Module 1:

From digital systems to computers

Introduction to computers II

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

Outline

- ✓ Application-specific circuit
- ✓ General purpose data paths + controller.
- ✓ General purpose circuit: computer.
- ✓ Von-Neumann model.
- ✓ Processor architecture.
- ✓ Processor organization.
- ✓ Other basic concepts.

These slides are based on:

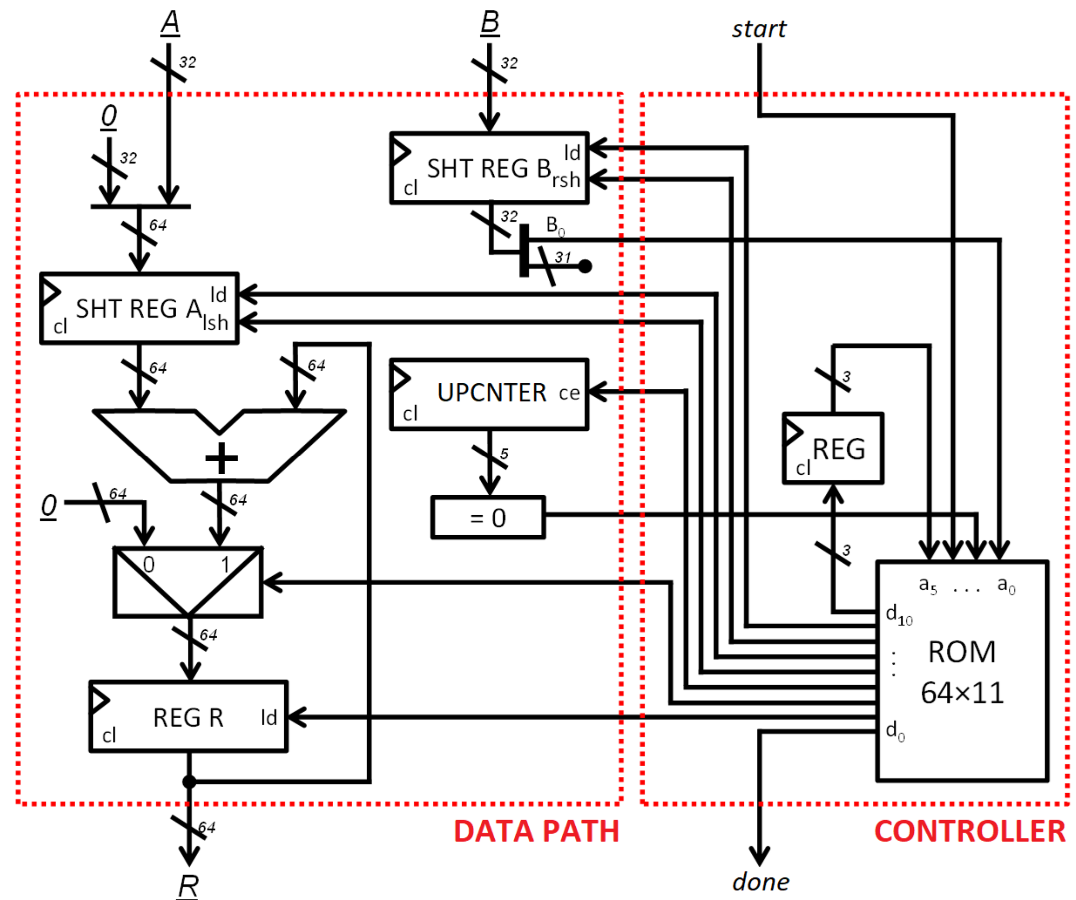
- Katzalin Olcoz et al. *Fundamentos de Computadores II. UCM*
- Chris Terman. *Computation Structures. MIT Open Courseware*

Application-specific circuit

- An algorithm is implemented in hardware connecting:
  - Data path: performs operations and stores partial results.
  - Controller: sequences the operations according to the algorithm.

```
A = Ain;
B = Bin;
R = 0;
for( C=0; C<32; C++ )
{
  if( B0==1 )
    R = R+A;
  A = A<<1;
  B = B>>1;
};
Rout = R;
```

Algorithm to multiply two ***32-bit*** *numbers*

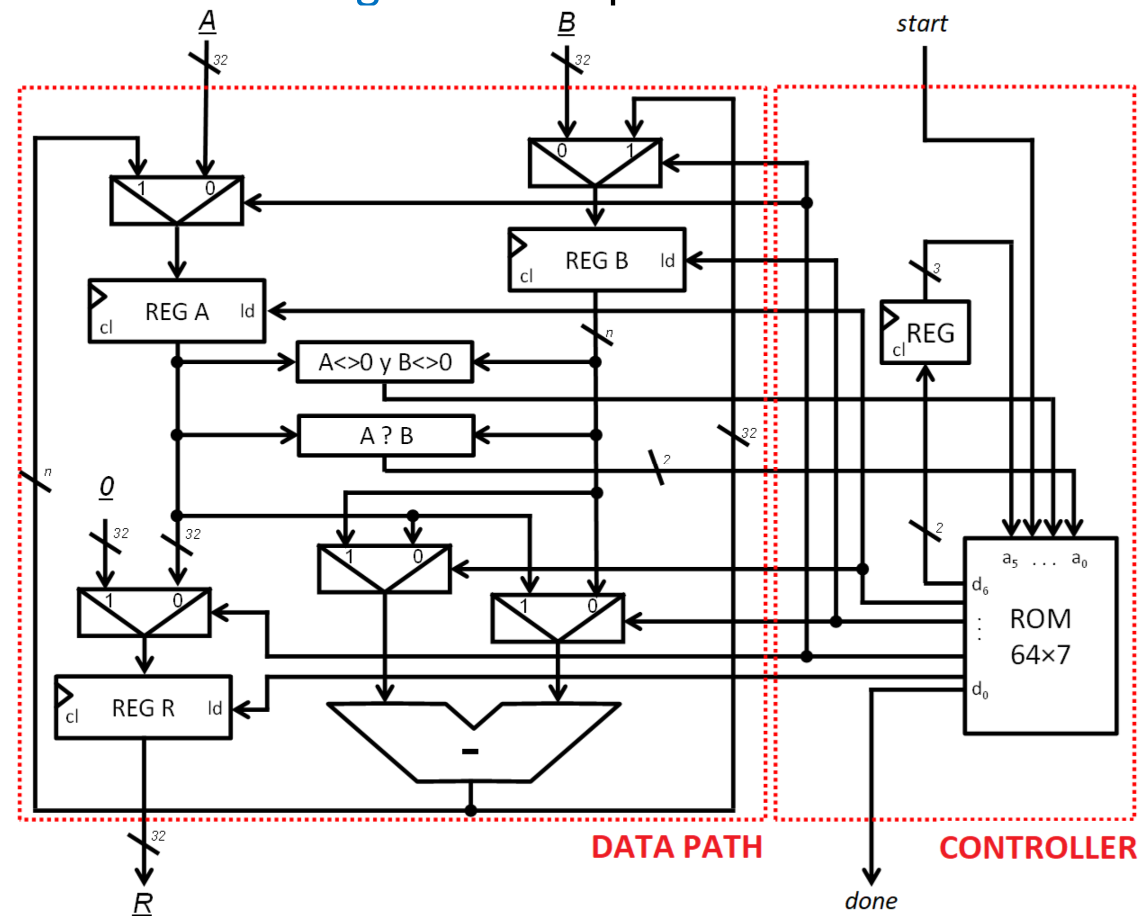


Application-specific circuit

- The circuits obtained through algorithmic design are very efficient, but they have one single behavior.
  - Different algorithms require different circuits.
  - To change its behavior, a hardware redesign must be performed.

```
A = Ain;
B = Bin;
R = 0;
if( A!=0 && B!=0 )
{
  while( A!=B )
    if( A>B )
      A = A-B;
    else
      B = B-A;
  R = A;
};
Rout = R;
```

Algorithm to calculate the GCD of two ***32-bit*** *numbers*



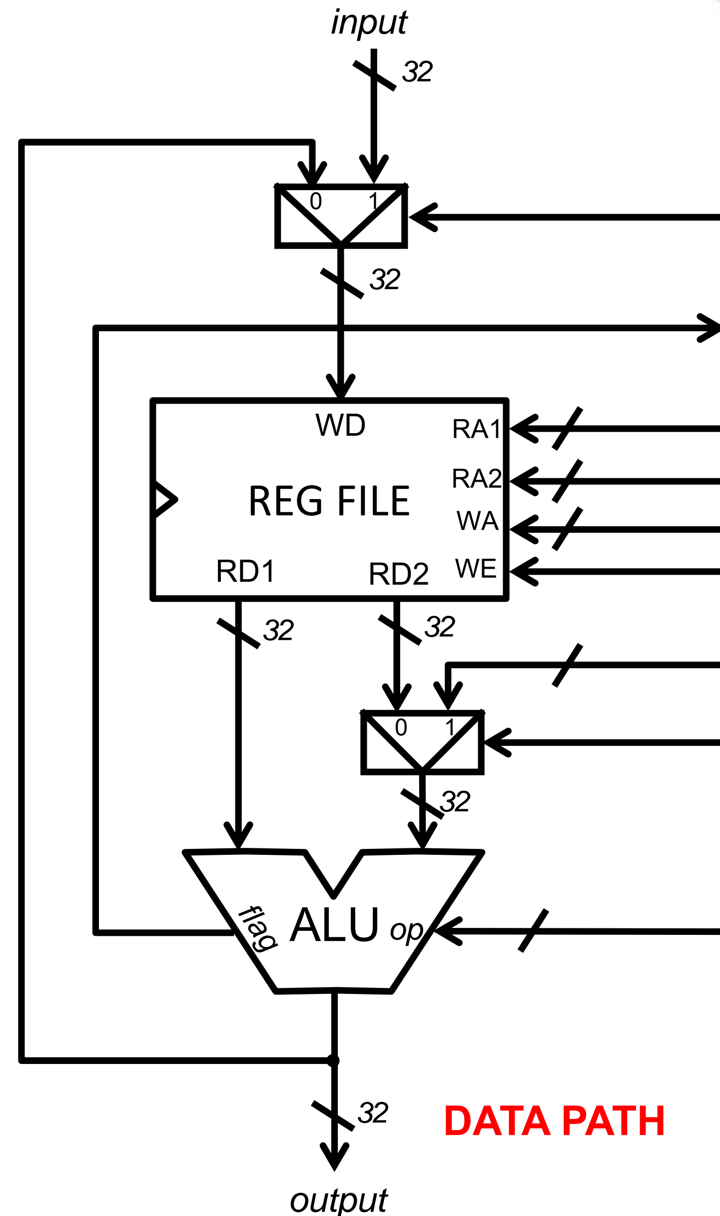
Application-specific circuit

- In general, these data paths are very specific because for each algorithm:
  - The number of registers depends on the number of variables
  - The number and type of functional units depends on the number and type of calculations to be performed.
  - The number of connections is maximized to perform simultaneously as many register transfers in parallel as possible.
  - The width of each interconnection depends on the widths required by each calculation.

- Besides, these controllers are very specific because:
  - The number of control/status signals of each data path is different.
  - They follow a fixed state sequence stored in the ROM

General purpose data path

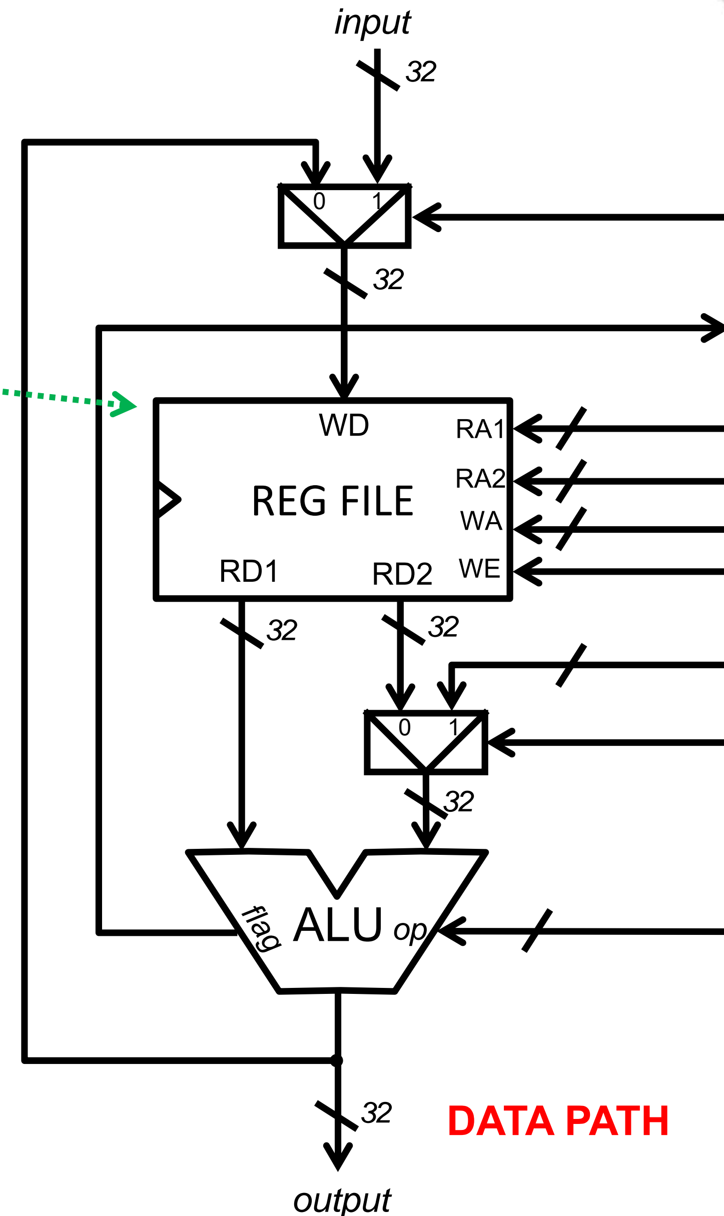
- However, it is possible to design a more generic data path:



General purpose data path

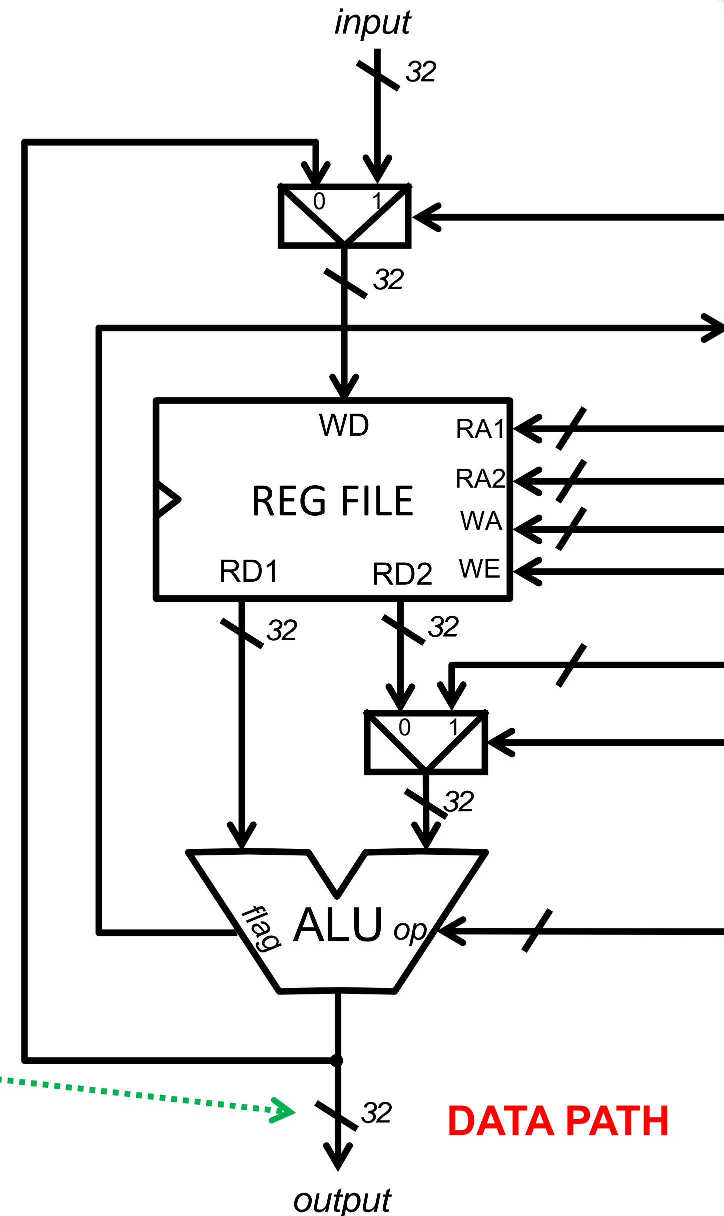
- However, it is possible to design a more generic data path:

A register file with a large enough number of registers is used



General purpose data path

- However, it is possible to design a more generic data path:

