
         ...
foo:
         add   sp, sp, -12
         sw    ra  8(sp)
         sw    fp, 4(sp)
         sw    s1, 0(sp)
         add   fp, sp, 12
         add   sp, sp, -4
         li    t6, 0xff
         sw    t6, -16(fp)
         ...
         mv    a0, ...
         add   sp, sp, 4
         lw    ra  8(sp)
         lw    fp, 4(sp)
         lw    s1, 0(sp)
         add   sp, sp, 12
         ret
```

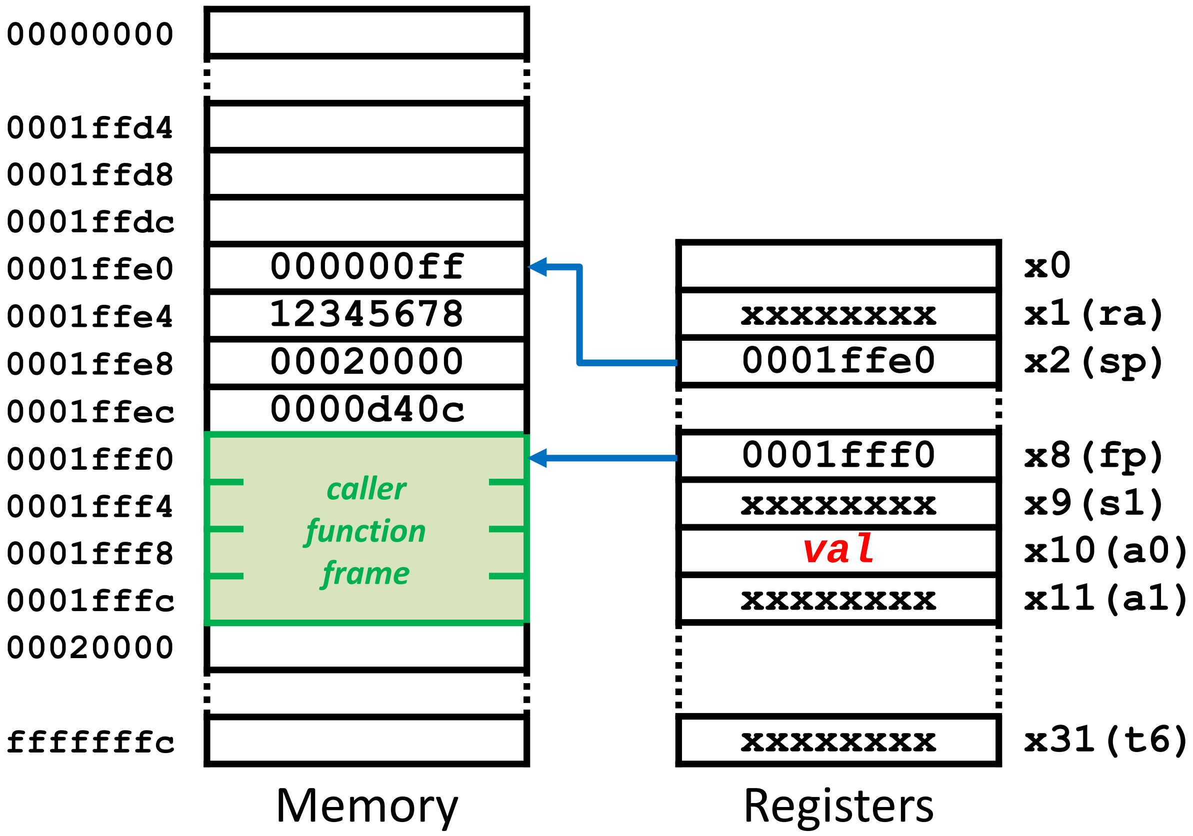


Functions

Frame management (iii)

ASM

```
y:  .space 4
    ...
    li    a0, 10
    li    a1, 30
    call  foo
0000d40c la    t6, y
    sw    a0, 0(t6)
    ...
foo:
    add   sp, sp, -12
    sw    ra  8(sp)
    sw    fp, 4(sp)
    sw    s1, 0(sp)
    add   fp, sp, 12
    add   sp, sp, -4
    li    t6, 0xff
    sw    t6, -16(fp)
    ...
    mv    a0, ...
    add   sp, sp, 4
    lw    ra  8(sp)
    lw    fp, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 12
    ret
```



00000000	
0001ffd4	
0001ffd8	
0001ffdc	
0001ffe0	000000ff
0001ffe4	12345678
0001ffe8	00020000
0001ffec	0000d40c
0001fff0	*caller function frame*
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
xxxxxxxx	x1(ra)
0001ffe0	x2(sp)
0001fff0	x8(fp)
xxxxxxxx	x9(s1)
val	x10(a0)
xxxxxxxx	x11(a1)
xxxxxxxx	x31(t6)

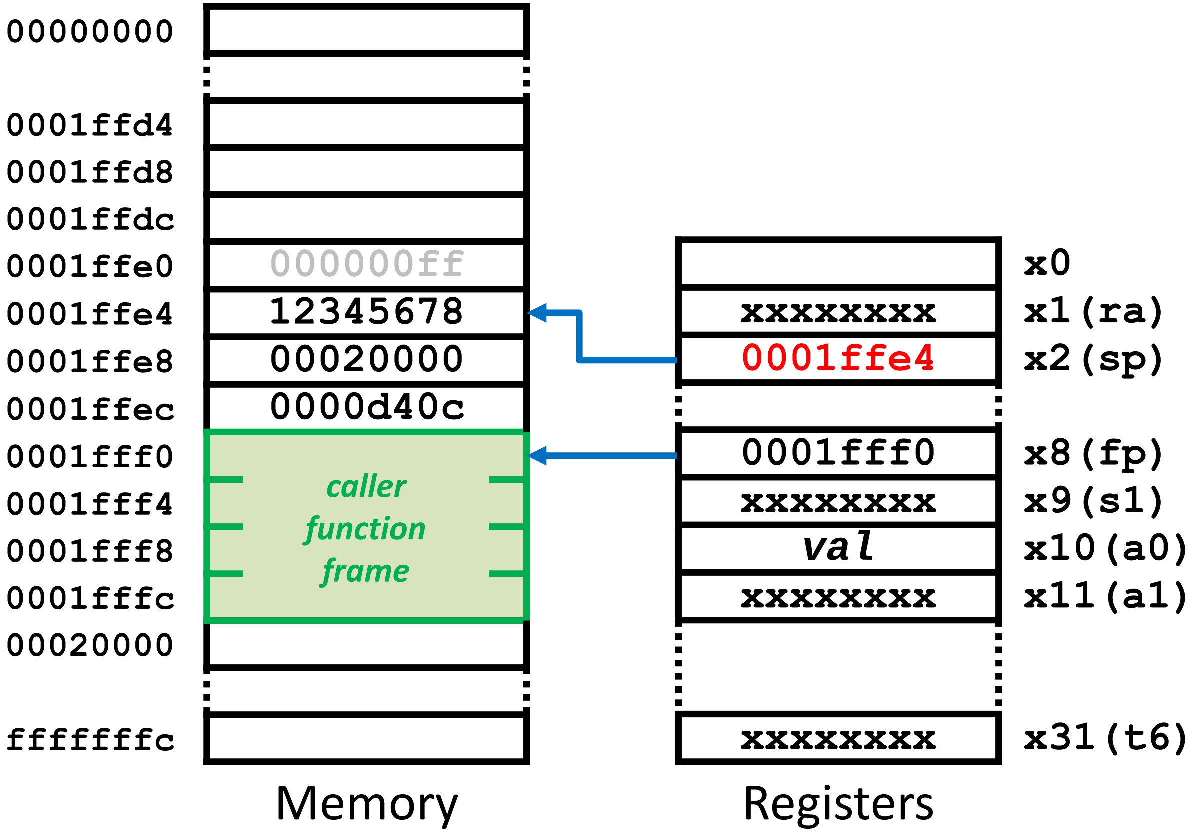
Registers

Functions

Frame management (iii)

ASM

```
y:  .space 4
         ...
         li    a0, 10
         li    a1, 30
         call  foo
0000d40c la    t6, y
         sw    a0, 0(t6)
         ...
foo:
         add   sp, sp, -12
         sw    ra  8(sp)
         sw    fp, 4(sp)
         sw    s1, 0(sp)
         add   fp, sp, 12
         add   sp, sp, -4
         li    t6, 0xff
         sw    t6, -16(fp)
         ...
         mv    a0, ...
         add   sp, sp, 4
         lw    ra  8(sp)
         lw    fp, 4(sp)
         lw    s1, 0(sp)
         add   sp, sp, 12
         ret
```

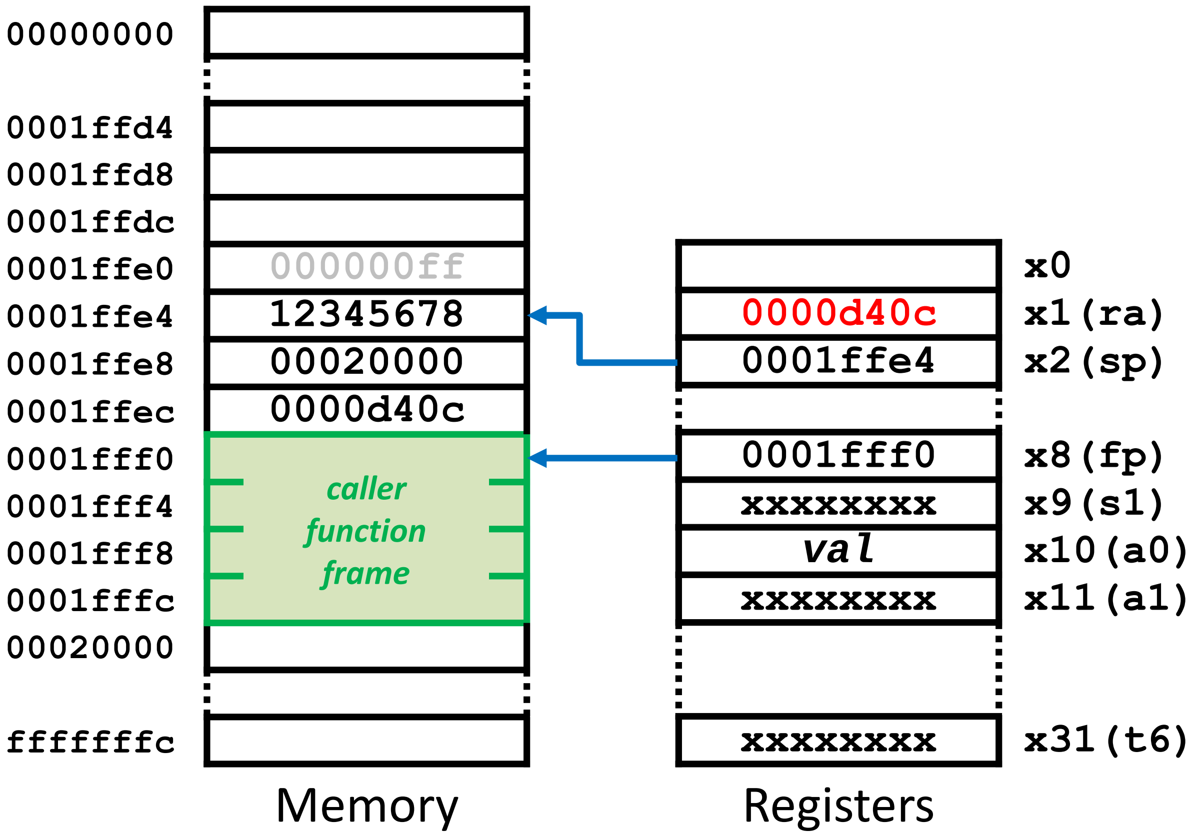


Functions

Frame management (iii)

ASM

```
         y:  .space 4
             ...
             li    a0, 10
             li    a1, 30
             call  foo
0000d40c     la    t6, y
             sw    a0, 0(t6)
             ...
         foo:
             add   sp, sp, -12
             sw    ra  8(sp)
             sw    fp, 4(sp)
             sw    s1, 0(sp)
             add   fp, sp, 12
             add   sp, sp, -4
             li    t6, 0xff
             sw    t6, -16(fp)
             ...
             mv    a0, ...
             add   sp, sp, 4
             lw    ra  8(sp)
             lw    fp, 4(sp)
             lw    s1, 0(sp)
             add   sp, sp, 12
             ret
```



00000000	
0001ffd4	
0001ffd8	
0001ffdc	
0001ffe0	000000ff
0001ffe4	12345678
0001ffe8	00020000
0001ffec	0000d40c
0001fff0	caller function frame
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001ffe4	x2(sp)
0001fff0	x8(fp)
xxxxxxxx	x9(s1)
val	x10(a0)
xxxxxxxx	x11(a1)
xxxxxxxx	x31(t6)

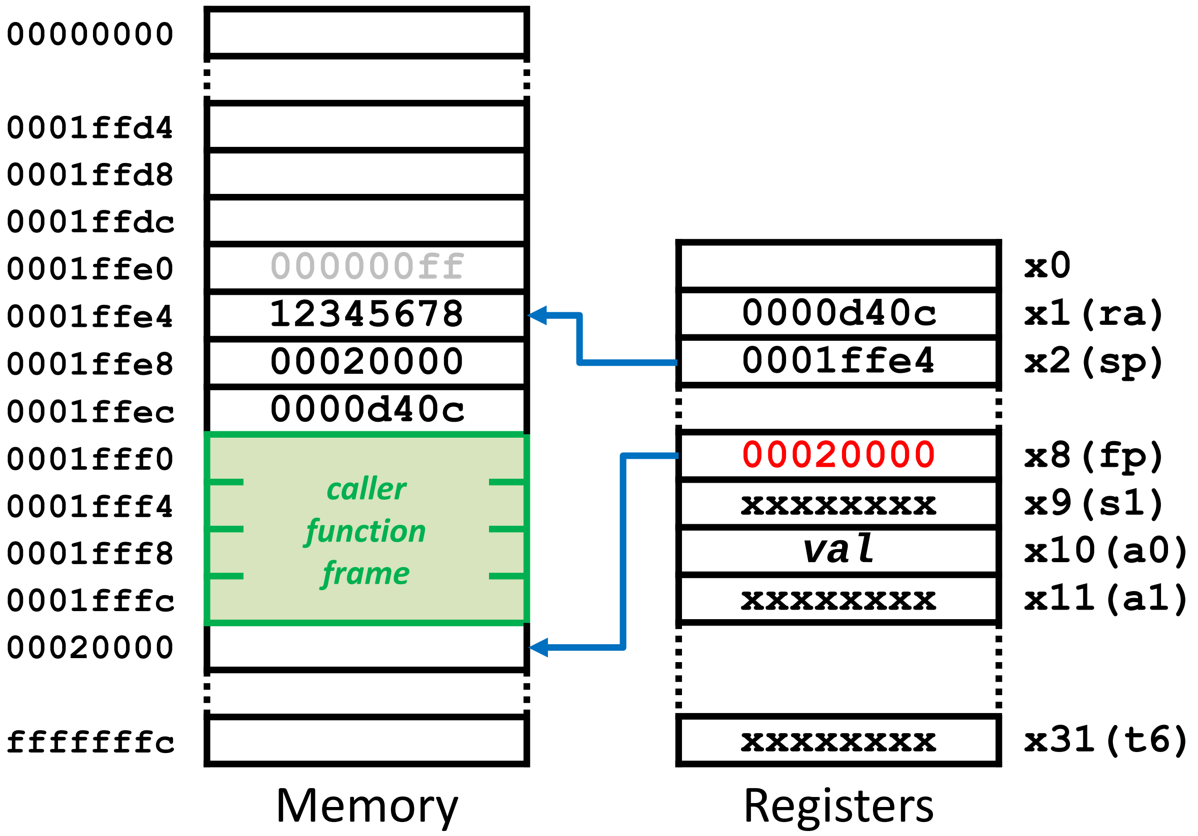
Registers

Functions

Frame management (iii)

ASM

```
y:  .space 4
    ...
    li    a0, 10
    li    a1, 30
    call  foo
0000d40c la    t6, y
    sw    a0, 0(t6)
    ...
foo:
    add   sp, sp, -12
    sw    ra  8(sp)
    sw    fp, 4(sp)
    sw    s1, 0(sp)
    add   fp, sp, 12
    add   sp, sp, -4
    li    t6, 0xff
    sw    t6, -16(fp)
    ...
    mv    a0, ...
    add   sp, sp, 4
    lw    ra  8(sp)
    lw    fp, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 12
    ret
```

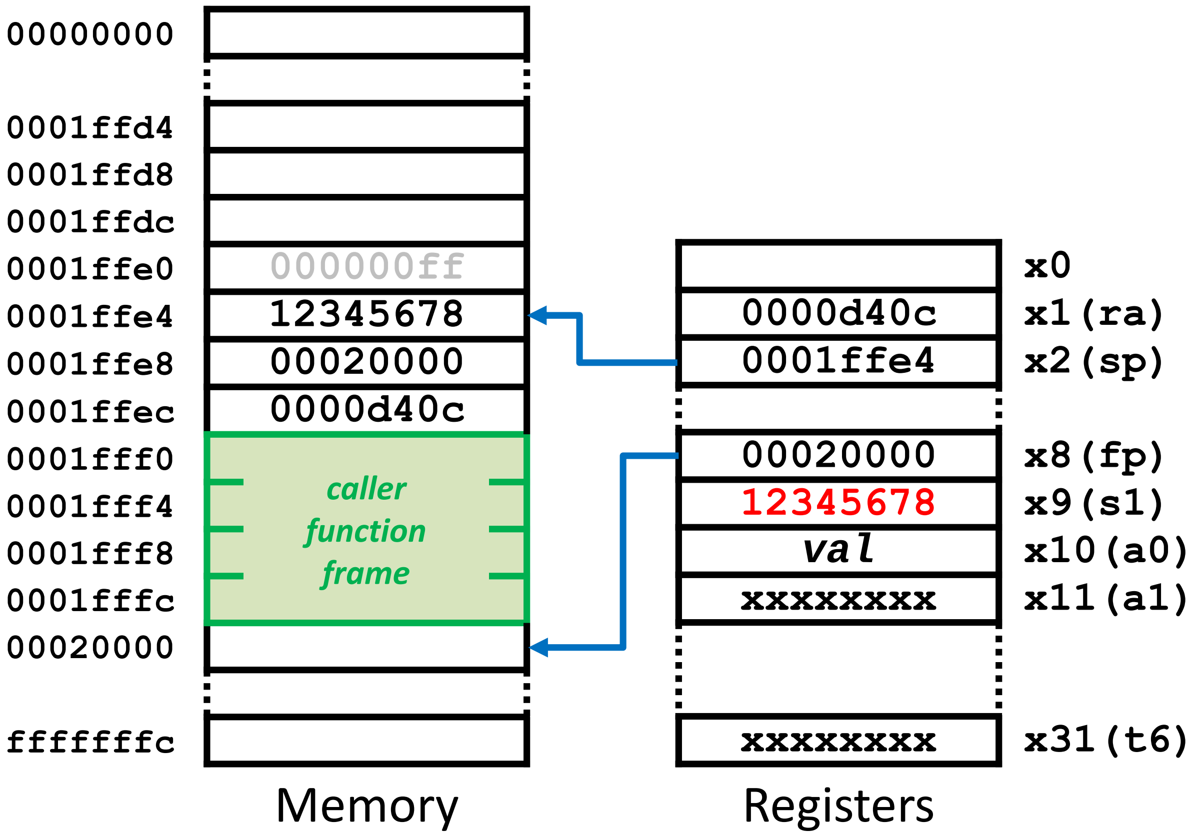


Functions

Frame management (iii)

ASM

```
         y:  .space 4
             ...
             li    a0, 10
             li    a1, 30
             call  foo
0000d40c     la    t6, y
             sw    a0, 0(t6)
             ...
         foo:
             add   sp, sp, -12
             sw    ra  8(sp)
             sw    fp, 4(sp)
             sw    s1, 0(sp)
             add   fp, sp, 12
             add   sp, sp, -4
             li    t6, 0xff
             sw    t6, -16(fp)
             ...
             mv    a0, ...
             add   sp, sp, 4
             lw    ra  8(sp)
             lw    fp, 4(sp)
  ⇒          lw    s1, 0(sp)
             add   sp, sp, 12
             ret
```

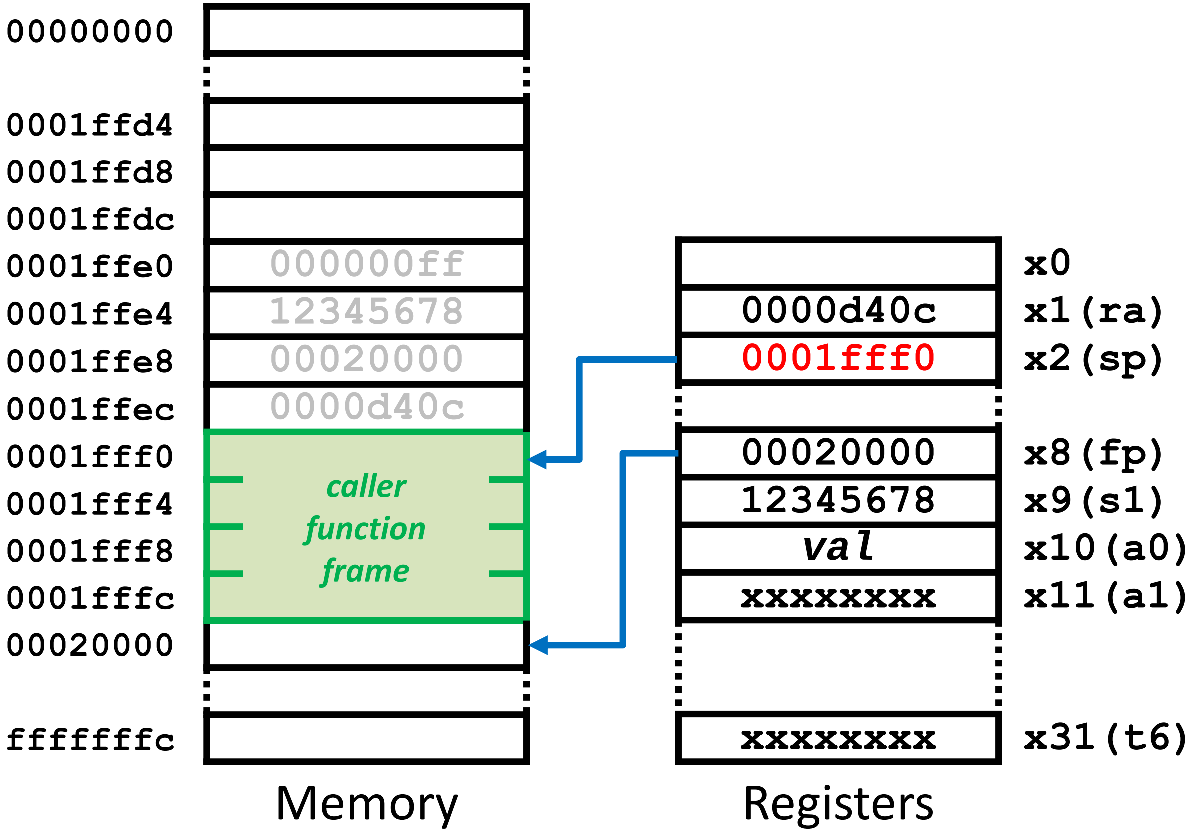


Functions

Frame management (iii)

ASM