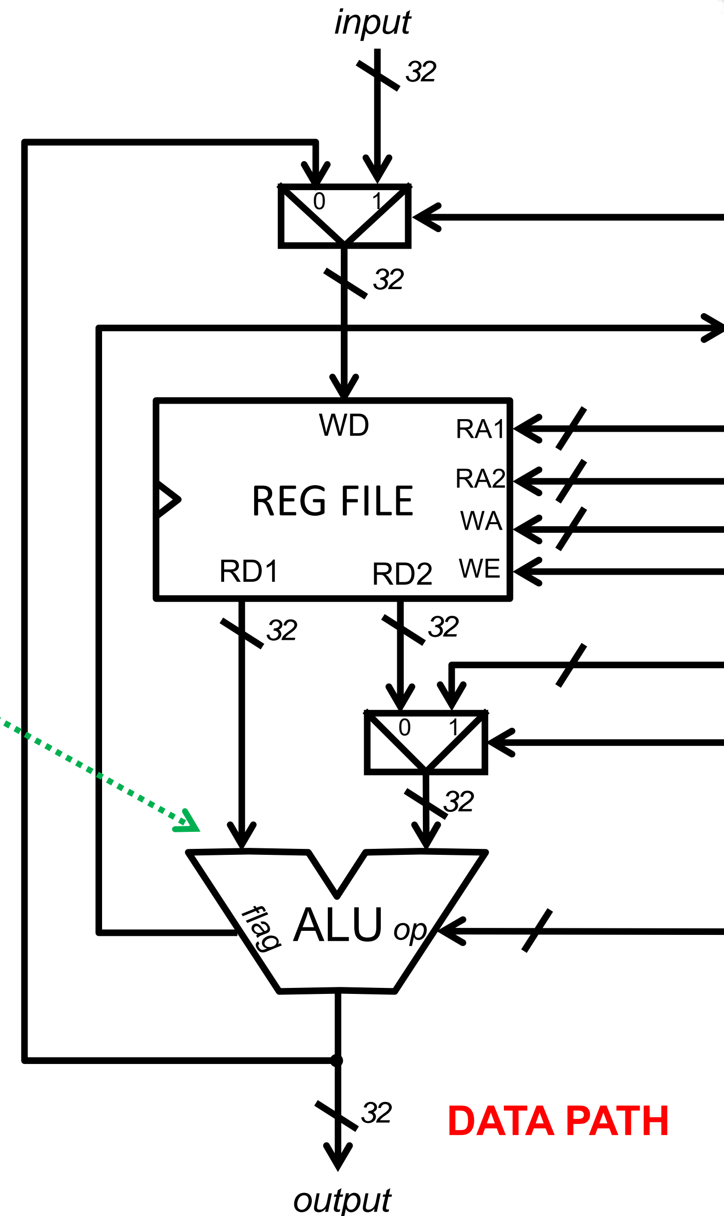


A homogeneous data width is chosen, wide enough for all the interconnections

General purpose data path

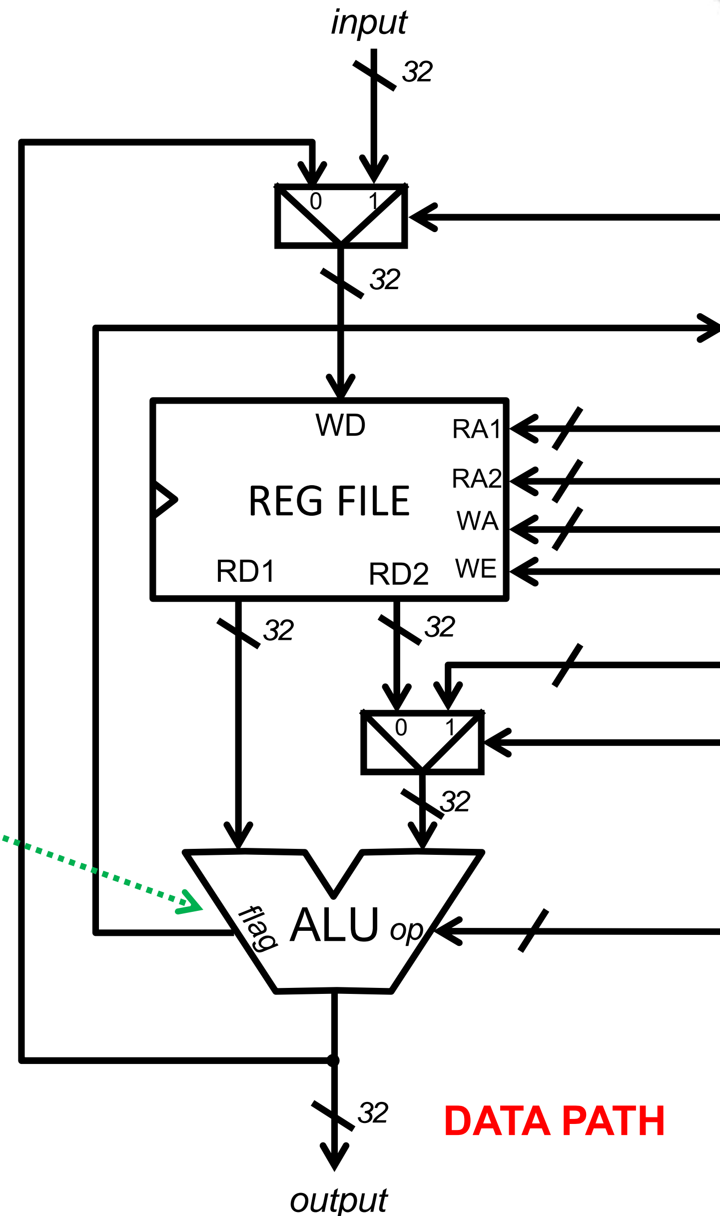
- However, it is possible to design a more generic data path:

A generic ALU is chosen, able to perform a wide enough range of arithmetic, logic and relational operations



General purpose data path

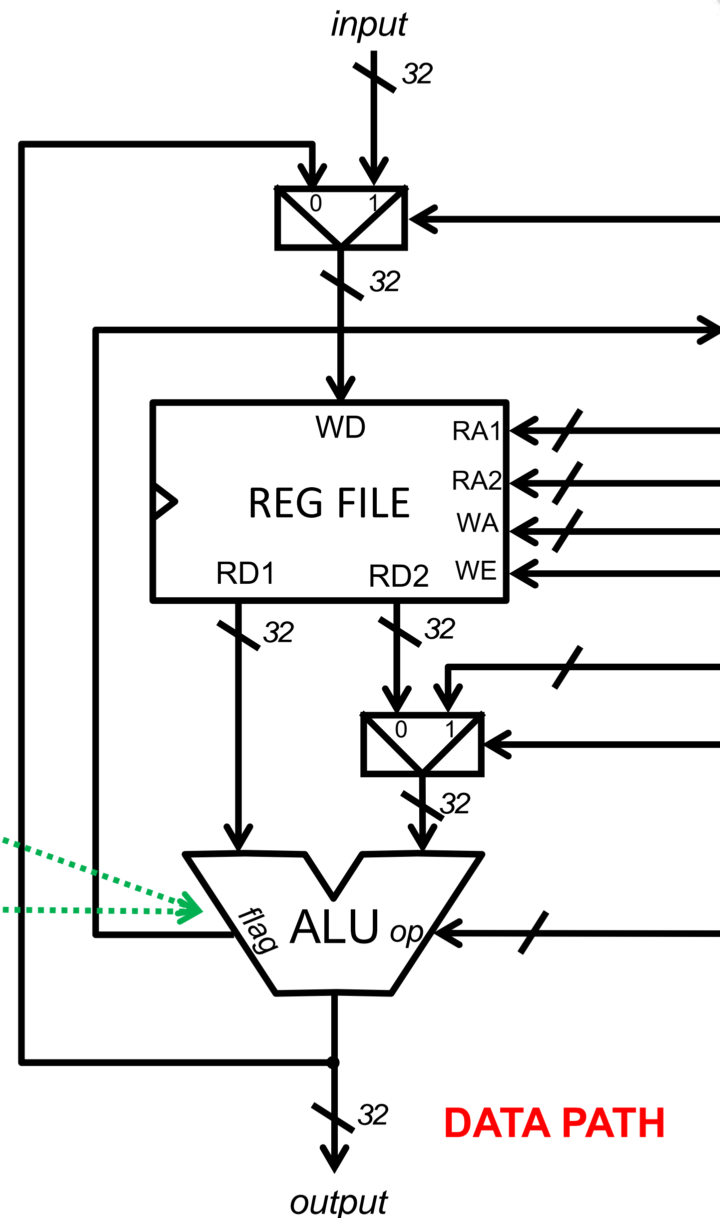
- However, it is possible to design a more generic data path:



The ALU has enough flags to indicate if the operands meet any kind of relation

General purpose data path

- However, it is possible to design a more generic data path:



The ALU has enough flags to indicate if the operands meet any kind of relation

Just for this case, let us assume that there is only one flag that takes the value of 1 when the ALU performs a comparison operation that it true

General purpose data path

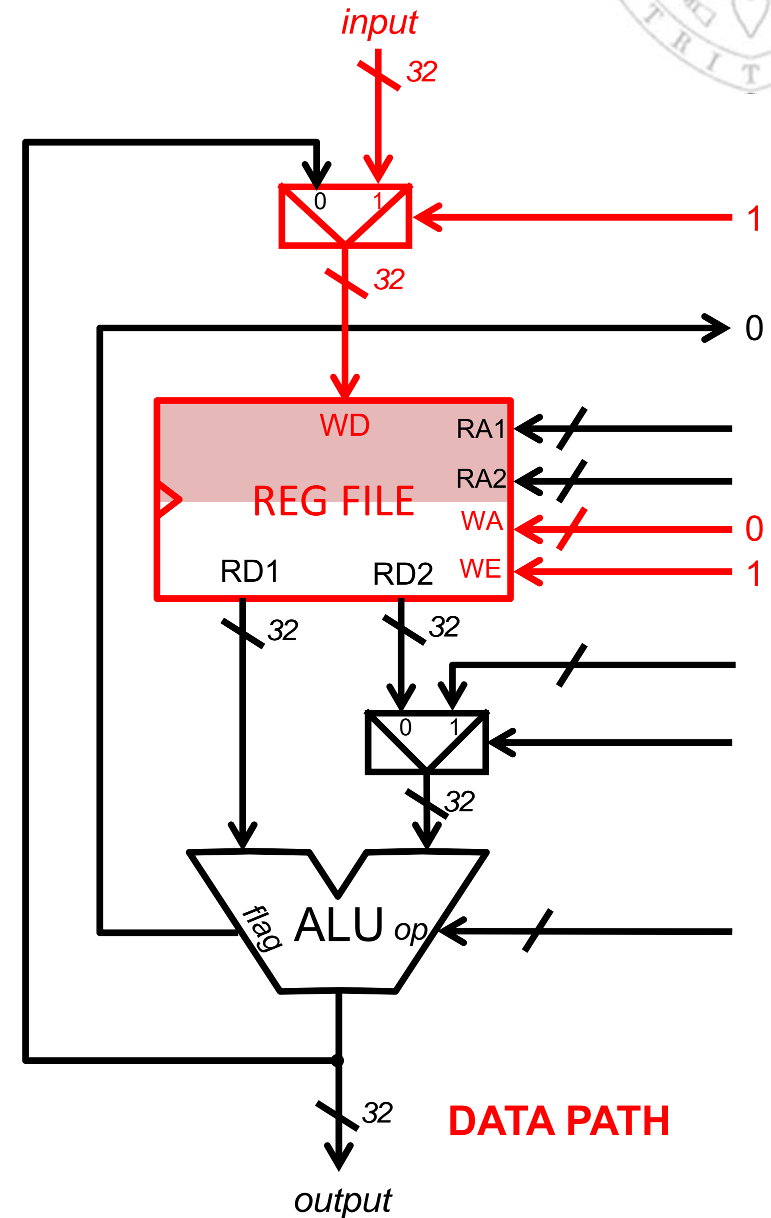
Multiplication

Algorithm to multiply two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
for( C=0; C<=31; C++ )
{
 if( B0==1 )

   R = R+A;
 A = A<<1;
 B = B>>1;
};