```
y:  .space 4
    ...
    li    a0, 10
    li    a1, 30
    call  foo
0000d40c la    t6, y
    sw    a0, 0(t6)
    ...
foo:
    add   sp, sp, -12
    sw    ra  8(sp)
    sw    fp, 4(sp)
    sw    s1, 0(sp)
    add   fp, sp, 12
    add   sp, sp, -4
    li    t6, 0xff
    sw    t6, -16(fp)
    ...
    mv    a0, ...
    add   sp, sp, 4
    lw    ra  8(sp)
    lw    fp, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 12
    ret
```



00000000	
0001ffd4	
0001ffd8	
0001ffdc	
0001ffe0	000000ff
0001ffe4	12345678
0001ffe8	00020000
0001ffec	0000d40c
0001fff0	*caller function frame*
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001fff0	x2(sp)
00020000	x8(fp)
12345678	x9(s1)
val	x10(a0)
xxxxxxxx	x11(a1)
xxxxxxxx	x31(t6)

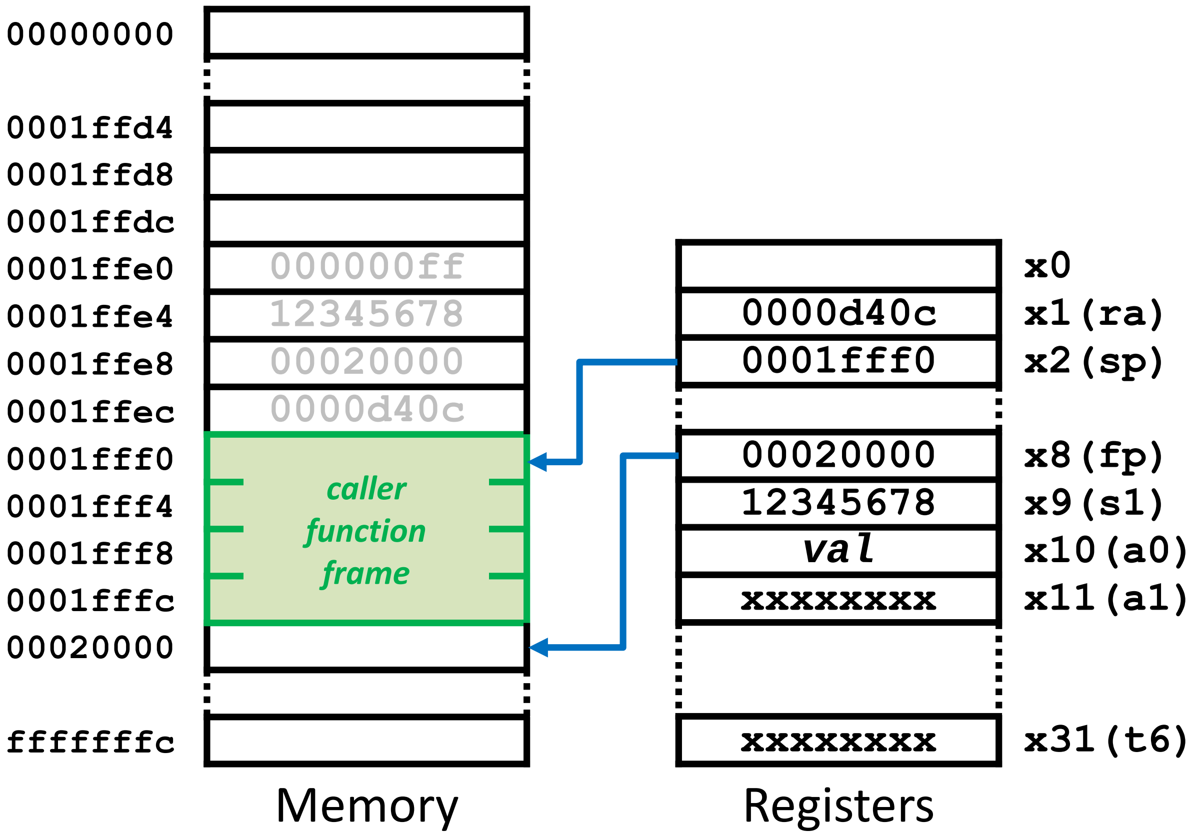
Registers

Functions

Frame management (iii)

ASM

```
y:  .space 4
    ...
    li    a0, 10
    li    a1, 30
    call  foo
0000d40c la    t6, y
    sw    a0, 0(t6)
    ...
foo:
    add   sp, sp, -12
    sw    ra  8(sp)
    sw    fp, 4(sp)
    sw    s1, 0(sp)
    add   fp, sp, 12
    add   sp, sp, -4
    li    t6, 0xff
    sw    t6, -16(fp)
    ...
    mv    a0, ...
    add   sp, sp, 4
    lw    ra  8(sp)
    lw    fp, 4(sp)
    lw    s1, 0(sp)
    add   sp, sp, 12
    ret
```



00000000	
0001ffd4	
0001ffd8	
0001ffdc	
0001ffe0	000000ff
0001ffe4	12345678
0001ffe8	00020000
0001ffec	0000d40c
0001fff0	caller function frame
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

Memory

	x0
0000d40c	x1(ra)
0001fff0	x2(sp)
00020000	x8(fp)
12345678	x9(s1)
val	x10(a0)
xxxxxxxx	x11(a1)
xxxxxxxx	x31(t6)

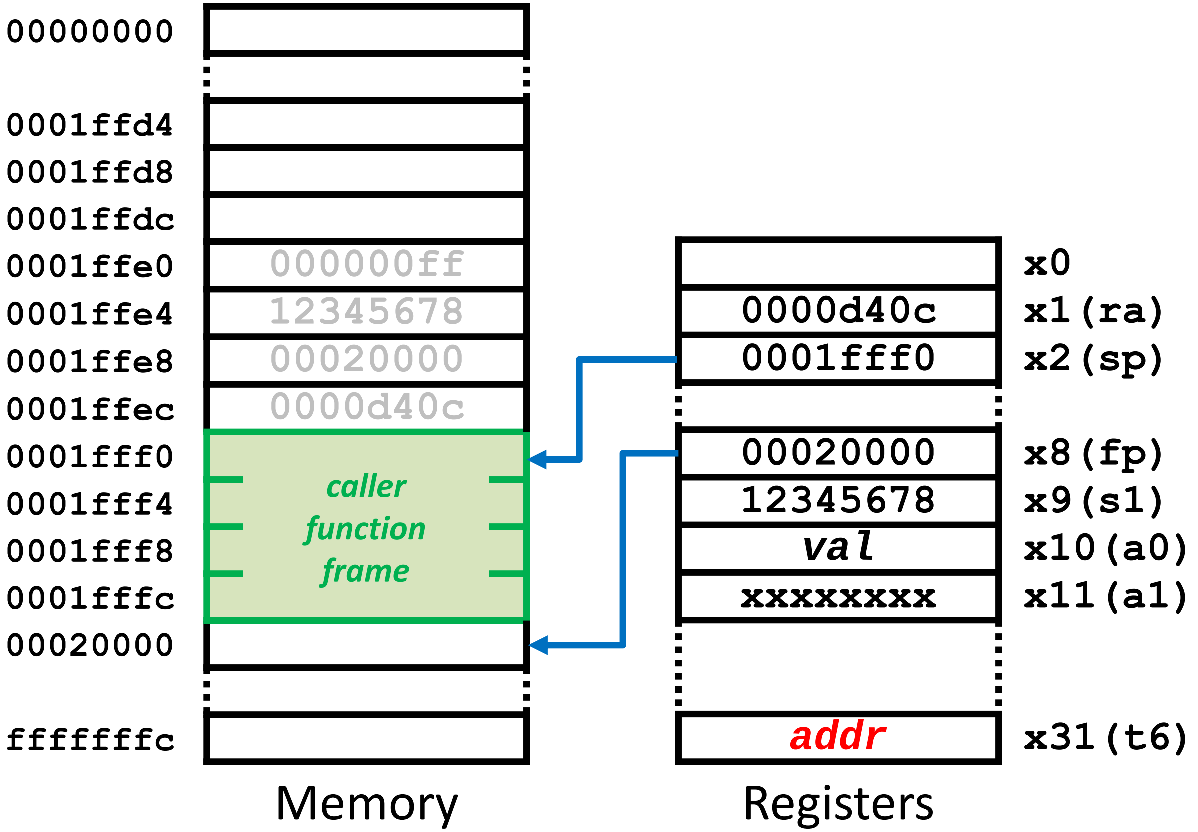
Registers

Functions

Frame management (iii)

ASM

```
y:  .space 4
   ...
   li    a0, 10
   li    a1, 30
   call  foo
   la    t6, y
   sw    a0, 0(t6)
   ...
foo:
   add   sp, sp, -12
   sw    ra  8(sp)
   sw    fp, 4(sp)
   sw    s1, 0(sp)
   add   fp, sp, 12
   add   sp, sp, -4
   li    t6, 0xff
   sw    t6, -16(fp)
   ...
   mv    a0, ...
   add   sp, sp, 4
   lw    ra  8(sp)
   lw    fp, 4(sp)
   lw    s1, 0(sp)
   add   sp, sp, 12
   ret
```



00000000	
0001ffd4	
0001ffd8	
0001ffdc	
0001ffe0	000000ff
0001ffe4	12345678
0001ffe8	00020000
0001ffec	0000d40c
0001fff0	caller function frame
0001fff4	
0001fff8	
0001fffc	
00020000	
fffffffc	

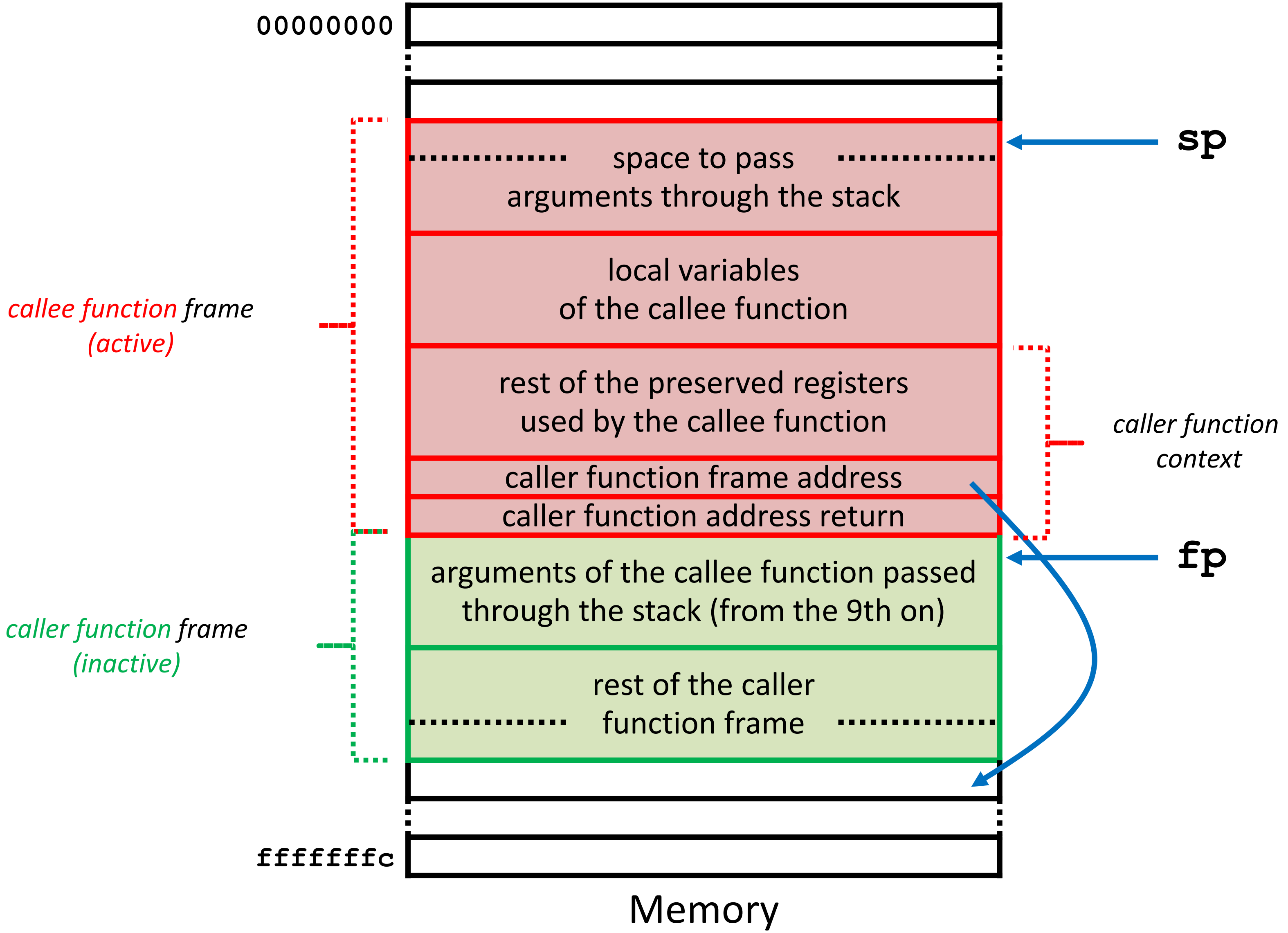
Memory

	x0
0000d40c	x1(ra)
0001fff0	x2(sp)
00020000	x8(fp)
12345678	x9(s1)
val	x10(a0)
xxxxxxxx	x11(a1)
addr	x31(t6)

Registers

Functions

Frame management (iv)



Functions

Frame management (v)

- According to the call procedure:
  - The limits of an active frame are marked by **`fp`** and **`sp`**.
  - The arguments from the 9th on (passed through the stack by the caller function) always have positive offsets respect to **`fp`**.
  - The context of the caller function and the local variables of the callee function always have negative offsets respect to **`fp`**.

- However, this procedure is simpler depending on the callee function:
  - If it is a leaf function, **`ra`** is not stored in its frame.
  - If it does not use preserved registers, these are not stored in its frame.
  - If it has fewer than 8 parameters, there are no arguments on the caller function frame.
  - If all its local variables are stored in registers, there is no need to save and use the **`fp`**.

Functions

Example (i)

- Bubble sort algorithm to sort the elements of an array:

C/C++