Rout = R;
```

Inputs and register transfers

S0	R0 ← input



R0 is A

General purpose data path

Multiplication

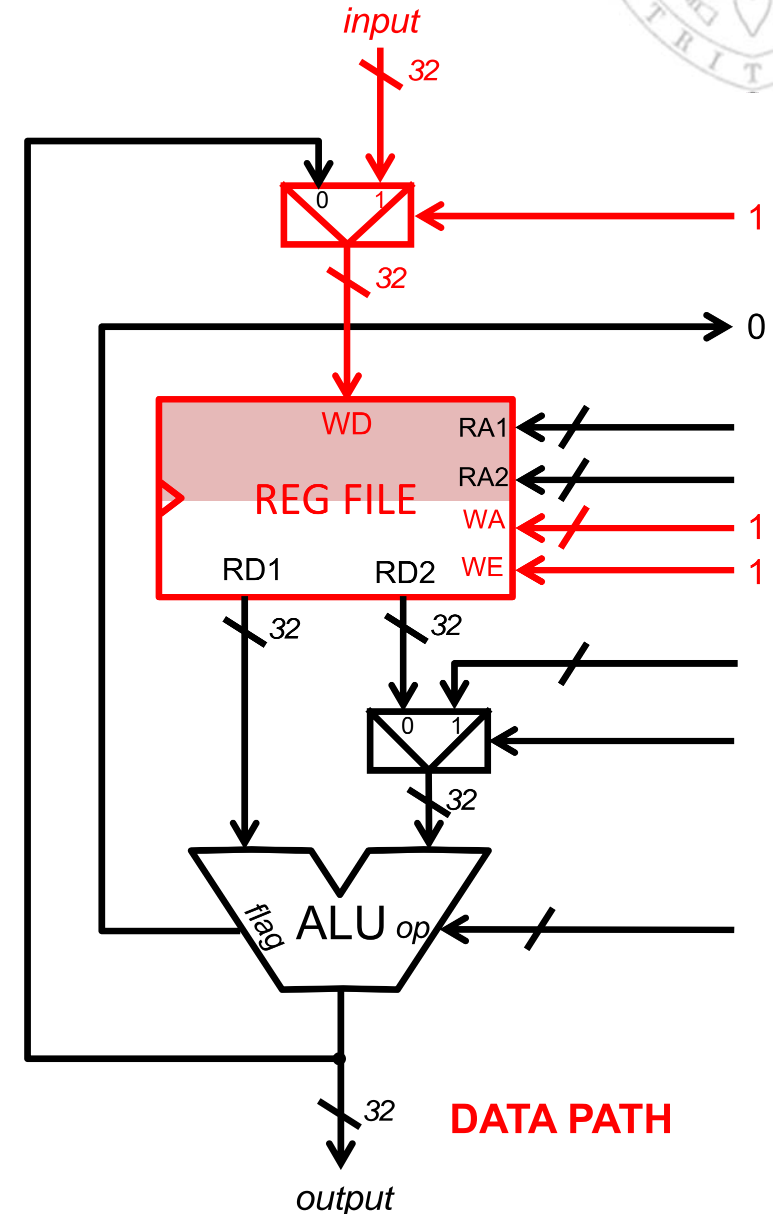
Algorithm to multiply two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
for( C=0; C<=31; C++ )
{
 if( B0==1 )

   R = R+A;
 A = A<<1;
 B = B>>1;
};

Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input

R0 is A   R1 is B



General purpose data path

Multiplication

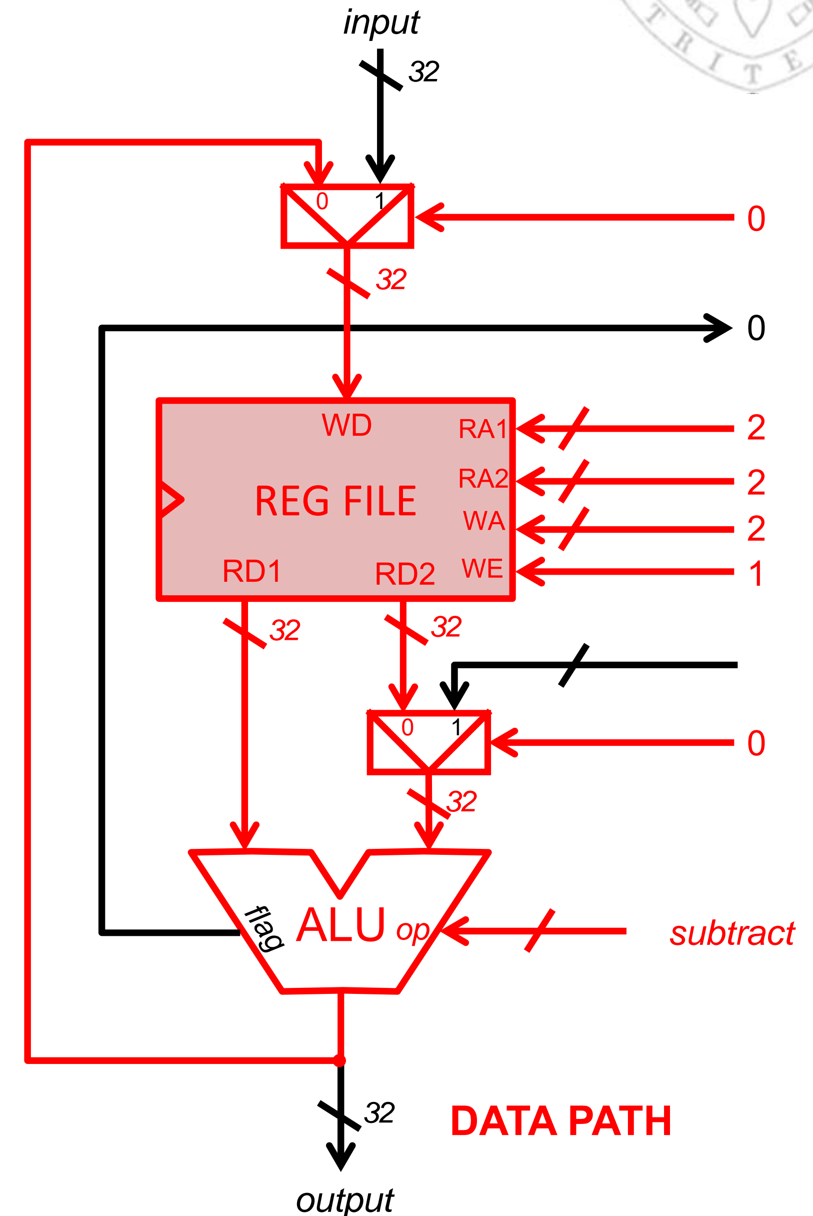
Algorithm to multiply two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
for( C=0; C<=31; C++ )
{
 if( B0==1 )

   R = R+A;
 A = A<<1;
 B = B>>1;
};

Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2

R0 is A   R1 is B   R2 is R



General purpose data path

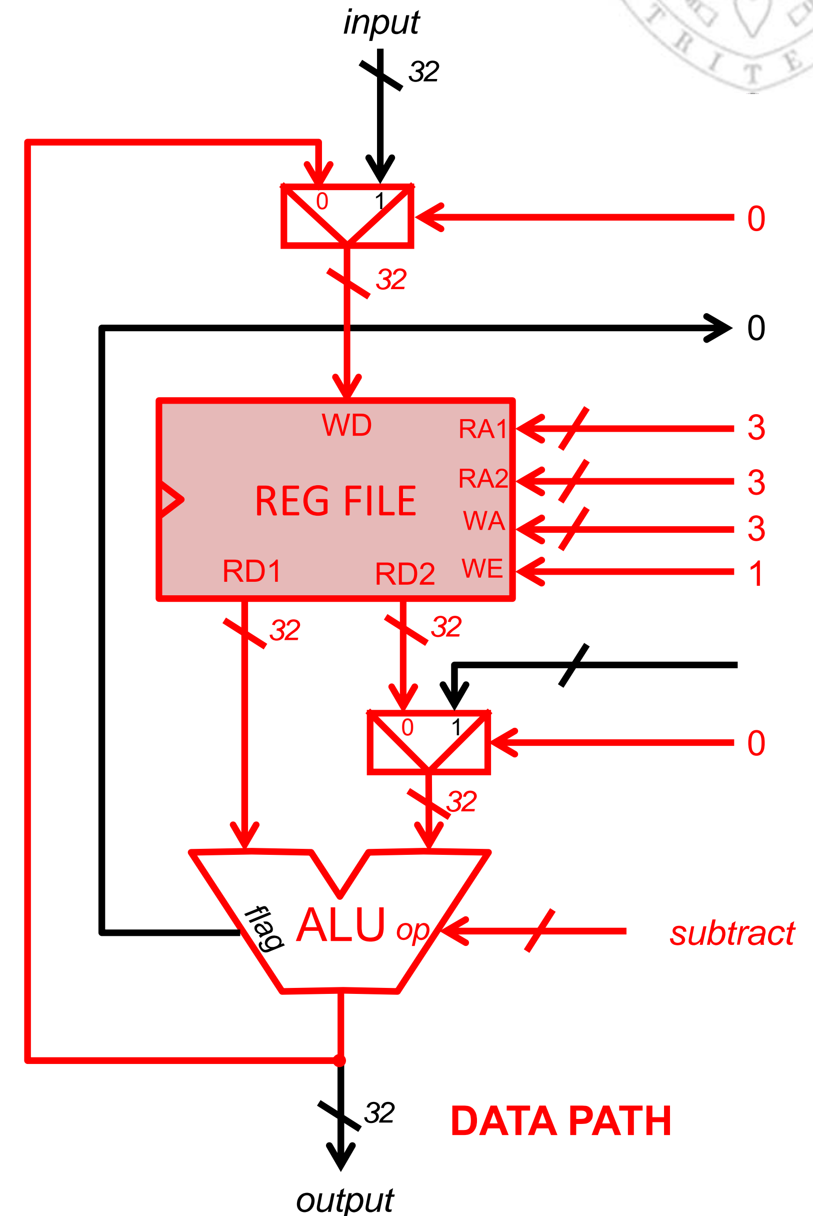
Multiplication

Algorithm to multiply two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
for( C=0; C<=31; C++ )
{
 if( B0==1 )

   R = R+A;
 A = A<<1;
 B = B>>1;
};

Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	R3 ← R3 – R3



R0 is A R1 is B R2 is R R3 is C

General purpose data path

Multiplication

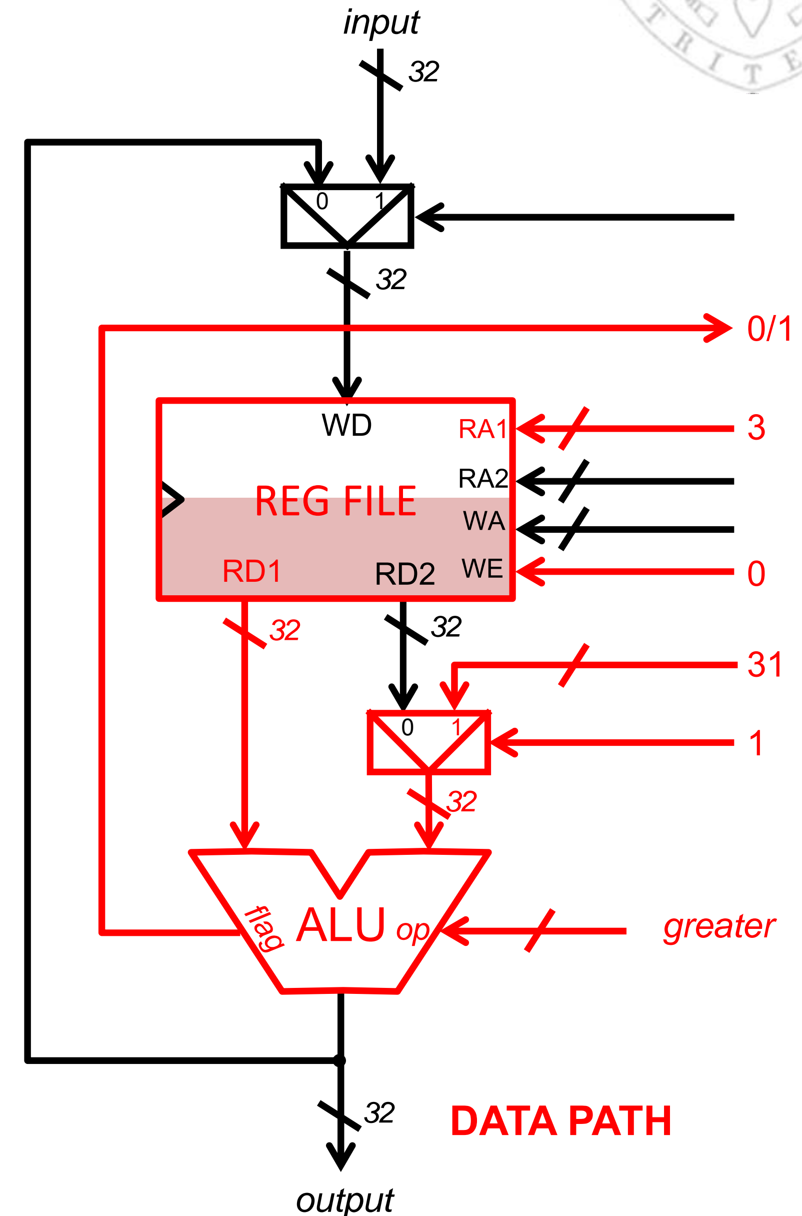
Algorithm to multiply two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
for( C=0; C<=31; C++ )
{
 if( B0==1 )

   R = R+A;
 A = A<<1;
 B = B>>1;
};

Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	R3 ← R3 – R3
S4	if R3 > 31, go to S12

R0 is A R1 is B R2 is R R3 is C



General purpose data path

Multiplication

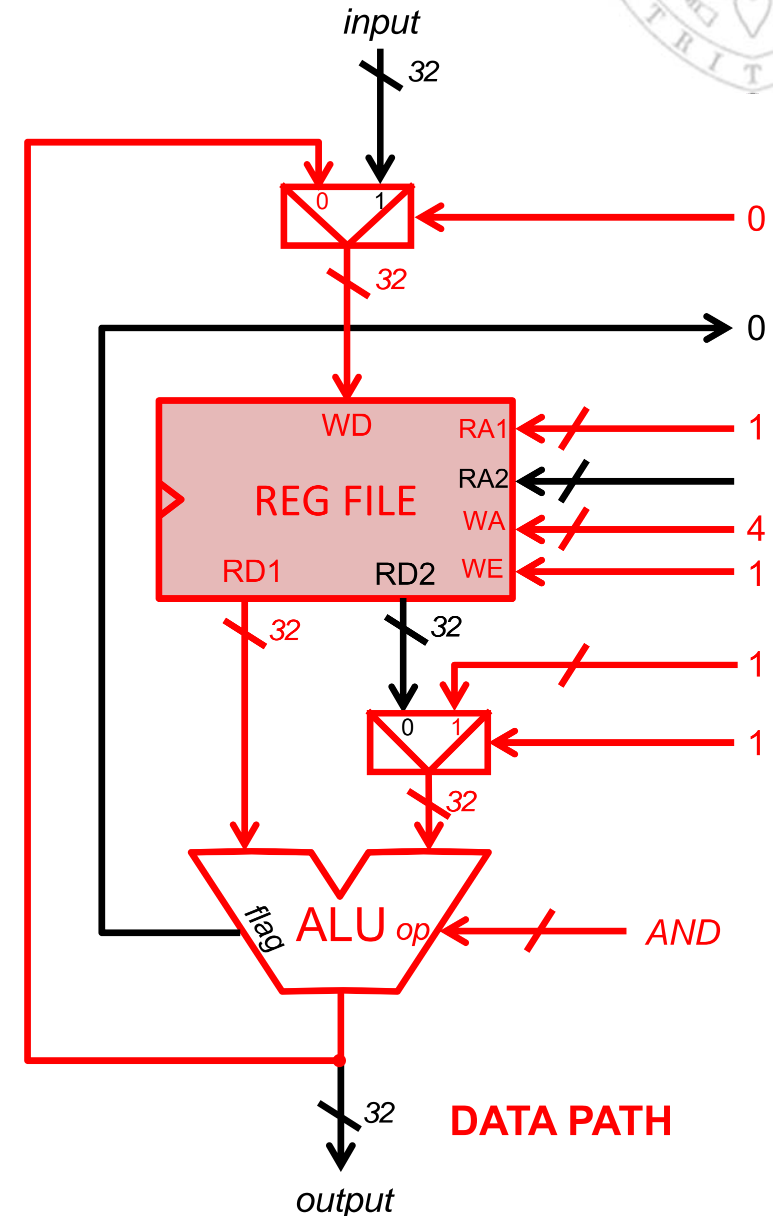
Algorithm to multiply two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
for( C=0; C<=31; C++ )
{
 if( B0==1 )

   R = R+A;
 A = A<<1;
 B = B>>1;
};

Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	R3 ← R3 – R3
S4	if R3 > 31, go to S12
S5	R4 ← R1 & 1

R0 is A   R1 is B   R2 is R   R3 is C



General purpose data path

Multiplication

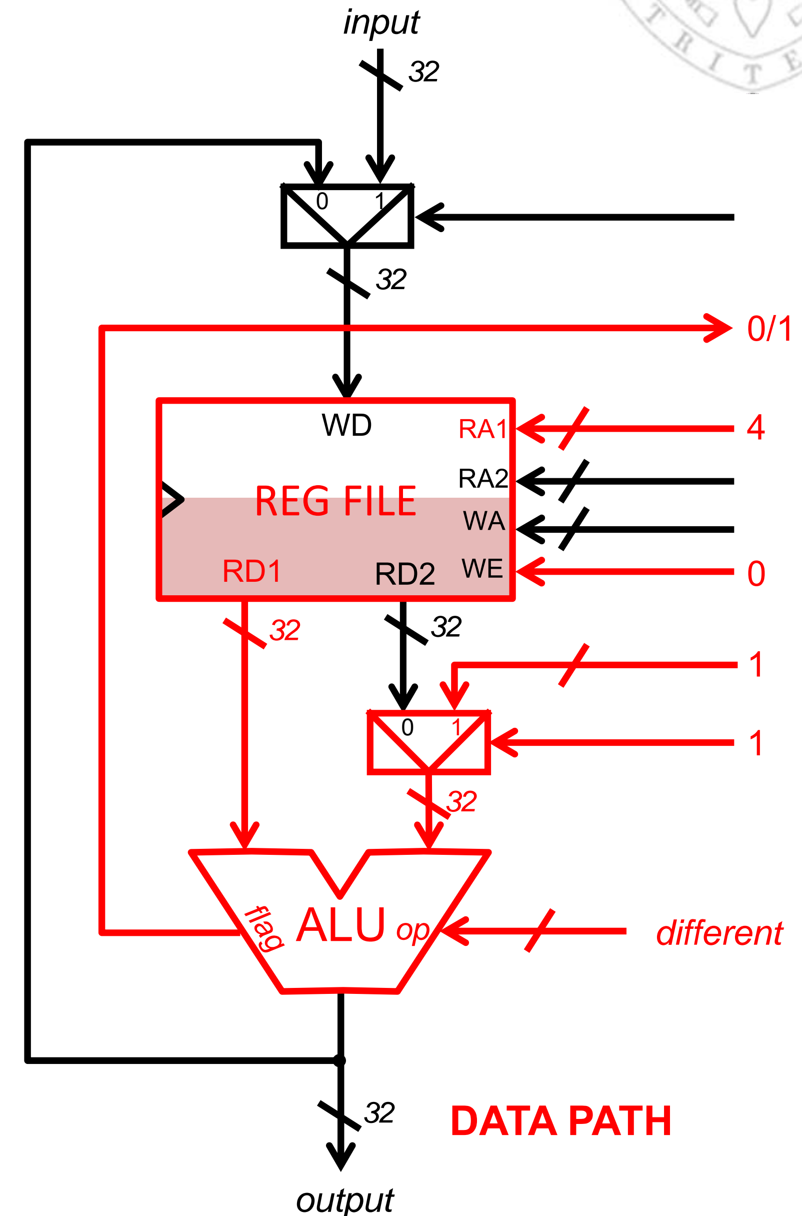
Algorithm to multiply two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
for( C=0; C<=31; C++ )
{
 if( B0==1 )

   R = R+A;
 A = A<<1;
 B = B>>1;
};

Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	R3 ← R3 – R3
S4	if R3 > 31, go to S12
S5	R4 ← R1 & 1
S6	if R4 != 1, go to S8

R0 is A R1 is B R2 is R R3 is C



General purpose data path

Multiplication

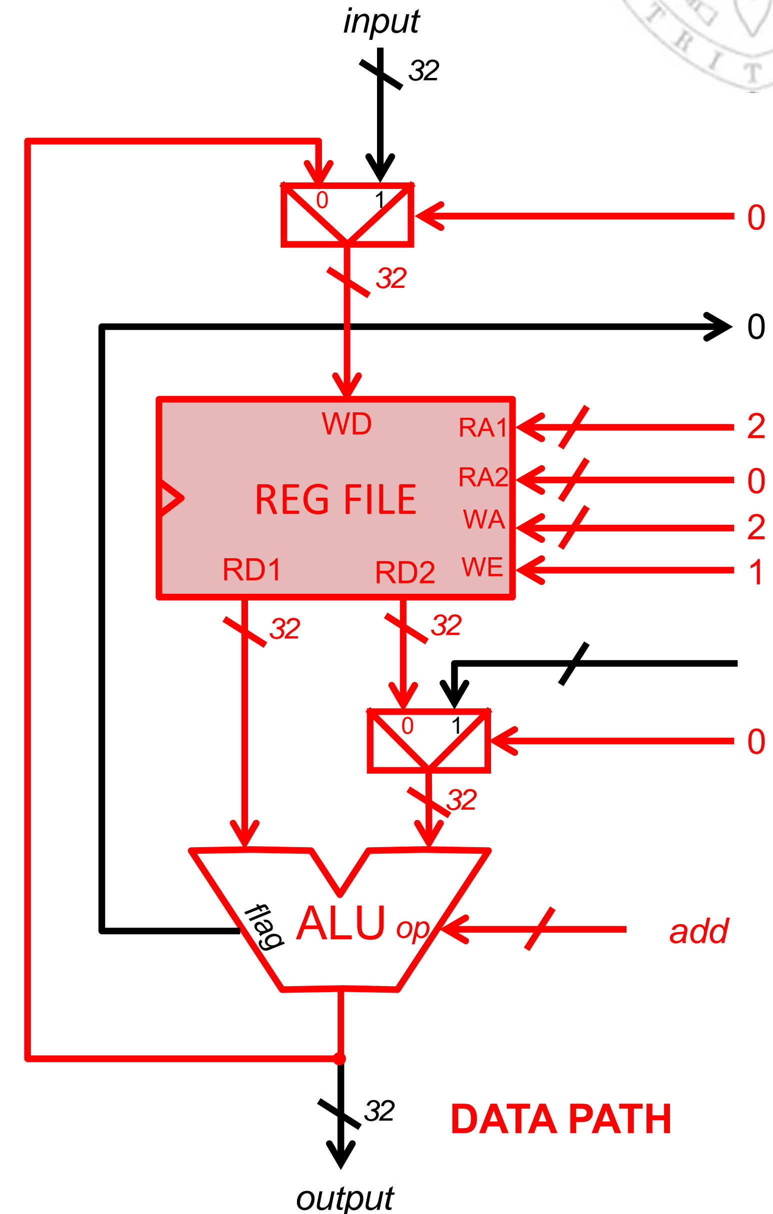
Algorithm to multiply two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
for( C=0; C<=31; C++ )
{
 if( B0==1 )

    R = R+A;
 A = A<<1;
 B = B>>1;
};

Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	R3 ← R3 – R3
S4	if R3 > 31, go to S12
S5	R4 ← R1 & 1
S6	if R4 != 1, go to S8
S7	R2 ← R2 + R0

R0 is A R1 is B R2 is R R3 is C



General purpose data path

Multiplication

Algorithm to multiply two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
for( C=0; C<=31; C++ )
{
 if( B0==1 )

   R = R+A;
 A = A<<1;
 B = B>>1;
};

Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	R3 ← R3 – R3
S4	if R3 > 31, go to S12
S5	R4 ← R1 & 1
S6	if R4 != 1, go to S8
S7	R2 ← R2 + R0
S8	R0 ← R0 << 1

R0 is A R1 is B R2 is R R3 is C

General purpose data path

Multiplication

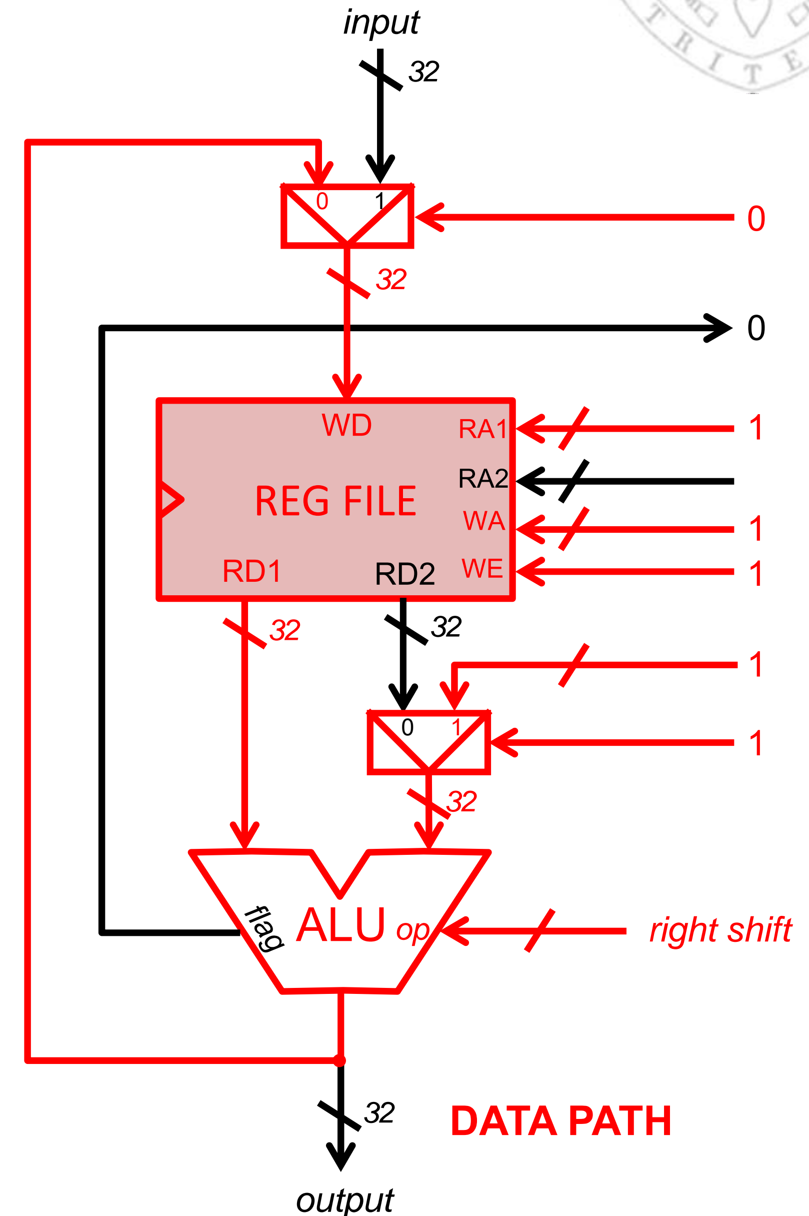
Algorithm to multiply two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
for( C=0; C<=31; C++ )
{
 if( B0==1 )

   R = R+A;
 A = A<<1;
 B = B>>1;
};

Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	R3 ← R3 – R3
S4	if R3 > 31, go to S12
S5	R4 ← R1 & 1
S6	if R4 != 1, go to S8
S7	R2 ← R2 + R0
S8	R0 ← R0 << 1
S9	R1 ← R1 >> 1

R0 is A R1 is B R2 is R R3 is C



General purpose data path

Multiplication

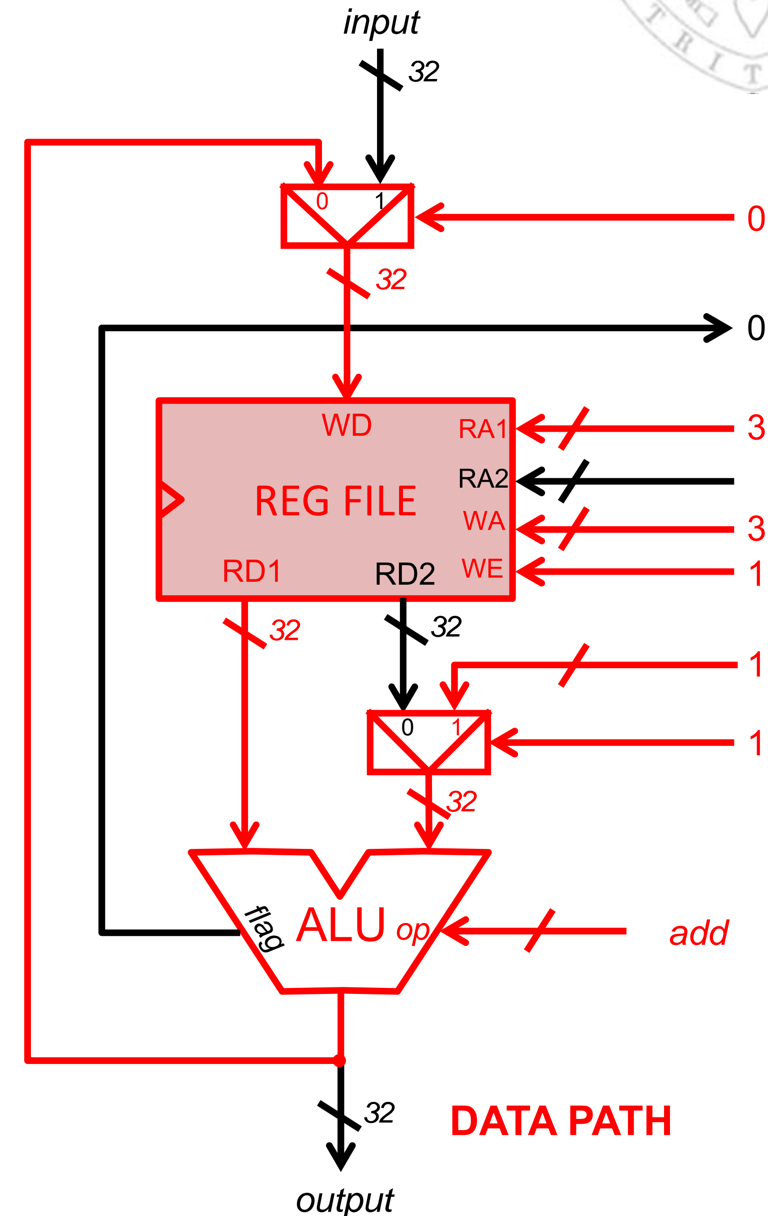
Algorithm to multiply two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
for( C=0; C<=31; C++ )
{
 if( B0==1 )

    R = R+A;
  A = A<<1;
  B = B>>1;
};

Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	R3 ← R3 – R3
S4	if R3 > 31, go to S12
S5	R4 ← R1 & 1
S6	if R4 != 1, go to S8
S7	R2 ← R2 + R0
S8	R0 ← R0 << 1
S9	R1 ← R1 >> 1
S10	R3 ← R3 + 1

R0 is A R1 is B R2 is R R3 is C



General purpose data path

Multiplication

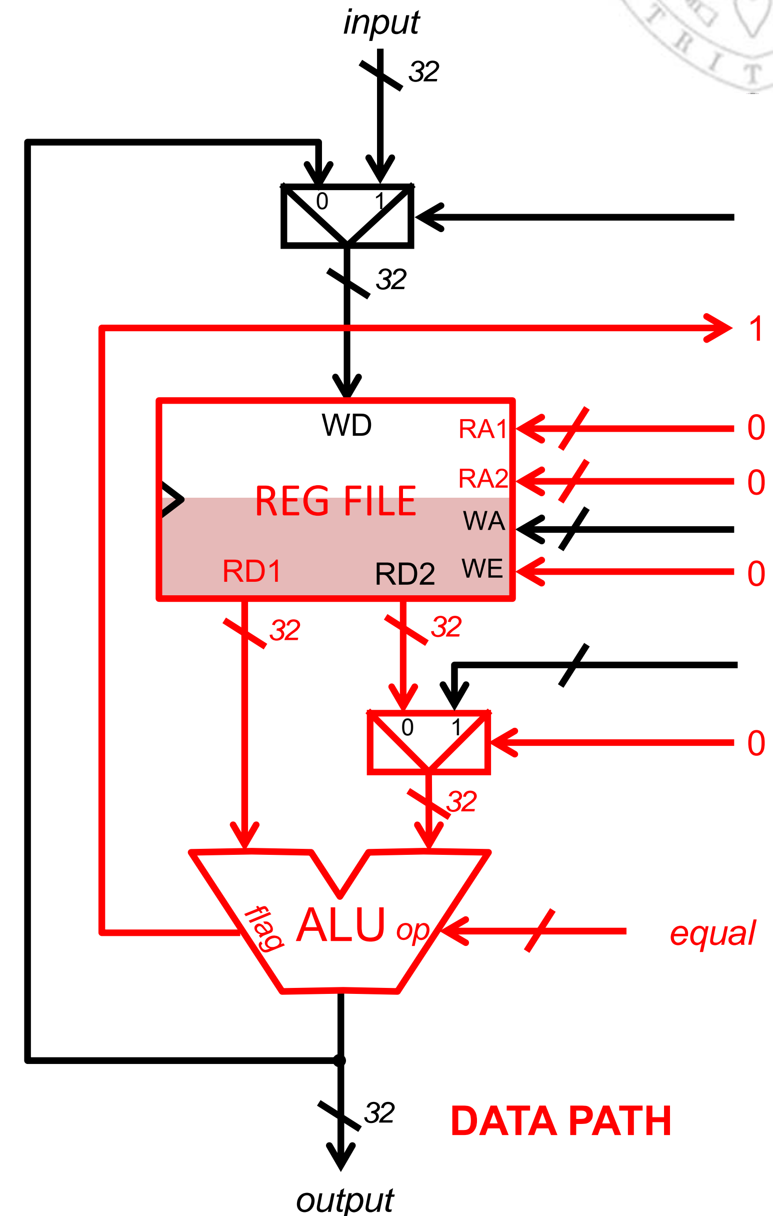
Algorithm to multiply two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
for( C=0; C<=31; C++ )
{
 if( B0==1 )

   R = R+A;
  A = A<<1;
  B = B>>1;
};

Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	R3 ← R3 – R3
S4	if R3 > 31, go to S12
S5	R4 ← R1 & 1
S6	if R4 != 1, go to S8
S7	R2 ← R2 + R0
S8	R0 ← R0 << 1
S9	R1 ← R1 >> 1
S10	R3 ← R3 + 1
S11	if R0 == R0, go to S4

R0 is A R1 is B R2 is R R3 is C



General purpose data path

Multiplication

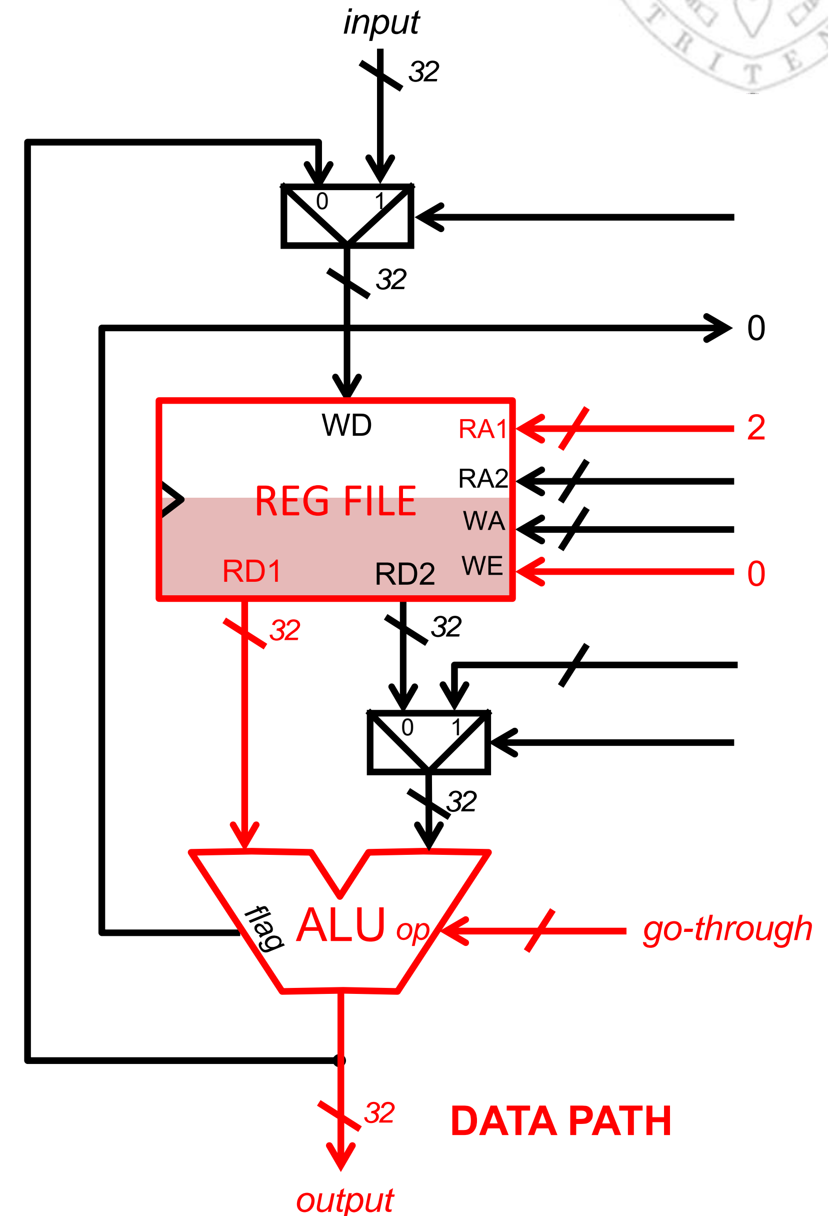
Algorithm to multiply two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
for( C=0; C<=31; C++ )
{
 if( B0==1 )

   R = R+A;
  A = A<<1;
  B = B>>1;
};

Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	R3 ← R3 – R3
S4	if R3 > 31, go to S12
S5	R4 ← R1 & 1
S6	if R4 != 1, go to S8
S7	R2 ← R2 + R0
S8	R0 ← R0 << 1
S9	R1 ← R1 >> 1
S10	R3 ← R3 + 1
S11	if R0 == R0, go to S4
S12	output ← R2

R0 is A R1 is B R2 is R R3 is C



General purpose data path

Multiplication

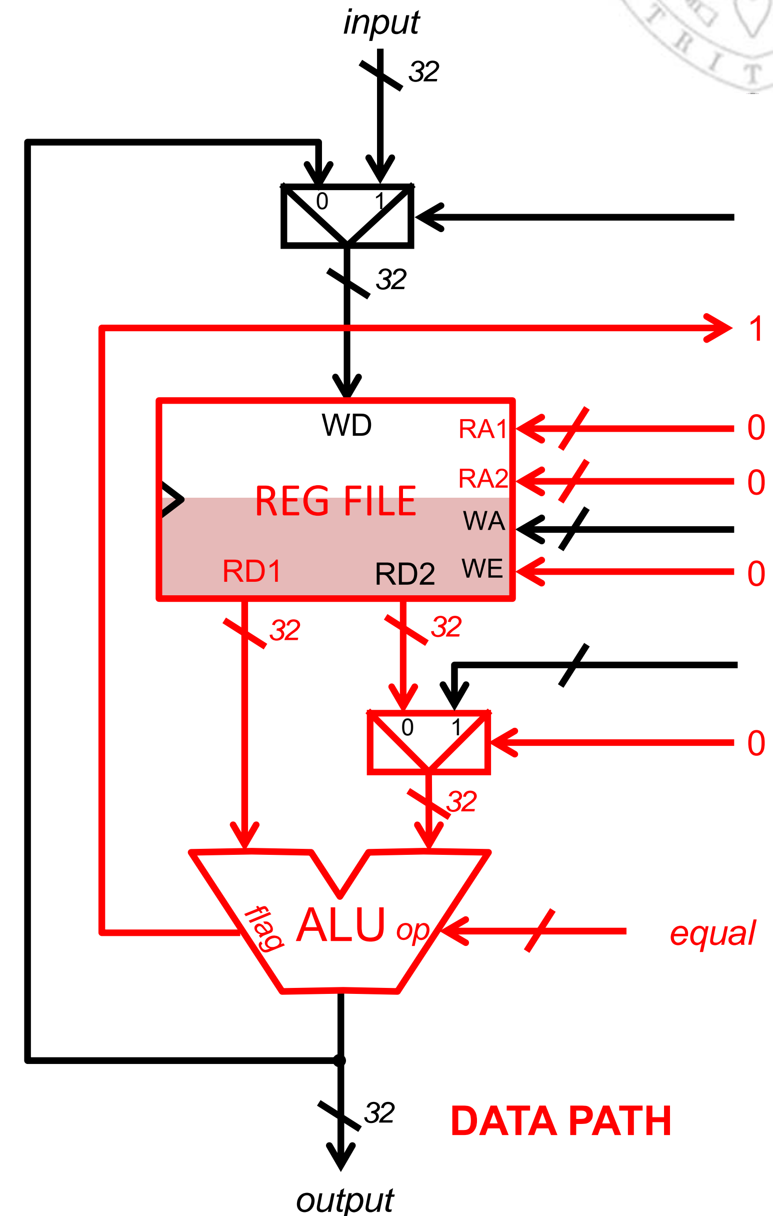
Algorithm to multiply two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
for( C=0; C<=31; C++ )
{
 if( B0==1 )

   R = R+A;
  A = A<<1;
  B = B>>1;
};

Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	R3 ← R3 – R3
S4	if R3 > 31, go to S12
S5	R4 ← R1 & 1
S6	if R4 != 1, go to S8
S7	R2 ← R2 + R0
S8	R0 ← R0 << 1
S9	R1 ← R1 >> 1
S10	R3 ← R3 + 1
S11	if R0 == R0, go to S4
S12	output ← R2
S13	if R0 == R0, go to S0

R0 is A R1 is B R2 is R R3 is C



General purpose data path

Greatest common divisor

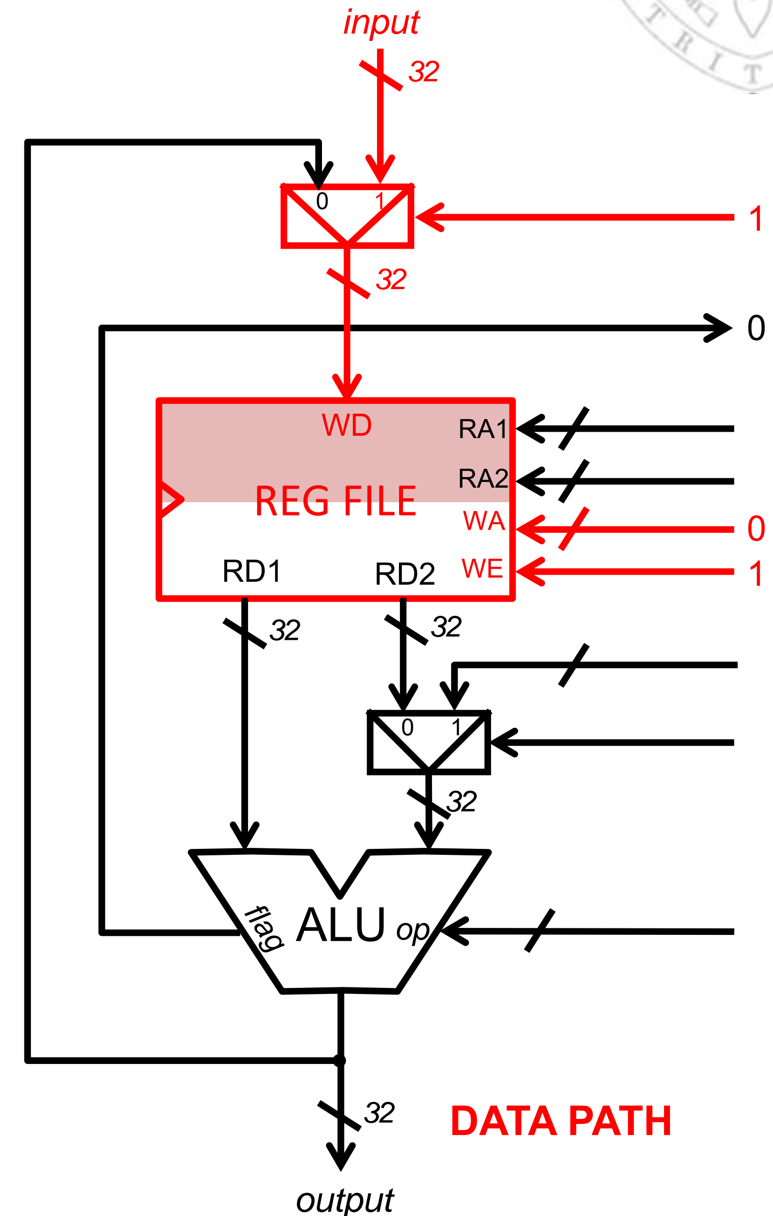
Algorithm to calculate the GCD of two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
if( A!=0 && B!=0 )
{
  while( A!=B )
    if( A>B )
      A = A-B;
    else
      B = B-A;

  R = A;
};
Rout = R;
```