```
void sort( int v[], int n )
{
   int i, j;

   for( i=0; i<n; i++ )
      for( j=i-1; j>=0 && v[j]>v[j+1]; j-- )
         swap( v, j );
}

void swap( int v[], int k )
{
   int temp;

   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Functions

Example (ii)

- The **swap** function is a leaf function and it only uses temporary registers.
  - Context does not have to be saved.
  - It will store its local variable in a register, and therefore **fp** is not used.
- It receives 2 arguments and it does not return any result.
  - It receives the base address of the array to be sorted in **a0**.
  - It receives the index of the element to be swapped with the following one in **a1**.
  - It does not return anything in **a0**.

C/C++

```c
void swap( int v[],
           int k    )
{
  int temp;

  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

ASM

```
swap:
  slli t0, a1, 2
  add  t0, a0, t0
  lw   t1, 0(t0)
  lw   t2, 4(t0)
  sw   t2, 0(t0)
  sw   t1, 4(t0)
  ret
```

Calculates the `k×4` *offset*

Adds base and offset

Loads `v[k]` *into* `t1`

Loads `v[k+1]` *into* `t2`

Stores `t2` *into* `v[k]`

Stores `t1` *into* `v[k+1]`

Functions

Example (iii)

- The `sort` function is a non-leaf function that will use preserved registers.
  - The return address and the context will have to be saved.
  - It will store its local variables in registers, therefore it will not use `fp`.
- It receives 2 arguments and it does not return any result.
  - It receives the base address of the array to be sorted in `a0`.
  - It receives the number of elements of the array in `a1`.
  - It does not return anything in `a0`.

C/C++

```
void sort( int v[], int n )
{
  int i, j;

  for( i=0; i<n; i++ )
    for( j=i-1; j>=0 && v[j]>v[j+1]; j-- )
      swap( v, j );
}
```

Functions

Example (iv)

C/C++

```
void sort( int v[], int n )
{
  int i, j;

  for( i=0; i<n; i++ )
    for( j=i-1;
         j>=0 && v[j]>v[j+1];
         j-- )
      swap( v, j );
}
```

v[] (a0) → s1
n (a1) → s2
i → s3
j → s4

ASM

```
sort:
    ...
    mv    s1, a0
    mv    s2, a1




    ...
    ret
```

PROLOGUE

The arguments are copied

EPILOGUE

Functions

Example (iv)

C/C++

```
void sort( int v[], int n )
{
  int i, j;

  for( i=0; i<n; i++ )
    for( j=i-1;
         j>=0 && v[j]>v[j+1];
         j-- )
      swap( v, j );
}
```

v[] (a0) → s1
n (a1) → s2
i → s3
j → s4

ASM

```
sort:
   ...
   mv    s1, a0
   mv    s2, a1
   mv    s3, zero
fori:
   bge   s3, s2, efori





efori:
   ...
   ret
```

PROLOGUE

The arguments are copied

`i = 0`

`i ≥ n ?`

EPILOGUE

Functions

Example (iv)

C/C++

```
void sort( int v[], int n )
{
  int i, j;

  for( i=0; i<n; i++ )
    for( j=i-1;
         j>=0 && v[j]>v[j+1];
         j-- )
      swap( v, j );
}
```

v[] (a0) → s1
n (a1) → s2
i → s3
j → s4

ASM

```
sort:
    ...
    mv    s1, a0
    mv    s2, a1
    mv    s3, zero
fori:
    bge   s3, s2, efori














    add   s3, s3, 1
    j     fori
efori:
    ...
    ret
```

PROLOGUE

The arguments are copied

`i = 0`

`i ≥ n` ?

Increments `i`

EPILOGUE

Functions

Example (iv)

C/C++

```
void sort( int v[], int n )
{
  int i, j;

  for( i=0; i<n; i++ )
    for( j=i-1;
         j>=0 && v[j]>v[j+1];
         j-- )
      swap( v, j );
}
```

v[] (a0) → s1
n (a1) → s2
i → s3
j → s4

ASM

```
sort:
   ...
   mv    s1, a0
   mv    s2, a1
   mv    s3, zero
fori:
   bge   s3, s2, efori
   add   s4, s3, -1
forj:
   blt   s4, zero, eforj









eforj:
   add   s3, s3, 1
   j     fori
efori:
   ...
   ret
```

PROLOGUE

The arguments are copied

`i = 0`

`i ≥ n` ?

`j = i-1`

`j < 0` ?

Increments `i`

EPILOGUE

Functions

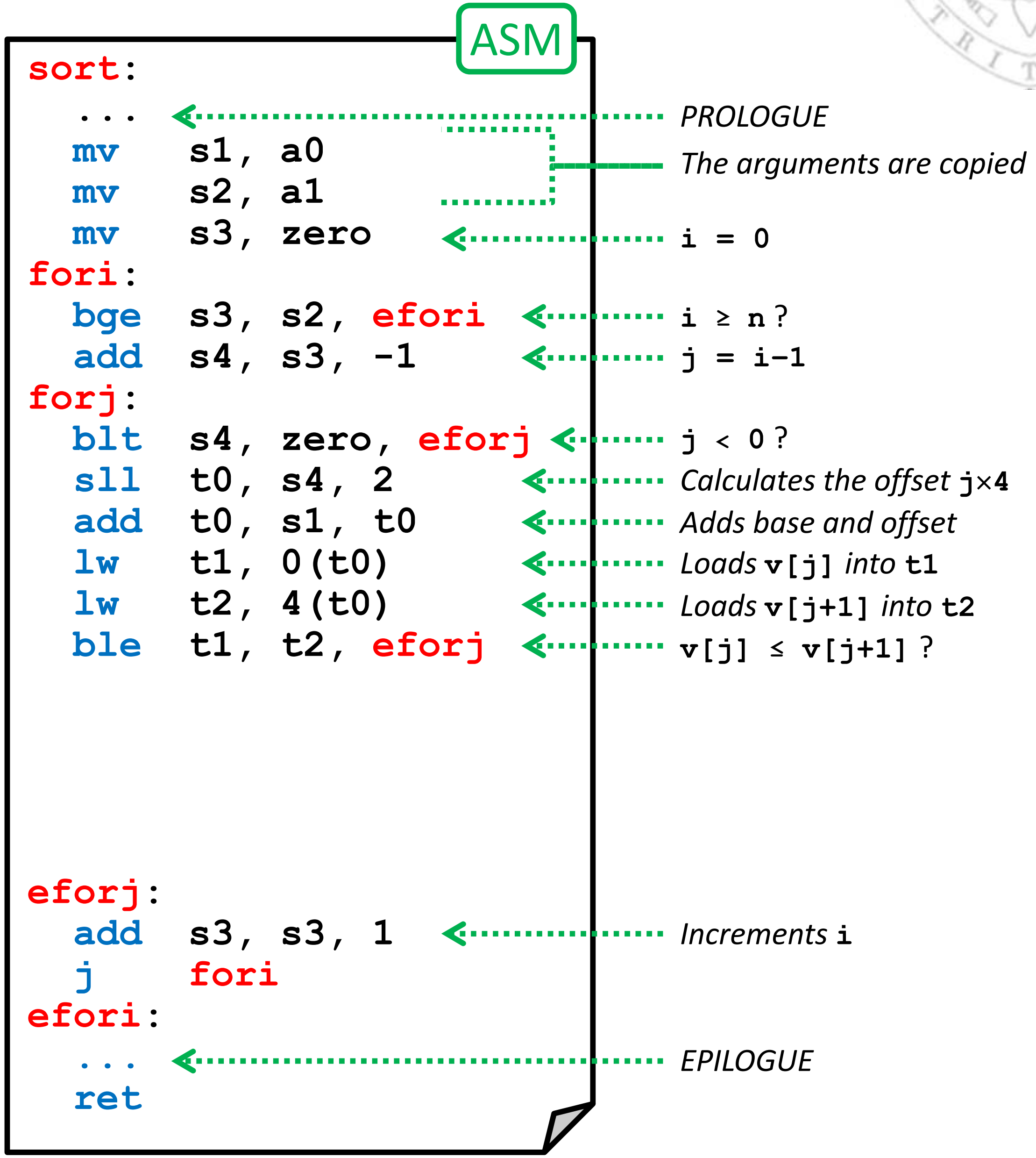
Example (iv)

C/C++

```
void sort( int v[], int n )
{
  int i, j;

  for( i=0; i<n; i++ )
    for( j=i-1;
         j>=0 && v[j]>v[j+1];
         j-- )
      swap( v, j );
}
```

v[] (a0) → s1
n (a1) → s2
i → s3
j → s4



ASM

```
sort:
    ...
    mv   s1, a0
    mv   s2, a1
    mv   s3, zero
fori:
    bge  s3, s2, efori
    add  s4, s3, -1
forj:
    blt  s4, zero, eforj
    sll  t0, s4, 2
    add  t0, s1, t0
    lw   t1, 0(t0)
    lw   t2, 4(t0)
    ble  t1, t2, eforj





eforj:
    add  s3, s3, 1
    j    fori
efori:
    ...
    ret
```

PROLOGUE

The arguments are copied

`i = 0`

`i ≥ n` ?

`j = i-1`

`j < 0` ?

Calculates the offset `j×4`

Adds base and offset

Loads `v[j]` *into* `t1`

Loads `v[j+1]` *into* `t2`

`v[j] ≤ v[j+1]` ?

Increments `i`

EPILOGUE

Functions

Example (iv)

C/C++

```
void sort( int v[], int n )
{
  int i, j;

  for( i=0; i<n; i++ )
    for( j=i-1;
         j>=0 && v[j]>v[j+1];
         j-- )
      swap( v, j );
}
```

v[] (a0) → s1
n (a1) → s2
i → s3
j → s4

ASM