Inputs and register transfers

S0	R0 ← input

R0 is A



General purpose data path

Greatest common divisor

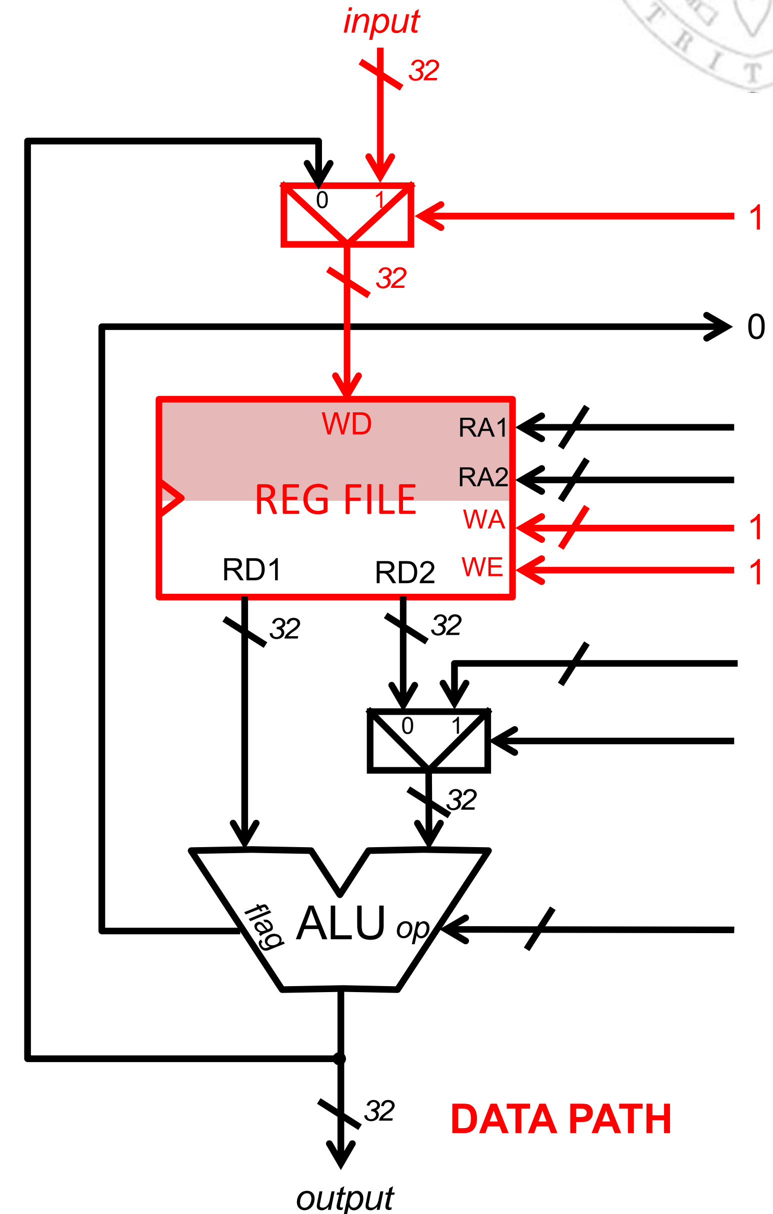
Algorithm to calculate the GCD of two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
if( A!=0 && B!=0 )
{
  while( A!=B )
    if( A>B )
      A = A-B;
    else
      B = B-A;

  R = A;
};
Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input

R0 is A   R1 is B



General purpose data path

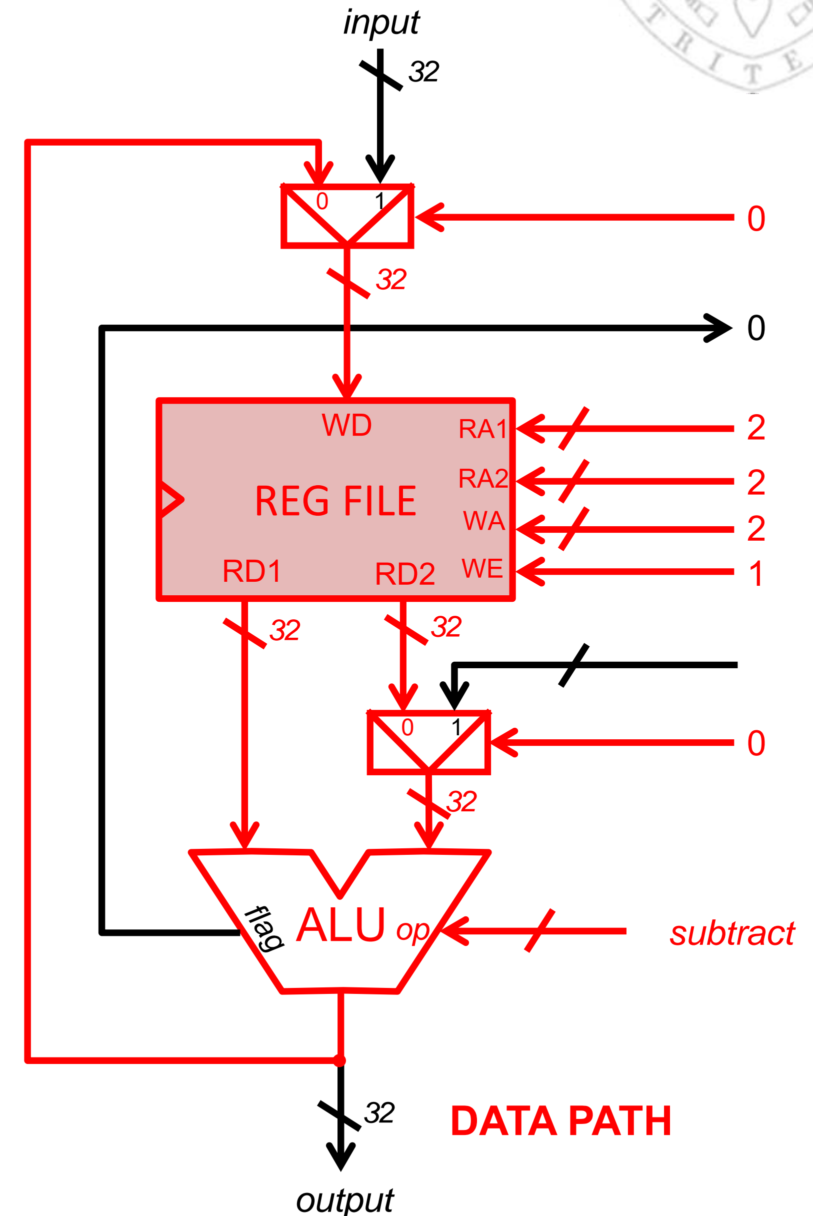
Greatest common divisor

Algorithm to calculate the GCD of two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
if( A!=0 && B!=0 )
{
  while( A!=B )
    if( A>B )
      A = A-B;
    else
      B = B-A;

  R = A;
};
Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2



R0 is A R1 is B R2 is R

General purpose data path

Greatest common divisor

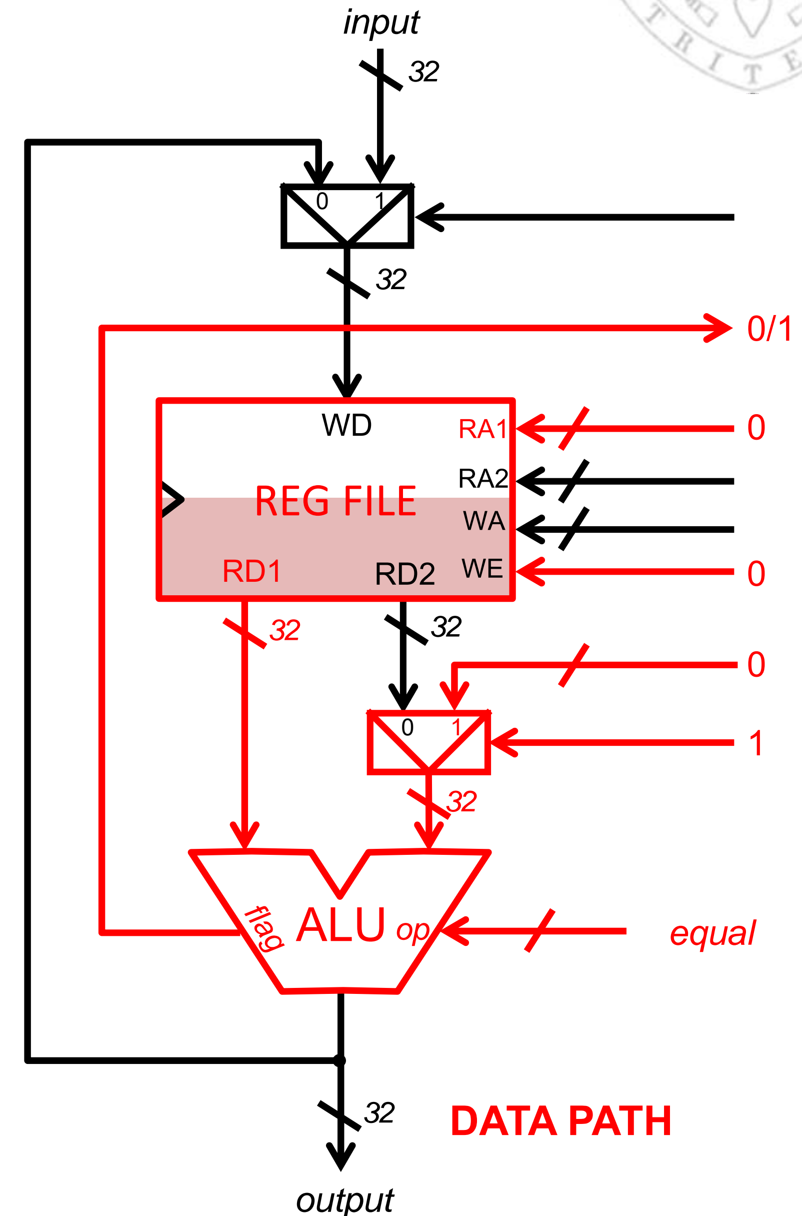
Algorithm to calculate the GCD of two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
if( A!=0 && B!=0 )
{
  while( A!=B )
    if( A>B )
      A = A-B;
    else
      B = B-A;

  R = A;
};
Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	if R0 == 0, go to S12

R0 is A   R1 is B   R2 is R



General purpose data path

Greatest common divisor

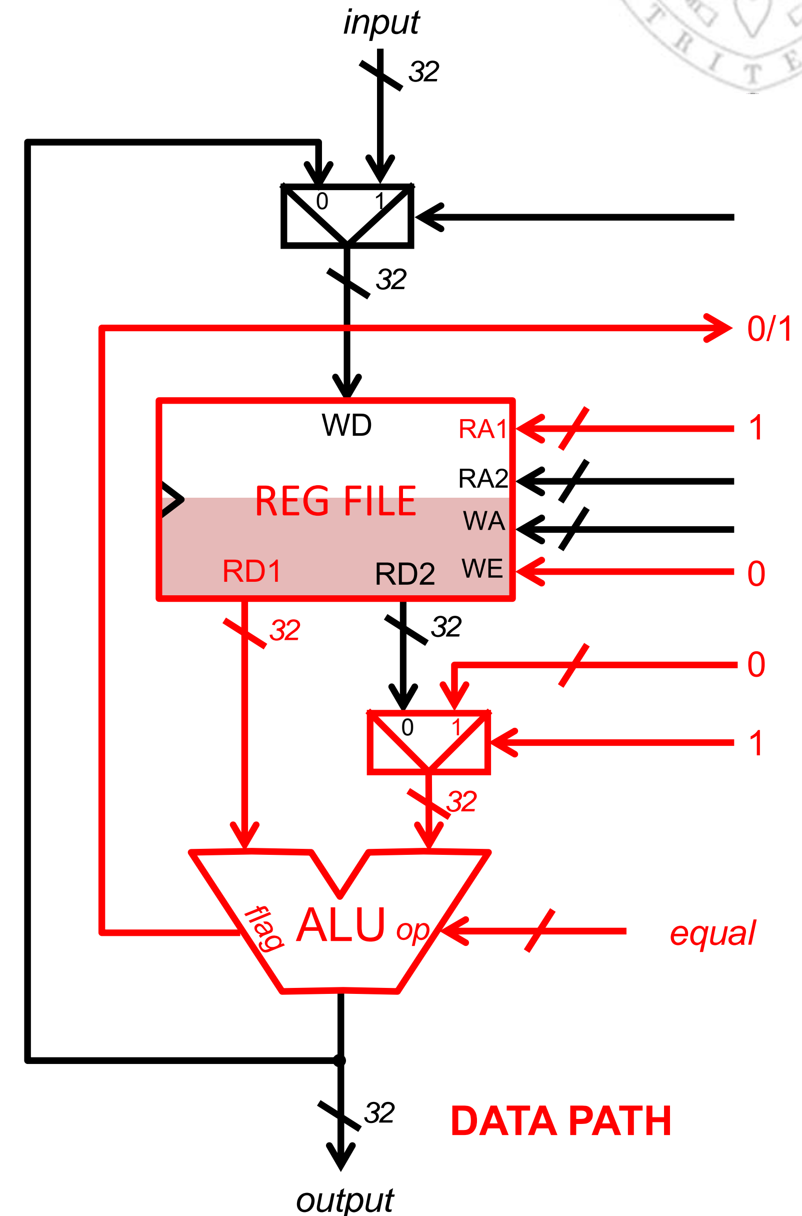
Algorithm to calculate the GCD of two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
if( A!=0 && B!=0 )
{
  while( A!=B )
    if( A>B )
      A = A-B;
    else
      B = B-A;

  R = A;
};
Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	if R0 == 0, go to S12
S4	if R1 == 0, go to S12

R0 is A R1 is B R2 is R



General purpose data path

Greatest common divisor

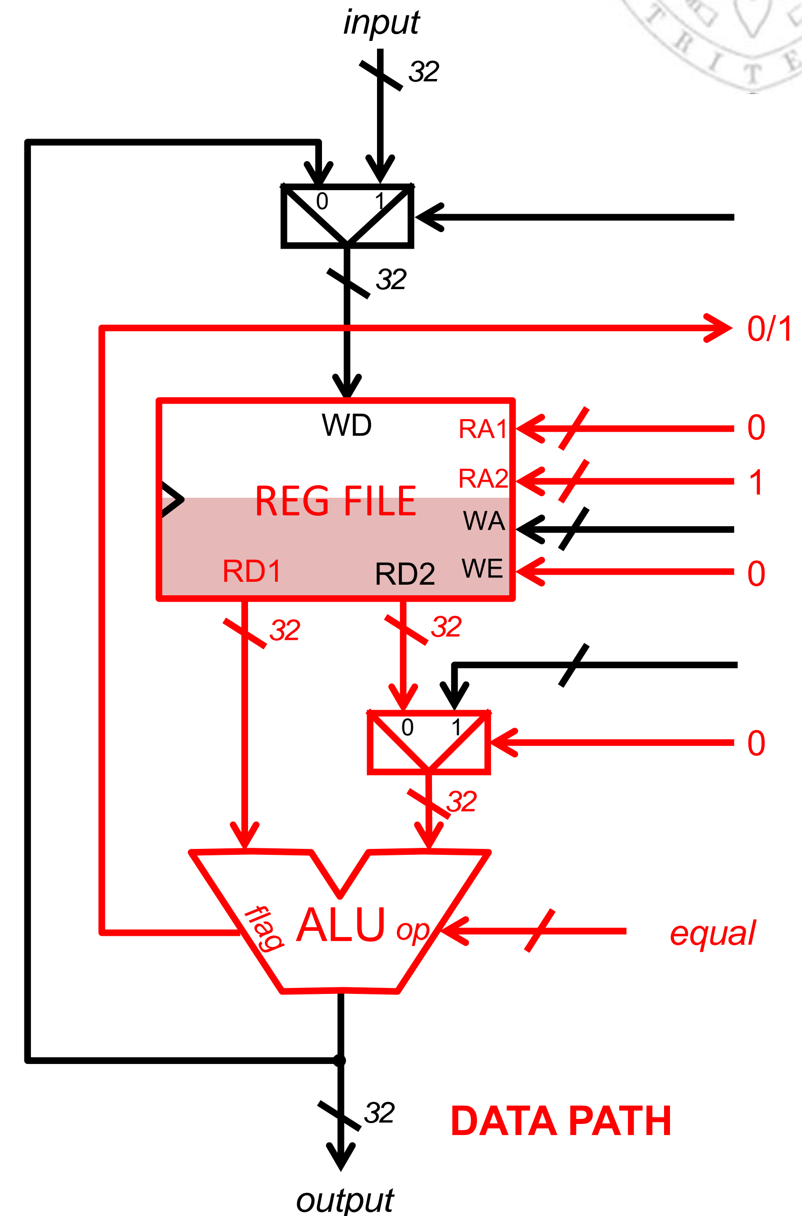
Algorithm to calculate the GCD of two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
if( A!=0 && B!=0 )
{
  while( A!=B )
    if( A>B )
      A = A-B;
    else
      B = B-A;

  R = A;
};
Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	if R0 == 0, go to S12
S4	if R1 == 0, go to S12
S5	if R0 == R1, go to S11

R0 is A R1 is B R2 is R



General purpose data path

Greatest common divisor

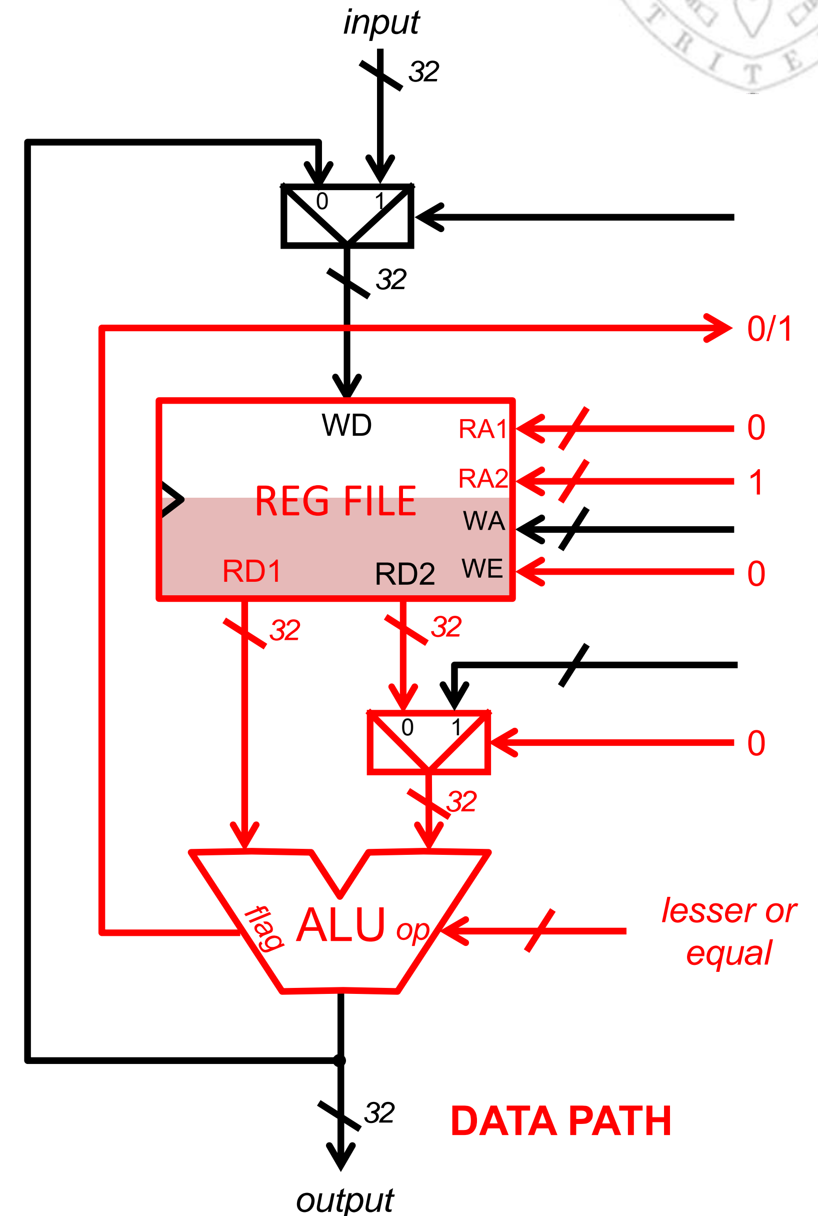
Algorithm to calculate the GCD of two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
if( A!=0 && B!=0 )
{
  while( A!=B )
    if( A>B )
      A = A-B;
    else
      B = B-A;

  R = A;
};
Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	if R0 == 0, go to S12
S4	if R1 == 0, go to S12
S5	if R0 == R1, go to S11
S6	if R0 <= R1, go to S9

R0 is A R1 is B R2 is R



General purpose data path

Greatest common divisor

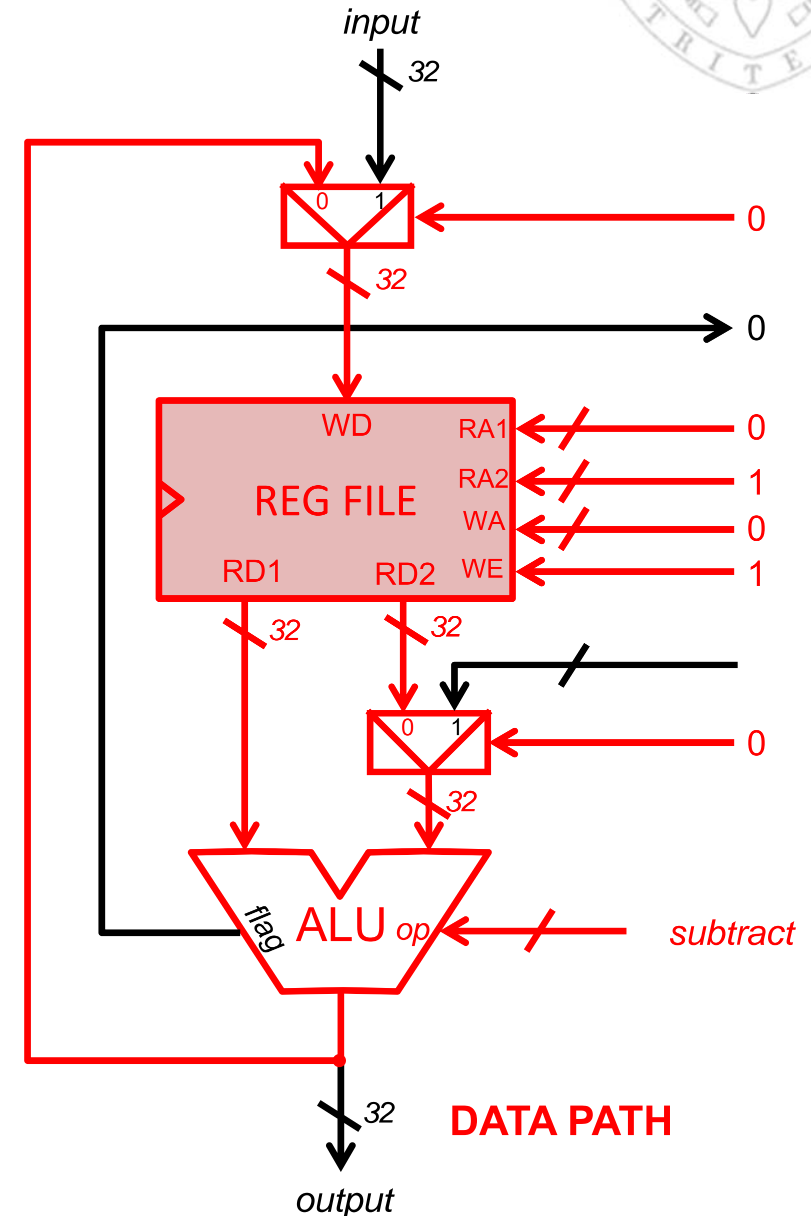
Algorithm to calculate the GCD of two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
if( A!=0 && B!=0 )
{
  while( A!=B )
    if( A>B )
      A = A-B;
    else
      B = B-A;

  R = A;
};
Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	if R0 == 0, go to S12
S4	if R1 == 0, go to S12
S5	if R0 == R1, go to S11
S6	if R0 <= R1, go to S9
S7	R0 ← R0 – R1

R0 is A R1 is B R2 is R



General purpose data path

Greatest common divisor

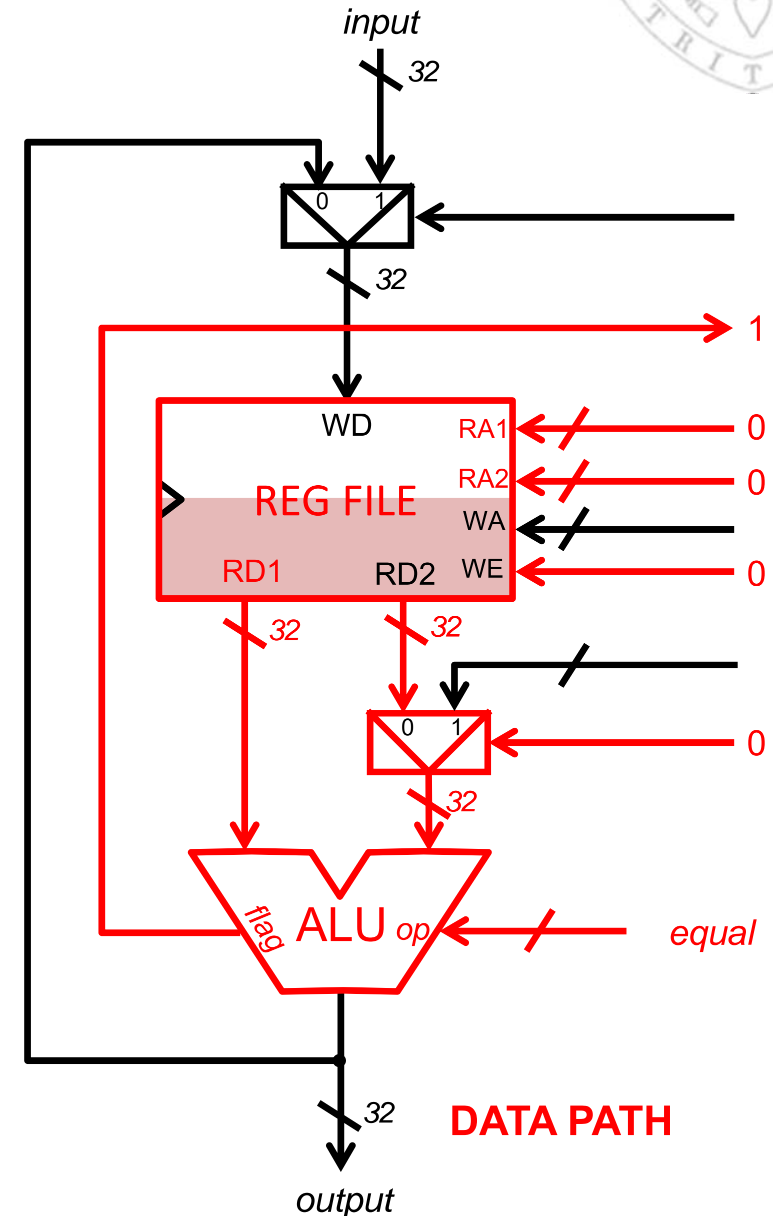
Algorithm to calculate the GCD of two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
if( A!=0 && B!=0 )
{
  while( A!=B )
    if( A>B )
      A = A-B;
    else
      A = B-A;

  R = A;
};
Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	if R0 == 0, go to S12
S4	if R1 == 0, go to S12
S5	if R0 == R1, go to S11
S6	if R0 <= R1, go to S9
S7	R0 ← R0 – R1
S8	if R0 == R0, go to S5