```
sort:
   ...                       ← PROLOGUE
   mv    s1, a0              ┐
   mv    s2, a1              ┘ The arguments are copied
   mv    s3, zero            ← i = 0
fori:
   bge   s3, s2, efori       ← i ≥ n ?
   add   s4, s3, -1          ← j = i-1
forj:
   blt   s4, zero, eforj     ← j < 0 ?
   sll   t0, s4, 2           ← Calculates the offset j×4
   add   t0, s1, t0          ← Adds base and offset
   lw    t1, 0(t0)           ← Loads v[j] into t1
   lw    t2, 4(t0)           ← Loads v[j+1] into t2
   ble   t1, t2, eforj       ← v[j] ≤ v[j+1] ?



   add   s4, s4, -1          ← Decrements j
   j     forj
eforj:
   add   s3, s3, 1           ← Increments i
   j     fori
efori:
   ...                       ← EPILOGUE
   ret
```

Functions

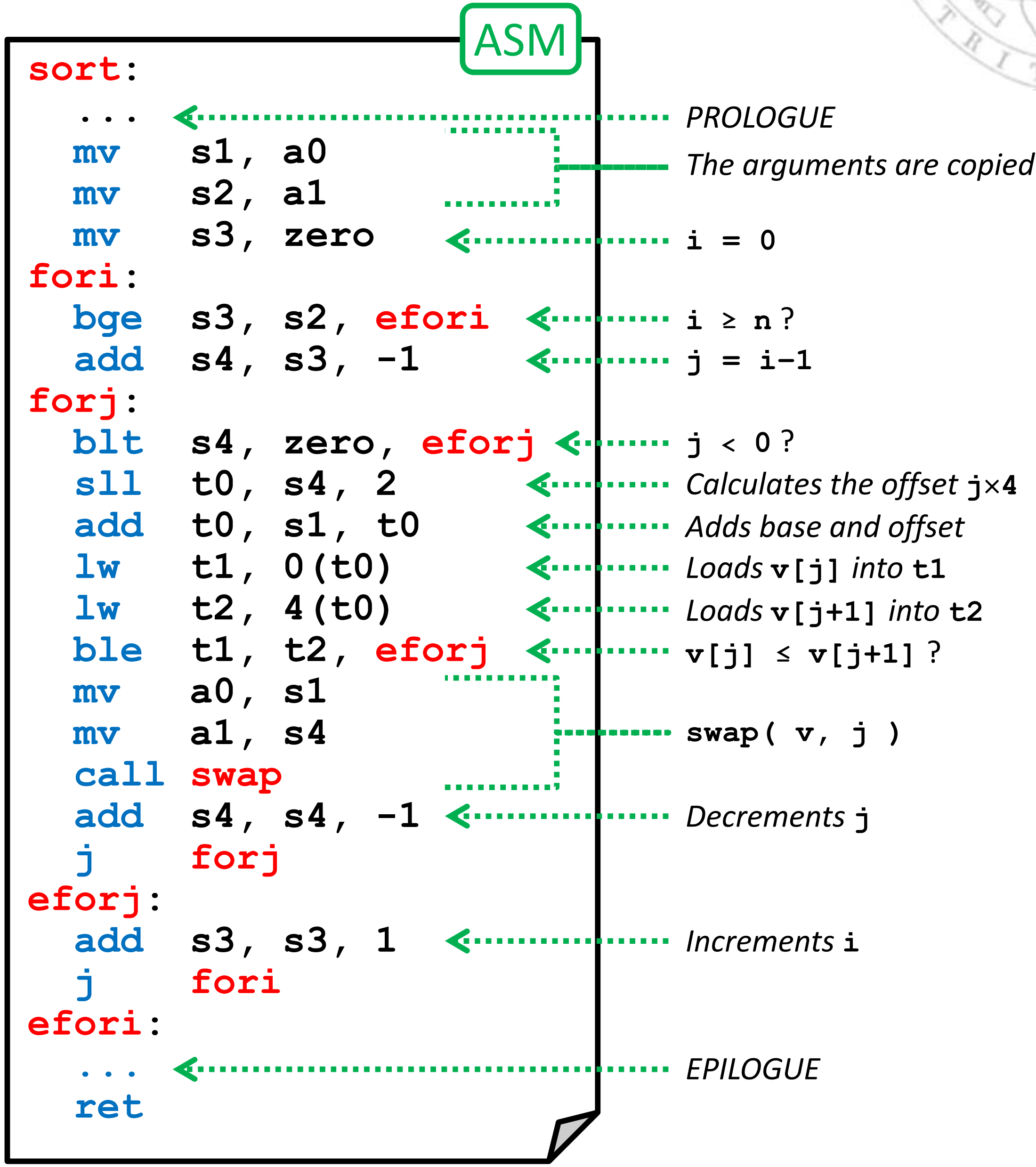
Example (iv)

C/C++

```c
void sort( int v[], int n )
{
  int i, j;

  for( i=0; i<n; i++ )
    for( j=i-1;
         j>=0 && v[j]>v[j+1];
         j-- )
      swap( v, j );
}
```

v[] (a0) → s1
n (a1) → s2
i → s3
j → s4



ASM

```
sort:
   ...
   mv    s1, a0
   mv    s2, a1
   mv    s3, zero
fori:
   bge   s3, s2, efori
   add   s4, s3, -1
forj:
   blt   s4, zero, eforj
   sll   t0, s4, 2
   add   t0, s1, t0
   lw    t1, 0(t0)
   lw    t2, 4(t0)
   ble   t1, t2, eforj
   mv    a0, s1
   mv    a1, s4
   call  swap
   add   s4, s4, -1
   j     forj
eforj:
   add   s3, s3, 1
   j     fori
efori:
   ...
   ret
```

PROLOGUE

The arguments are copied

`i = 0`

`i ≥ n` ?

`j = i-1`

`j < 0` ?

Calculates the offset `j×4`

Adds base and offset

Loads `v[j]` *into* `t1`

Loads `v[j+1]` *into* `t2`

`v[j] ≤ v[j+1]` ?

`swap( v, j )`

Decrements `j`

Increments `i`

EPILOGUE

Functions

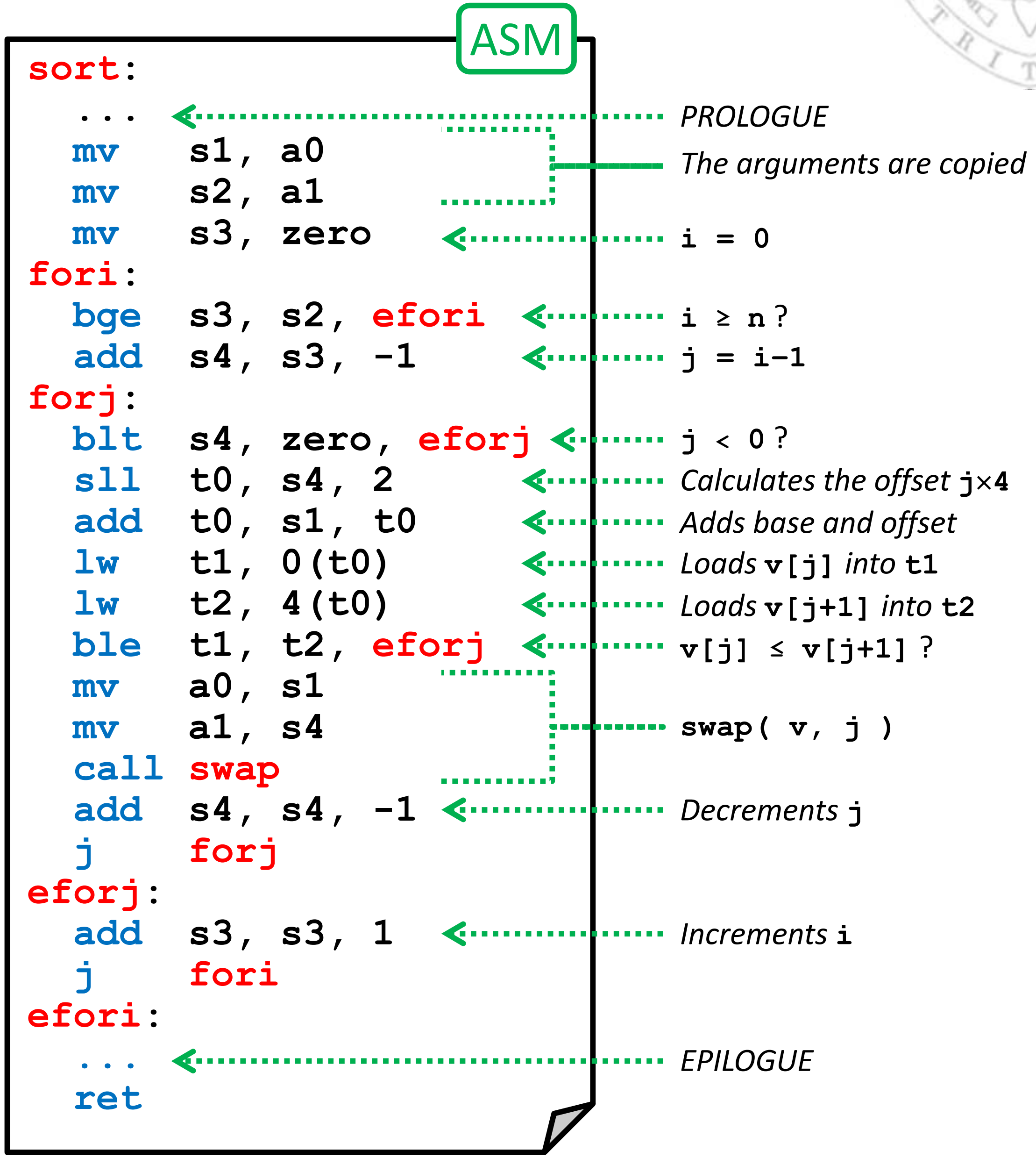
Example (iv)

C/C++

```
void sort( int v[], int n )
{
  int i, j;

  for( i=0; i<n; i++ )
    for( j=i-1;
         j>=0 && v[j]>v[j+1];
         j-- )
      swap( v, j );
}
```

v[] (a0) → s1
n (a1) → s2
i → s3
j → s4



ASM

```
sort:
   ...
   mv    s1, a0
   mv    s2, a1
   mv    s3, zero
fori:
   bge   s3, s2, efori
   add   s4, s3, -1
forj:
   blt   s4, zero, eforj
   sll   t0, s4, 2
   add   t0, s1, t0
   lw    t1, 0(t0)
   lw    t2, 4(t0)
   ble   t1, t2, eforj
   mv    a0, s1
   mv    a1, s4
   call  swap
   add   s4, s4, -1
   j     forj
eforj:
   add   s3, s3, 1
   j     fori
efori:
   ...
   ret
```

PROLOGUE

The arguments are copied

`i = 0`

`i ≥ n` ?

`j = i-1`

`j < 0` ?

Calculates the offset `j×4`

Adds base and offset

Loads `v[j]` *into* `t1`

Loads `v[j+1]` *into* `t2`

`v[j] ≤ v[j+1]` ?

`swap( v, j )`

Decrements `j`

Increments `i`

EPILOGUE

Functions

Example (v)

C/C++

```
void sort( int v[], int n )
{
  int i, j;

  for( i=0; i<n; i++ )
    for( j=i-1;
         j>=0 && v[j]>v[j+1];
         j-- )
      swap( v, j );
}
```

v[] (a0) → s1
n (a1) → s2
i → s3
j → s4

ASM

```
sort:
   add sp, sp, -4*5
   sw  ra, 4*4(sp)
   sw  s1, 3*4(sp)
   sw  s2, 2*4(sp)
   sw  s3, 1*4(sp)
   sw  s4, 0*0(sp)
    ...
fori:
    ...
forj:
  ...
eforj:
  ...
efori:
   lw  ra, 4*4(sp)
   lw  s1, 3*4(sp)
   lw  s2, 2*4(sp)
   lw  s3, 1*4(sp)
   lw  s4, 0*0(sp)
   add sp, sp, 4*5
   ret
```

PROLOGUE:
Pushes context

Function body

EPILOGUE:
Pops context

Local vs. global variables

- Accessing a global variable with a label requires performing between 2 and 3 instructions.

ASM