R0 is A　R1 is B　R2 is R



General purpose data path

Greatest common divisor

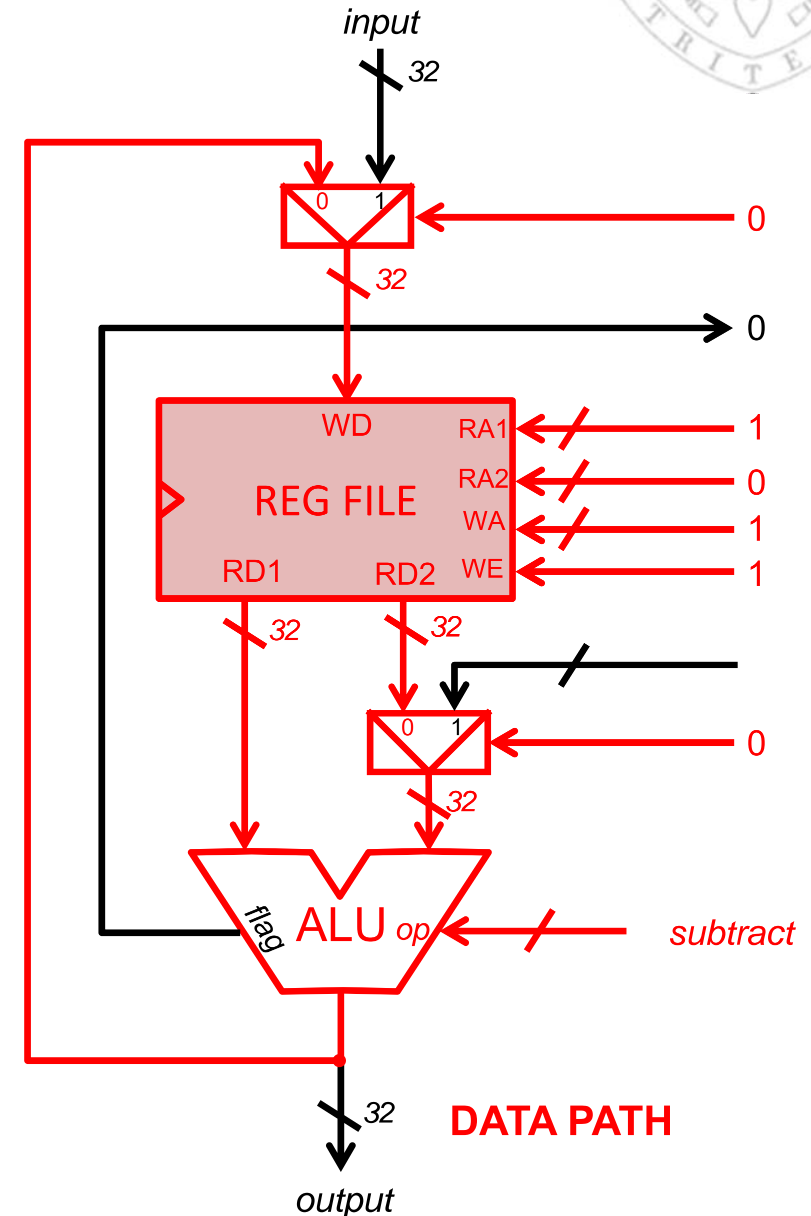
Algorithm to calculate the GCD of two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
if( A!=0 && B!=0 )
{
  while( A!=B )
    if( A>B )
      A = A-B;
    else
      B = B-A;

  R = A;
};
Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	if R0 == 0, go to S12
S4	if R1 == 0, go to S12
S5	if R0 == R1, go to S11
S6	if R0 <= R1, go to S9
S7	R0 ← R0 – R1
S8	if R0 == R0, go to S5
S9	R1 ← R1 – R0

R0 is A   R1 is B   R2 is R



General purpose data path

Greatest common divisor

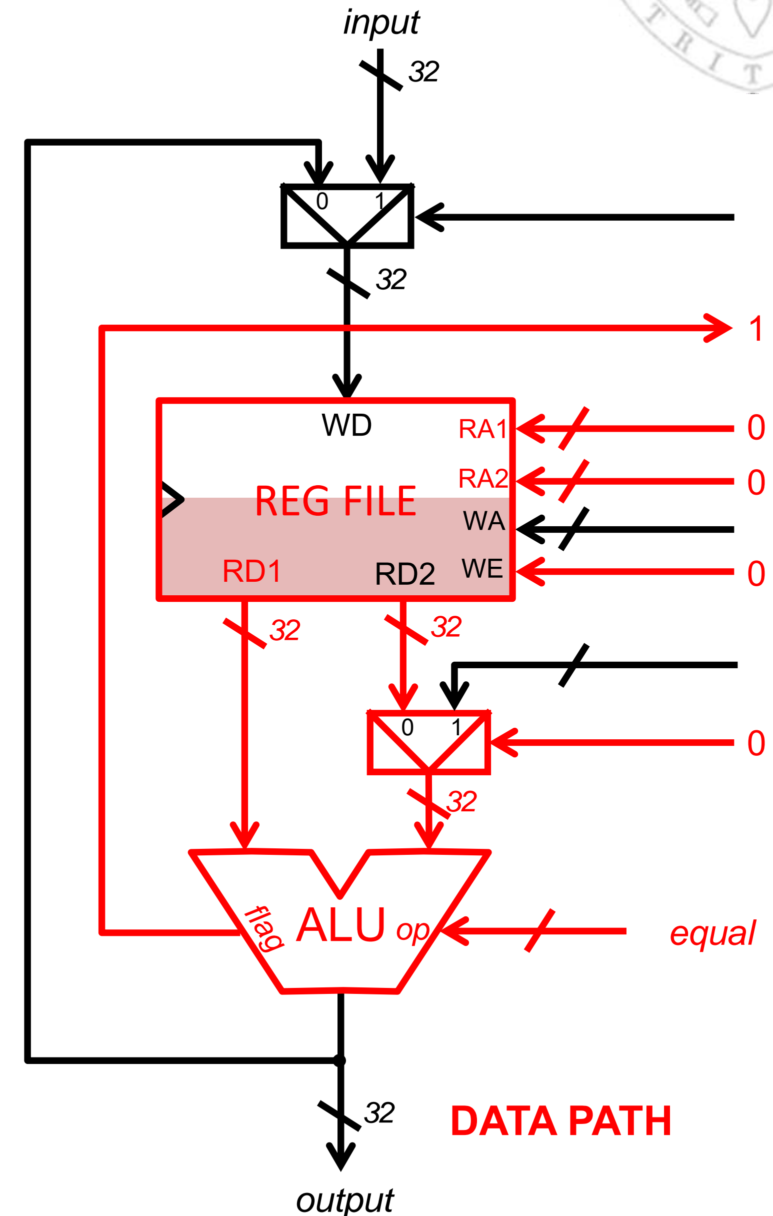
Algorithm to calculate the GCD of two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
if( A!=0 && B!=0 )
{
  while( A!=B )
    if( A>B )
      A = A-B;
    else
      B = B-A;

  R = A;
};
Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	if R0 == 0, go to S12
S4	if R1 == 0, go to S12
S5	if R0 == R1, go to S11
S6	if R0 <= R1, go to S9
S7	R0 ← R0 – R1
S8	if R0 == R0, go to S5
S9	R1 ← R1 – R0
S10	if R0 == R0, go to S5

R0 is A R1 is B R2 is R



General purpose data path

Greatest common divisor

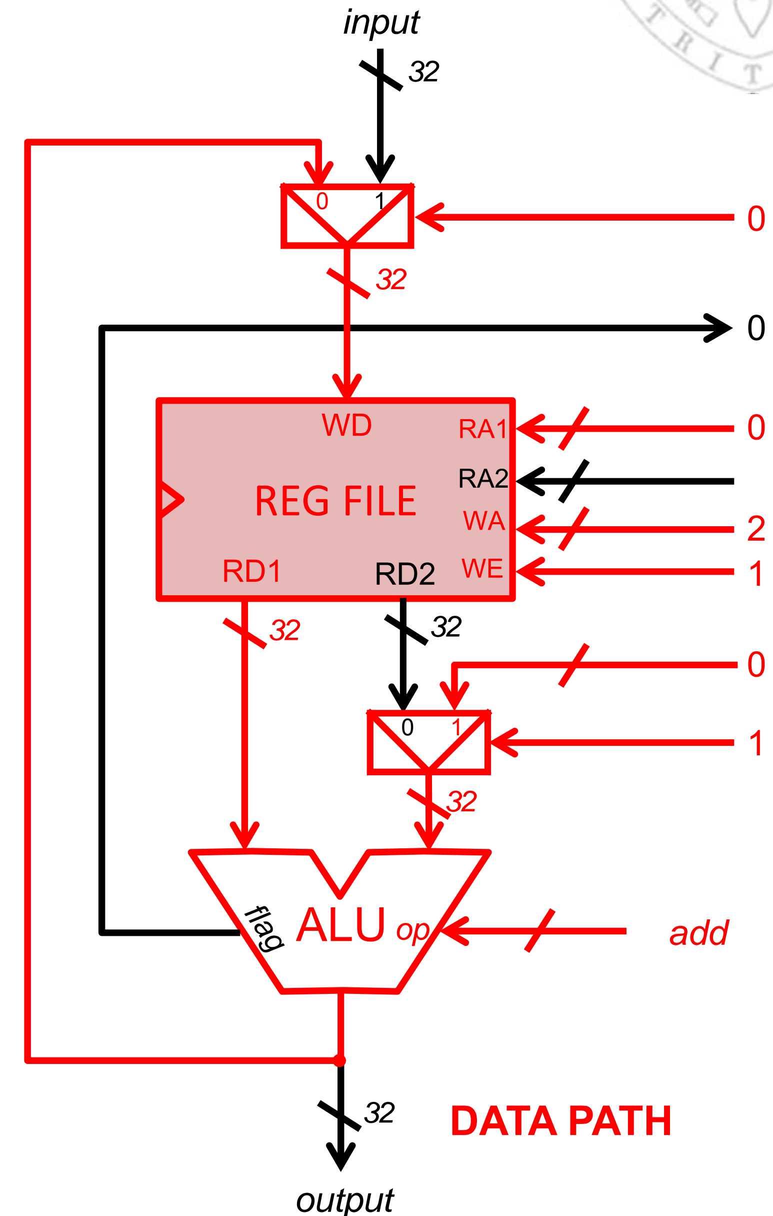
Algorithm to calculate the GCD of two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
if( A!=0 && B!=0 )
{
  while( A!=B )
    if( A>B )
      A = A-B;
    else
      B = B-A;

  R = A;
};
Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	if R0 == 0, go to S12
S4	if R1 == 0, go to S12
S5	if R0 == R1, go to S11
S6	if R0 <= R1, go to S9
S7	R0 ← R0 – R1
S8	if R0 == R0, go to S5
S9	R1 ← R1 – R0
S10	if R0 == R0, go to S5
S11	R2 ← R0 + 0

R0 is A R1 is B R2 is R



General purpose data path

Greatest common divisor

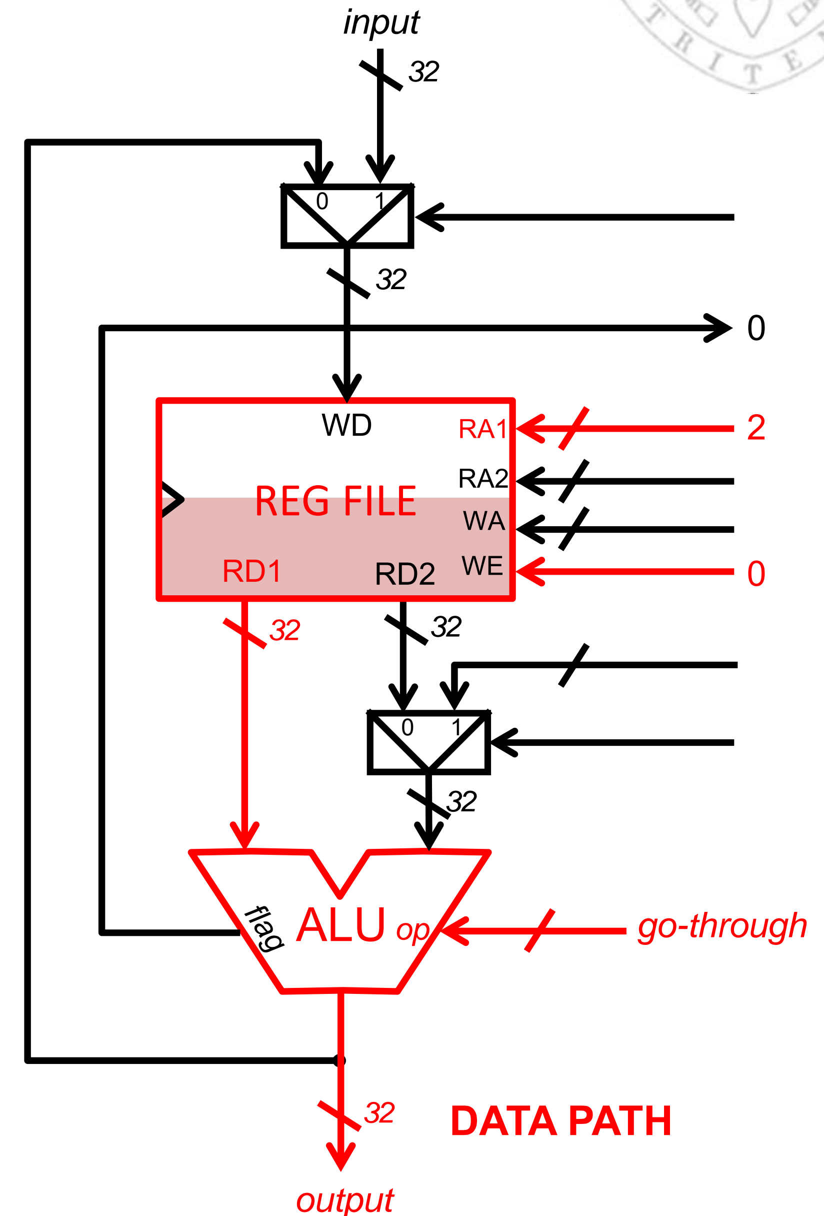
Algorithm to calculate the GCD of two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
if( A!=0 && B!=0 )
{
  while( A!=B )
    if( A>B )
      A = A-B;
    else
      B = B-A;

  R = A;
};
Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	if R0 == 0, go to S12
S4	if R1 == 0, go to S12
S5	if R0 == R1, go to S11
S6	if R0 <= R1, go to S9
S7	R0 ← R0 – R1
S8	if R0 == R0, go to S5
S9	R1 ← R1 – R0
S10	if R0 == R0, go to S5
S11	R2 ← R0 + 0
S12	output ← R2

R0 is A   R1 is B   R2 is R



General purpose data path

Greatest common divisor

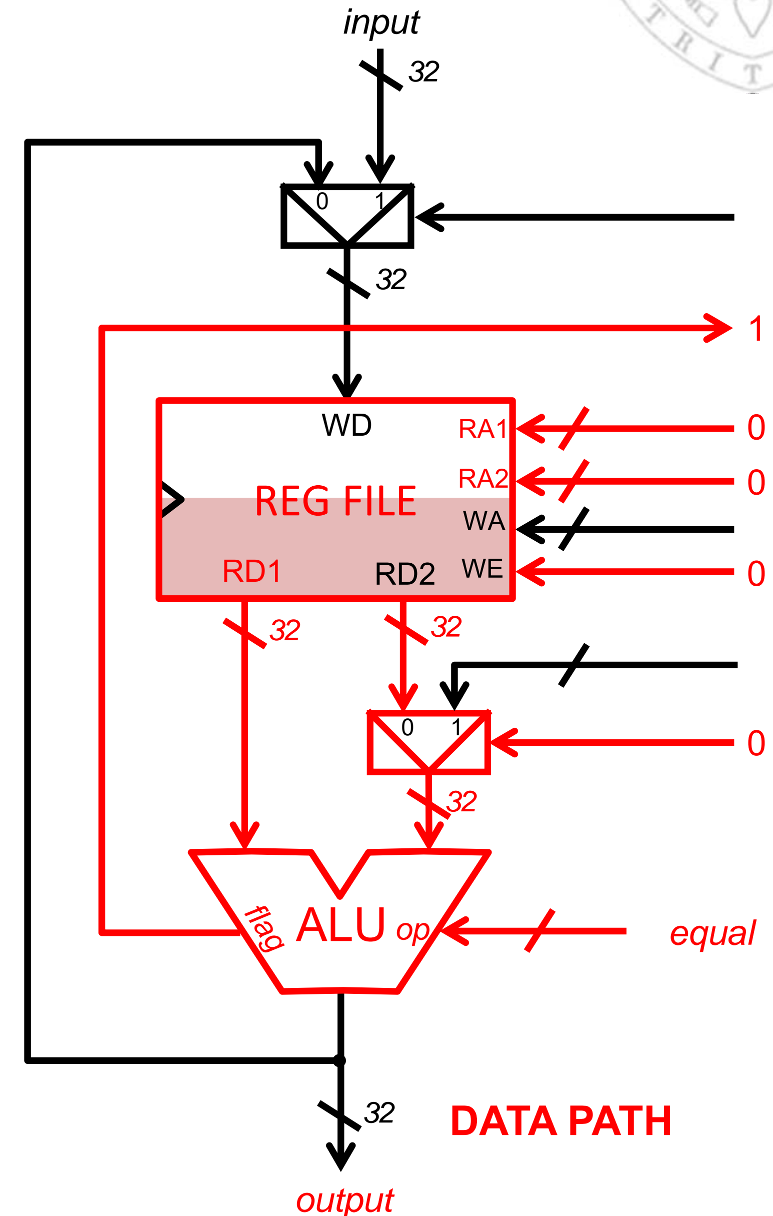
Algorithm to calculate the GCD of two 32-bit numbers

```
A = Ain;
B = Bin;
R = 0;
if( A!=0 && B!=0 )
{
  while( A!=B )
    if( A>B )
      A = A-B;
    else
      B = B-A;

  R = A;
};
Rout = R;
```