```
a:  .word   5
...
la    t0, a
lw    s1, 0(t0)
...
```

ASM

```
a:  .word   5
...
lw    s1, a
...
```

```
a:  .word   5
...
auipc t0, ...
addi  t0, t0, ...
lw    s1, 0(t0)
...
```

```
a:  .word   5
...
auipc s1, ...
lw    s1, ...(s1)
...
```

- To make the access faster, immediate offsets relative to registers are commonly used.
  - Similar to how local variables are addressed.

Local vs. global variables

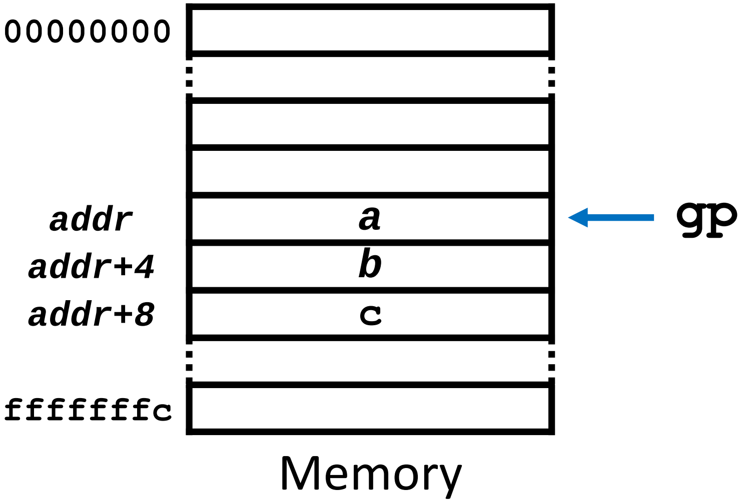
- For global variables, the **gp** register is used as the base.
  - At the beginning of the program, **gp** is initialized to point the region memory where the global variables are placed, and it never changes.
  - All global variables may be identified by a constant and unique offset related to **gp**.

ASM

```
a:  .word   76
b:  .word -39
c:  .space 4
...
la   t0, a
lw   s1, 0(t0)
la   t0, b
lw   s2, 0(t0)
add  s1, s1, s2
la   t0, c
sw   s1, 0(t0)
...
```

ASM

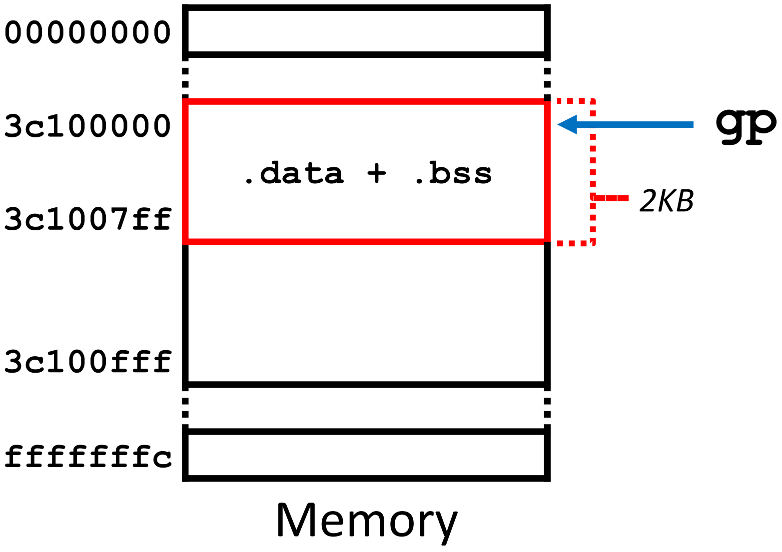
```
a:  .word   76
b:  .word -39
c:  .space 4
...
la   gp, a
...
lw   s1, 0(gp)
lw   s2, 4(gp)
add  s1, s1, s2
sw   s1, 8(gp)
...
```

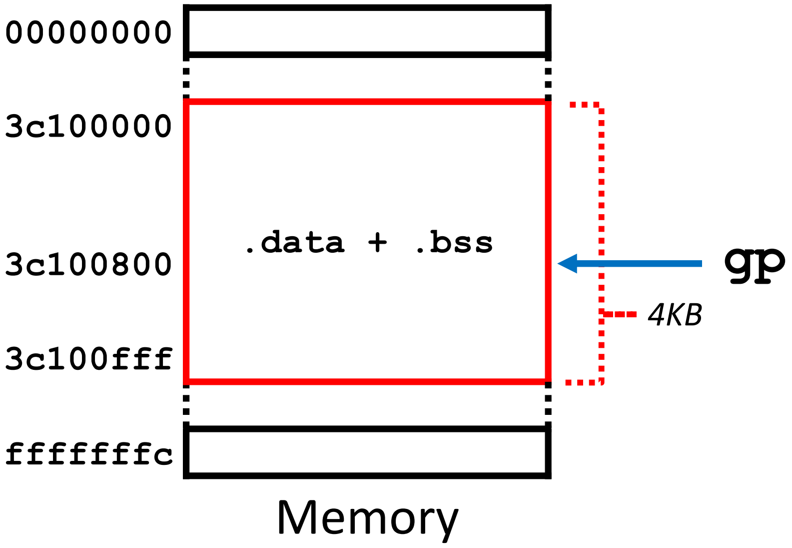


  - In multithread applications, the **tp** register is used in a similar way so that each thread has access to its space of local variables.

Local vs. global variables

- Offsets in the `lw`/`sw` instructions are represented in C2 with 12b.
  - If `gp` points to the beginning of the global data section, this region could have a size of at most 2KiB: `gp` + [0..2^{11}-1]
  - So, it is usual to initialize `gp` so that it points to the middle of that section. With this, a whole region of 4KiB could be addressed: `gp` ± [0..2^{11}-1]
    - i.e., adding 0x800 to the initial address of the section.





Local vs. global variables

- Global variables (static)
  - They are located in the main memory, in the `.data` or `.bss` sections
  - They have a fixed address during the whole execution of the program.
    - To address them, a label or an offset relative to `gp` are used.
  - They persist (are alive) during the whole execution of the program.
    - They are created (are available) when the program starts.
    - They are destroyed (their addresses are reused) when the program ends.

- Local variables (automatic)
  - They are located in the stack, in the activation frame of the function.
    - The stack is a memory region different from the code or global variable regions.
  - They have a different address in each function call.
    - To address them, relative offsets to `sp` or `fp` are used.
  - They persist (are alive) only during the execution of the function.
    - They are created after the function call (in the function prologue).
    - They are destroyed before returning (in the function epilogue).

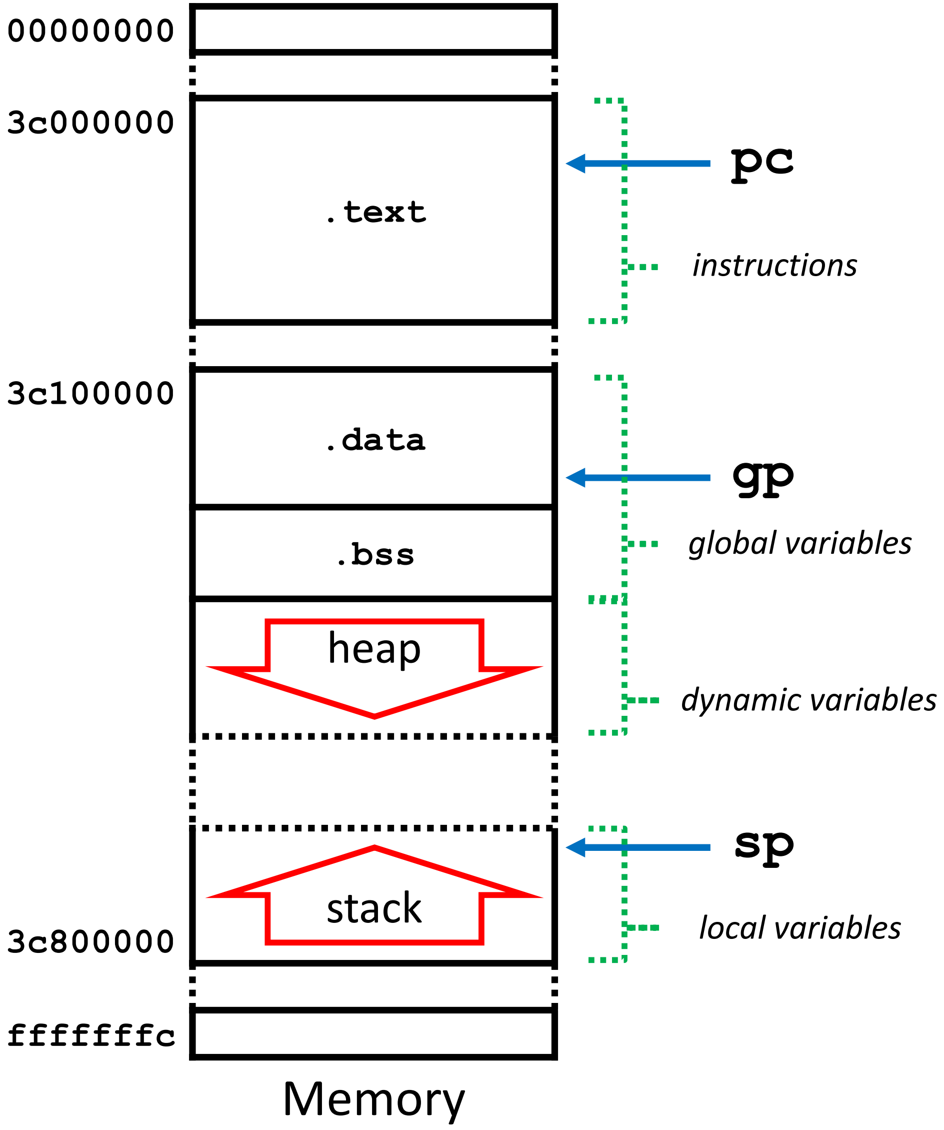
Local vs. global variables

Dynamic variables

- In C/C++, besides, there are dynamic variables
  - Which are created and destroyed explicitly by the programmer during the execution of a program, using **`malloc/free`** (C) or **`new/delete`** (C++).
- The heap is a memory region where the dynamic variables of a program are located.
  - It is usually placed in the memory side opposite to the stack, and it grows in the opposite direction.
- There are many alternatives to manage the dynamic memory of a computer. For example:
  - **`malloc/new`** returns the address of an adjacent region of free memory in the heap, with the requested size.
    - This region is accessed with offsets relative to a base register, which stores the address returned by **`malloc/new`**.
  - **`free/delete`** marks the region whose address is given as free, so that it can be reused.

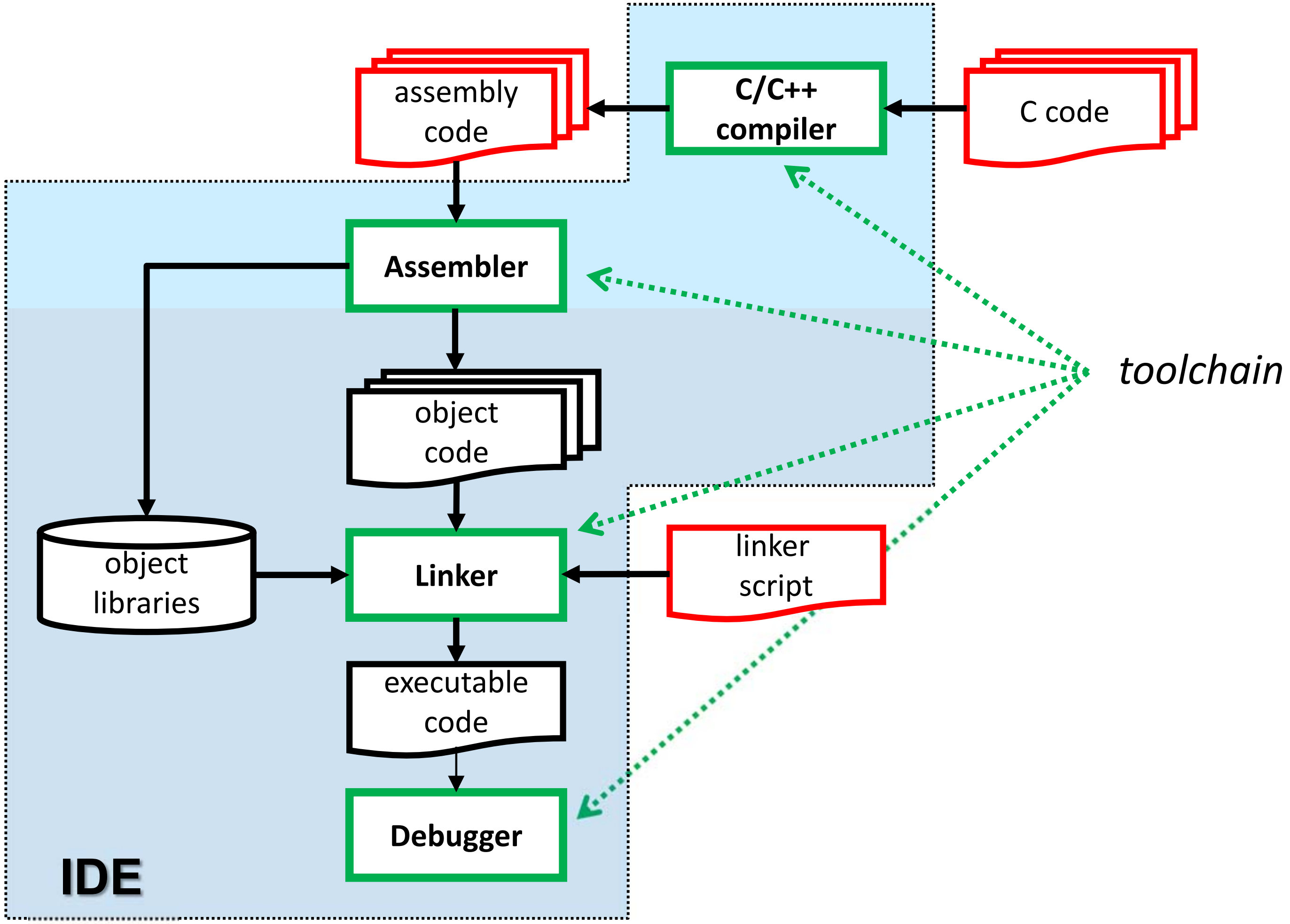
Local vs. global variables

Memory map



Development workflow

- To develop applications, an IDE (*Integrated Development Environment*) is used, which works as a toolchain interface.



Development workflow

- The assembler:
  - Interprets the assembly directives.
  - Expands pseudo-instructions and macros.
  - Transforms local labels of instructions and data into:
    - Branches and PC-relative offsets so that the code is relocatable.
  - Transforms constants and constant expressions into their binary representation.
  - Translates instructions into machine code.
  - Creates an object code file for each source file containing:
    - Header, sections, symbol table and debugging information.

- The linker, following the instructions in a script:
  - Combines object code input sections into output sections.
  - Assigns an adjacent memory region to each output section.
  - Resolves crossed references transforming the global labels.
  - Creates a unique file of executable machine code.

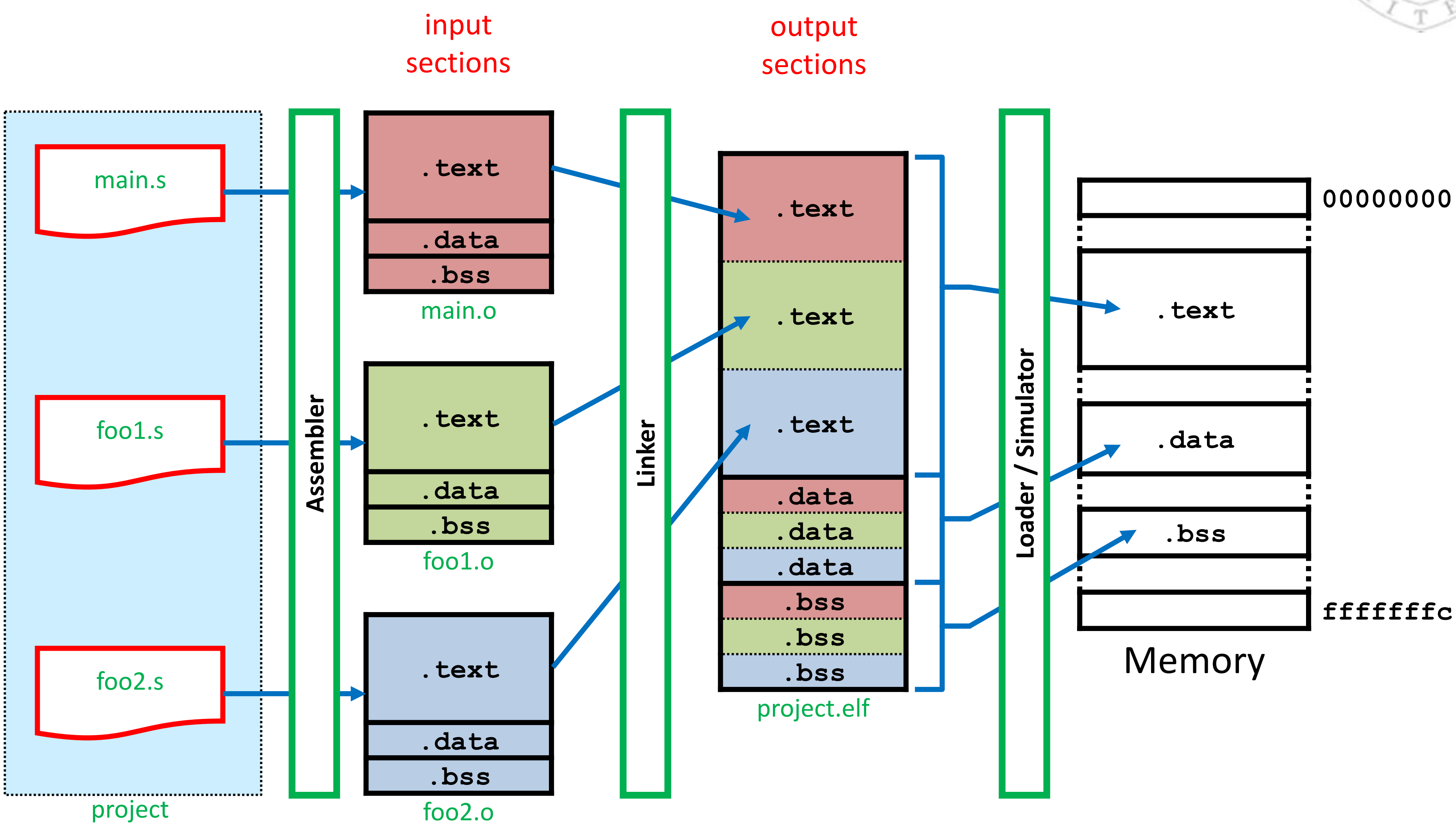
Development workflow

- The loader:
    - Copies in memory an executable file located in a secondary storage device and jumps to its first instruction.

- The debugger:
    - Allows executing a program instruction by instruction in order to inspect code and data.

- Applications can be developed in 2 scenarios:
    - Direct development: the application is compiled, assembled and linked in the same computer in which it is executed and debugged.
        - Or it is executed in a computer with the same architecture.
    - Crossed development: the application is compiled, assembled and linked in a different computer from the one in which it is executed and debugged.
        - Typically because the architectures of both computers are different.

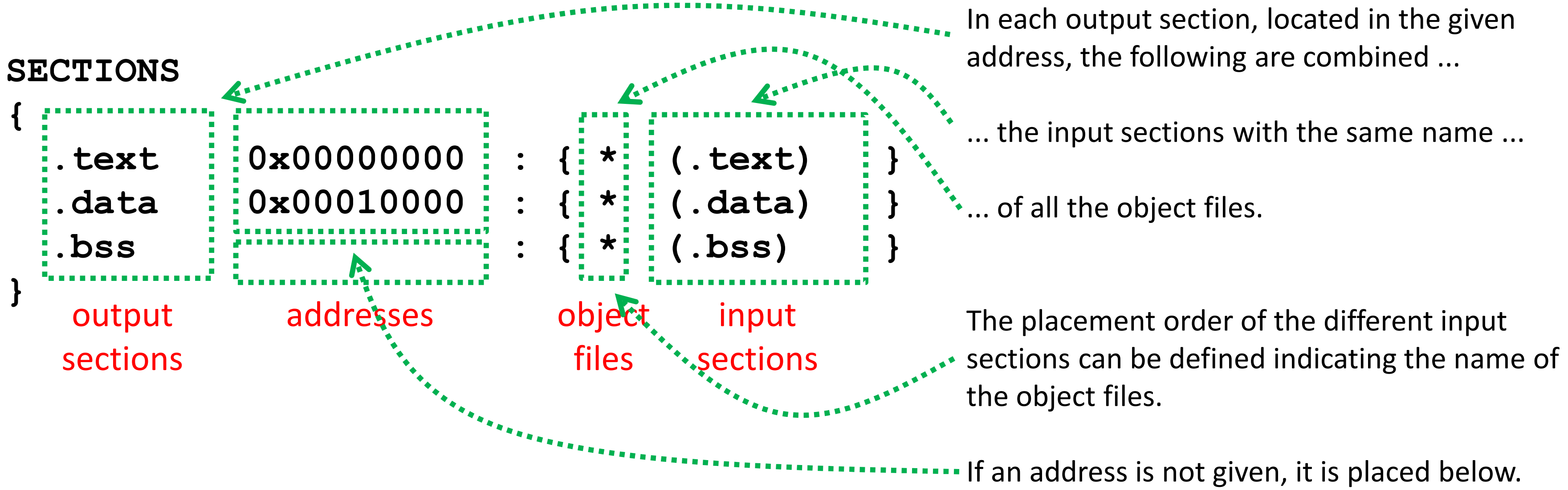
Development workflow



Development workflow

Linker script (i)

- A linker script is formed by a collection of commands that allow the programmer to direct the linking process.
  - It can define variables addressable by the source code.
  - The *location counter* (.) variable is predefined, which contains the current linking address of the output sections.
- The most important command is **`SECTIONS`**, which allows defining the name, location and content of the executable output sections.



```
SECTIONS
{
   .text    0x00000000  :  { *  (.text)  }
   .data    0x00010000  :  { *  (.data)  }
   .bss                 :  { *  (.bss)   }
}
```

In each output section, located in the given address, the following are combined ...

... the input sections with the same name ...

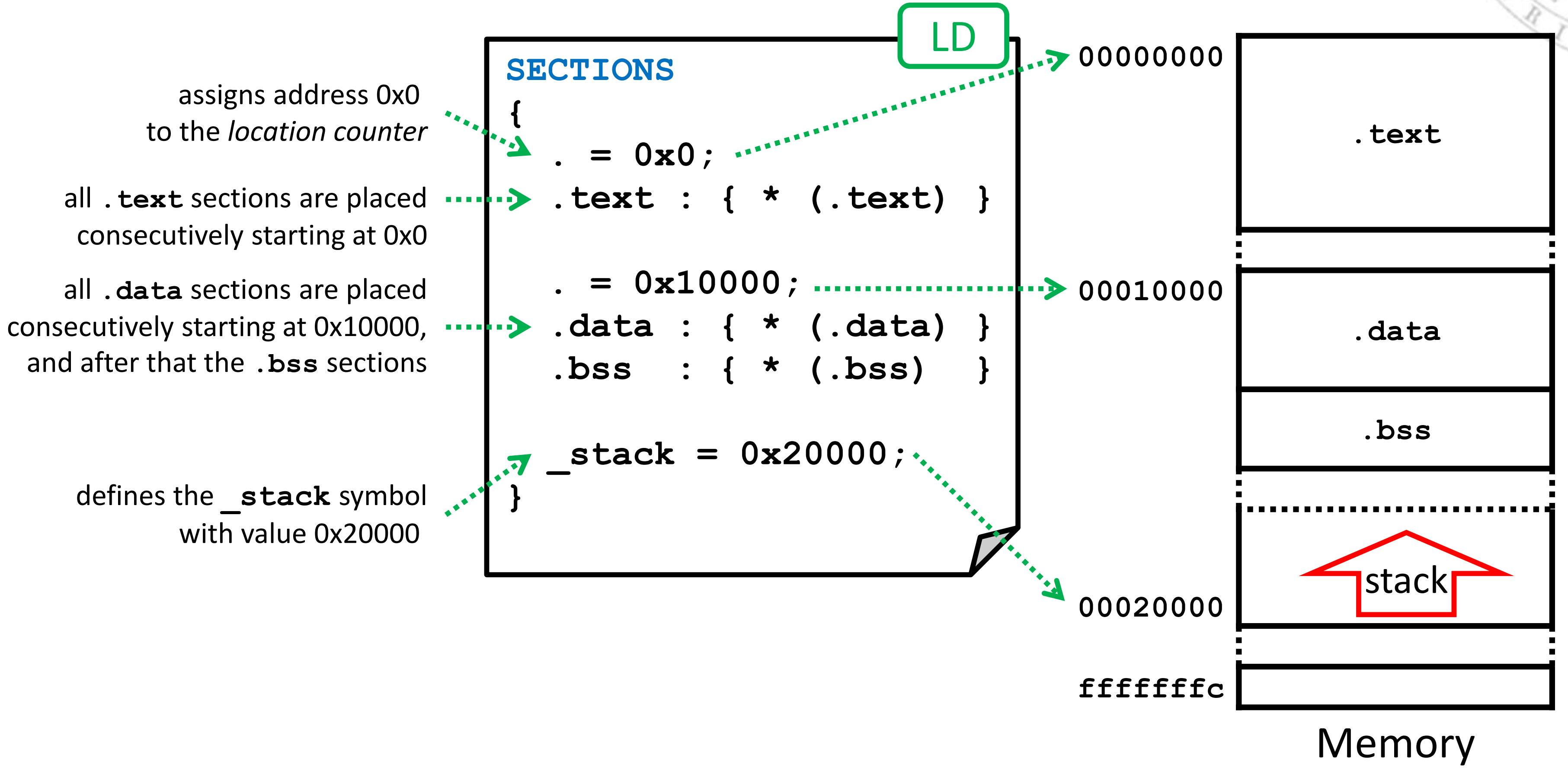
... of all the object files.

The placement order of the different input sections can be defined indicating the name of the object files.

If an address is not given, it is placed below.

Development workflow

Linker script (ii)



assigns address 0x0 to the *location counter*

all `.text` sections are placed consecutively starting at 0x0

all `.data` sections are placed consecutively starting at 0x10000, and after that the `.bss` sections

defines the `_stack` symbol with value 0x20000

```
SECTIONS
{
  . = 0x0;
  .text : { * (.text) }

  . = 0x10000;
  .data : { * (.data) }
  .bss  : { * (.bss)  }

  _stack = 0x20000;
}
```

- Addresses must be carefully chosen, in order to avoid errors
  - A stack overflow happens when the stack grows too much and it overwrites other sections of the program.

Development workflow

Combining assembly language and C/C++ (i)

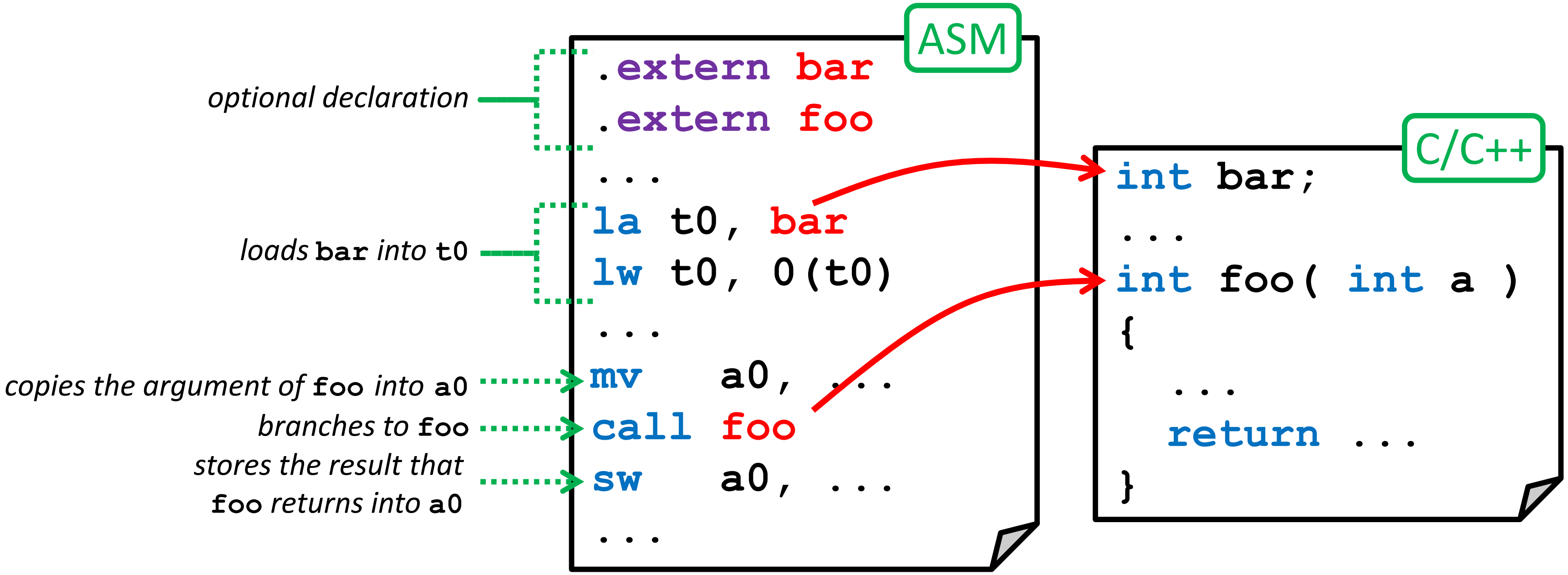
- The usual scenario is that a program is mostly written in C/C++, with just a very small portion in assembly.
  - Therefore, it will be necessary to call C/C++ functions and access global variables from the assembly code, and vice versa.

- By default:
  - The functions and global variables in C/C++ are visible from any project file.
    - However, their names are usually declared in the assembly code using the **`.extern`** directive, for readability's sake.
  - The functions and global variables in assembly are only visible inside the file where they are defined.
    - To make them visible outside, the **`.global`** directive must be used.
  - The C/C++ compiler follows the call convention defined in RISC-V
    - If the assembly programmer also follows this convention, the C/C++ and assembly codes will be able to interact seamlessly.

Development workflow

Combining assembly language and C/C++ (ii)

- To access a global variable declared in C/C++ from assembly:
  - Its identifier must be used as an operand.
- To call a function defined in C/C++ from assembly:
  - Its identifier is used as the branch destination.
  - The C/C++ function arguments must be passed following the RISC-V convention:
    - Using registers **`a0`**...**`a7`** for the first 8, and the rest through the stack.
  - The compiled C/C++ function will return the result in register **`a0`**.



ASM

```
.extern bar
.extern foo
...
la t0, bar
lw t0, 0(t0)
...
mv   a0, ...
call foo
sw   a0, ...
...
```

optional declaration

loads **`bar`** *into* **`t0`**

copies the argument of **`foo`** *into* **`a0`**

branches to **`foo`**

stores the result that **`foo`** *returns into* **`a0`**

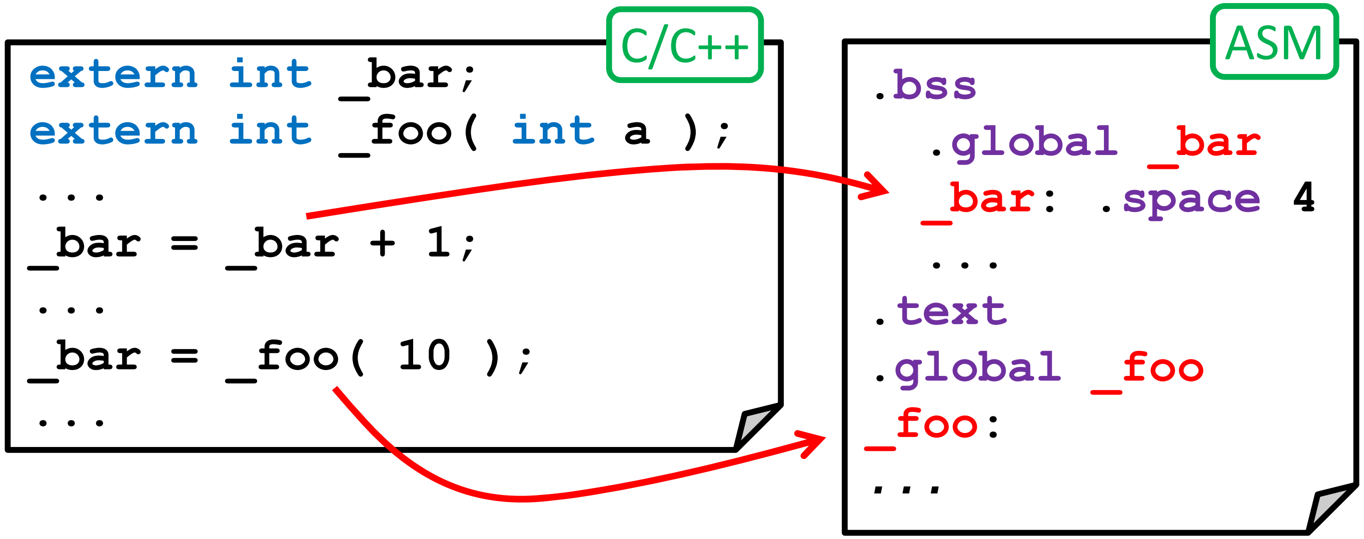
C/C++

```
int bar;
...
int foo( int a )
{
   ...
   return ...
}
```

Development workflow

Combining assembly language and C/C++ (iii)

- To access a global variable declared in assembler from C/C++:
  - It has to be declared as **`extern`** , and then it can be used normally.
- To call a function defined in assembly from C/C++:
  - Its prototype has to be declared as **`extern`**, and then it can be called normally.
  - The compiled C/C++ function passes its arguments following the RISC-V convention:
    - The assembly function will find the first 8 in registers **`a0...a7`** and the rest in the stack.
  - The assembly function must return the result in register **`a0`**.



C/C++

```
extern int _bar;
extern int _foo( int a );
...
_bar = _bar + 1;
...
_bar = _foo( 10 );
...
```

ASM

```
.bss
  .global _bar
  _bar: .space 4
  ...
.text
.global _foo
_foo:
...
```

Development workflow

Starting a program

- In C/C++, the **`main`** program is handled as another function.
  - With its own arguments, its return value and its local variables.
  - It is the operating system which, in order to execute a program, calls its **`main`** function, passing the arguments and receiving the return value.
  - Besides, the operating system has previously initialized the system.
- In computers without an operating system (*bare metal*), the system initialization and the branch to **`main`** are done by the startup code.

ASM

```
.extern main
.extern _stack
.text
.global start
start:
   la    sp, _stack
   mv    fp, sp
   call  main
   j     .
.end
```

*By default, the linker recognizes the **`start`** label as the address of the first instruction of the program*

*Initializes **`sp`***

*Initializes **`fp`***

*Calls **`main`** without arguments*

*If it returns from **`main`**, it remains here indefinitely*

About *Creative Commons*

- CC license (Creative Commons)
  - This license enables reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix, adapt, or build upon the material, you must license the modified material under identical terms:

Attribution:
Credit must be given to the creator.

Non commercial:
Only noncommercial uses of the work are permitted.

Share alike:
Adaptations must be shared under the same terms.

More information: https://creativecommons.org/licenses/by-nc-sa/4.0/