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	if R0 == 0, go to S12
S4	if R1 == 0, go to S12
S5	if R0 == R1, go to S11
S6	if R0 <= R1, go to S9
S7	R0 ← R0 – R1
S8	if R0 == R0, go to S5
S9	R1 ← R1 – R0
S10	if R0 == R0, go to S5
S11	R2 ← R0 + 0
S12	output ← R2
S13	if R0 == R0, go to S0

R0 is A   R1 is B   R2 is R



ROM-implemented controller

Multiplication

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	R3 ← R3 – R3
S4	if R3 > 31, go to S12
S5	R4 ← R1 & 1
S6	if R4 != 1, go to S8
S7	R2 ← R2 + R0
S8	R0 ← R0 << 1
S9	R1 ← R1 >> 1
S10	R3 ← R3 + 1
S11	if R0 == R0, go to S4
S12	output ← R2
S13	if R0 == R0, go to S0

ROM content (not encoded)

addr	op	rd	rs1	rs2	const	offset	next addr.
0	*load*	0	–	–	–	–	*addr+1*
1	*load*	1	–	–	–	–	*addr+1*
2	*subtract*	2	2	2	–	–	*addr+1*
3	*subtract*	3	3	3	–	–	*addr+1*
4	*branch if > const*	–	3	–	31	12	*addr+1* or 12
5	*AND const*	4	1	–	1	–	*addr+1*
6	*branch if diff const*	–	4	–	1	8	*addr+1* or 8
7	*add*	2	2	0	–	–	*addr+1*
8	*left shift const*	0	0	–	1	–	*addr+1*
9	*right shift const*	1	1	–	1	–	*addr+1*
10	*add const*	3	3	–	1	–	*addr+1*
11	*branch if equal*	–	0	0	–	4	4
12	*store*	–	2	–	–	–	*addr+1*
13	*branch if equal*	–	0	0	–	0	0

ROM-implemented controller

Greatest common divisor

Inputs and register transfers

S0	R0 ← input
S1	R1 ← input
S2	R2 ← R2 – R2
S3	if R0 == 0, go to S12
S4	if R1 == 0, go to S12
S5	if R0 == R1, go to S11
S6	if R0 <= R1, go to S9
S7	R0 ← R0 – R1
S8	if R0 == R0, go to S5
S9	R1 ← R1 – R0
S10	if R0 == R0, go to S5
S11	R2 ← R0 + 0
S12	output ← R2
S13	if R0 == R0, go to S0

ROM content (not encoded)

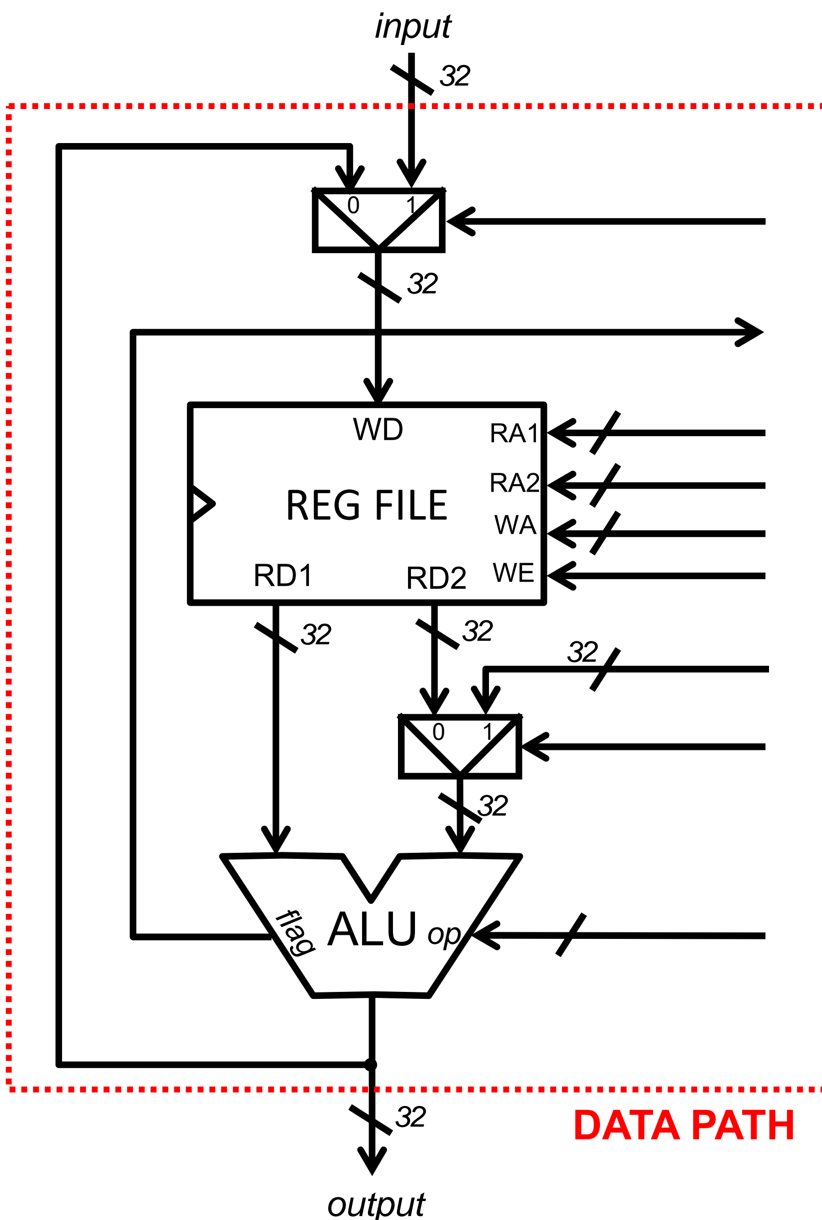
addr	op	rd	rs1	rs2	const	offset	next addr.
0	*load*	0	–	–	–	–	*addr+1*
1	*load*	1	–	–	–	–	*addr+1*
2	*subtract*	2	2	2	–	–	*addr+1*
3	*branch if > const*	–	0	–	0	12	*addr+1* or 12
4	*branch if > const*	–	1	–	0	12	*addr+1* or 12
5	*branch if equal*	–	0	1	–	11	*addr+1* or 11
6	*branch if <=*	–	0	1	–	9	*addr+1* or 9
7	*subtract*	0	0	1	–	–	*addr+1*
8	*branch if equal*	–	0	0	–	5	5
9	*subtract*	1	1	0	–	–	*daddrir+1*
10	*branch if equal*	–	0	0	–	5	5
11	*add const*	2	0	–	0	–	*addr+1*
12	*store*	–	2	–	–	–	*addr+1*
13	*branch if equal*	–	0	0	–	0	0

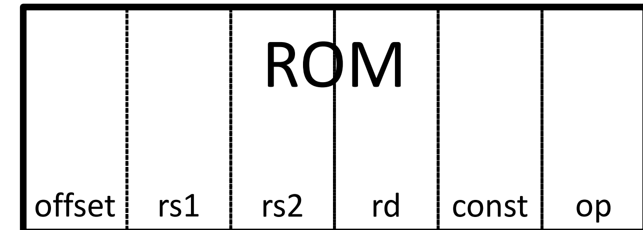
Generic data path + controller

input
32
0 1
32
WD
RA1
RA2
WA
WE
REG FILE
RD1
RD2
32
32
32
0 1
32
flag
ALU
op
32
output
DATA PATH

ROM
CONTROLLER

Generic data path + controller





DATA PATH

CONTROLLER

Generic data path + controller

Generic data path + controller

Generic data path + controller

Generic data path + controller

Generic data path + controller

Generic data path + controller

Generic data path + controller

input
32
0 1
32
WD
RA1
RA2
WA
WE
REG FILE
RD1
RD2
32
32
32
0 1
32
ALU
flag
op
32
output
DATA PATH
ROM
offset
rs1
rs2
rd
const
op
DEC
CONTROLLER

Generic data path + controller

input
32
0 1
32
WD
RA1
RA2
WA
WE
REG FILE
RD1
RD2
32
32
32
0 1
32
ALU
flag
op
32
output
DATA PATH

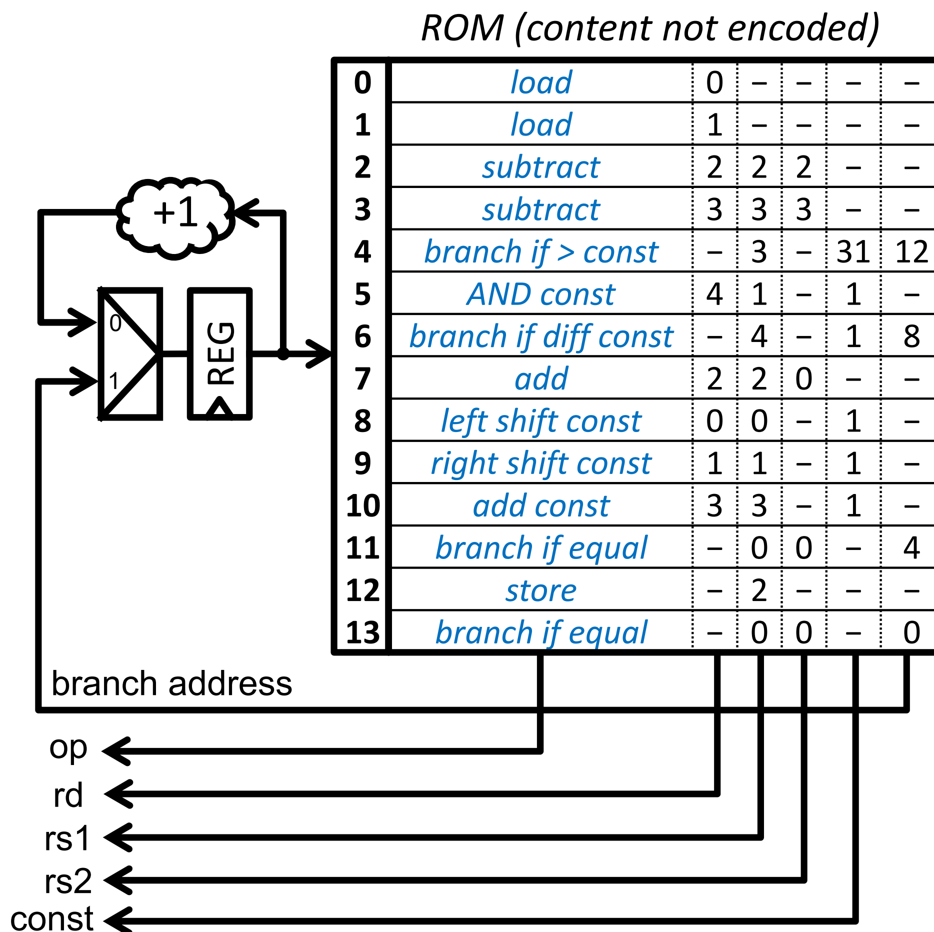
ROM
offset rs1 rs2 rd const op
DEC
CONTROLLER

Generic data path + controller

Generic data path + controller

Generic data path + controller

- In this controller, many elements that are common to a generic processor can be found:



ROM (content not encoded)

0	*load*	0	–	–	–	–
1	*load*	1	–	–	–	–
2	*subtract*	2	2	2	–	–
3	*subtract*	3	3	3	–	–
4	*branch if > const*	–	3	–	31	12
5	*AND const*	4	1	–	1	–
6	*branch if diff const*	–	4	–	1	8
7	*add*	2	2	0	–	–
8	*left shift const*	0	0	–	1	–
9	*right shift const*	1	1	–	1	–
10	*add const*	3	3	–	1	–
11	*branch if equal*	–	0	0	–	4
12	*store*	–	2	–	–	–
13	*branch if equal*	–	0	0	–	0

Generic data path + controller

- In this controller, many elements that are common to a generic processor can be found:

ROM (content not encoded)

0	*load*	0	–	–	–	–
1	*load*	1	–	–	–	–
2	*subtract*	2	2	2	–	–
3	*subtract*	3	3	3	–	–
4	*branch if > const*	–	3	–	31	12
5	*AND const*	4	1	–	1	–
6	*branch if diff const*	–	4	–	1	8
7	*add*	2	2	0	–	–
8	*left shift const*	0	0	–	1	–
9	*right shift const*	1	1	–	1	–
10	*add const*	3	3	–	1	–
11	*branch if equal*	–	0	0	–	4
12	*store*	–	2	–	–	–
13	*branch if equal*	–	0	0	–	0

+1
0
1
REG
branch address
op
rd
rs1
rs2
const

The ROM stores instructions: simplest orders that can be executed in the data path

Generic data path + controller

- In this controller, many elements that are common to a generic processor can be found:

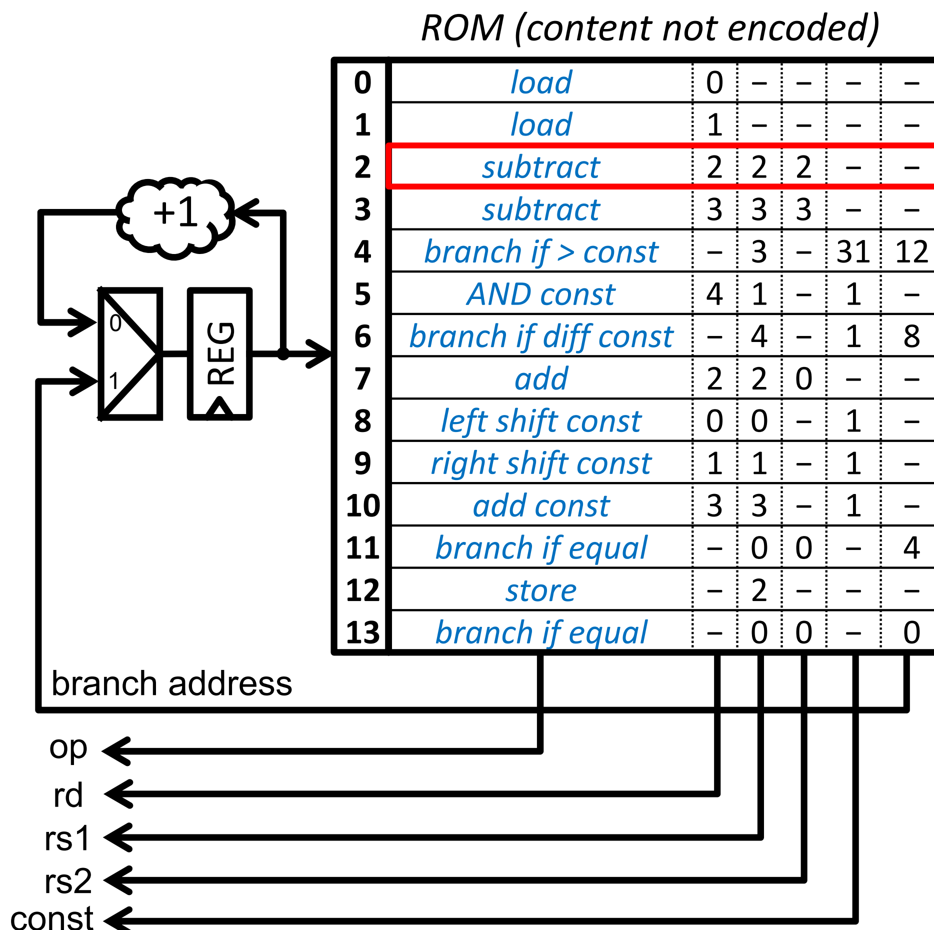
ROM (content not encoded)

0	load	0	–	–	–	–
1	load	1	–	–	–	–
2	subtract	2	2	2	–	–
3	subtract	3	3	3	–	–
4	branch if > const	–	3	–	31	12
5	AND const	4	1	–	1	–
6	branch if diff const	–	4	–	1	8
7	add	2	2	0	–	–
8	left shift const	0	0	–	1	–
9	right shift const	1	1	–	1	–
10	add const	3	3	–	1	–
11	branch if equal	–	0	0	–	4
12	store	–	2	–	–	–
13	branch if equal	–	0	0	–	0

Each instruction defines the operation to perform, the source operands to use and the destination of the result

Generic data path + controller

- In this controller, many elements that are common to a generic processor can be found:



0	*load*	0	–	–	–	–
1	*load*	1	–	–	–	–
2	*subtract*	2	2	2	–	–
3	*subtract*	3	3	3	–	–
4	*branch if > const*	–	3	–	31	12
5	*AND const*	4	1	–	1	–
6	*branch if diff const*	–	4	–	1	8
7	*add*	2	2	0	–	–
8	*left shift const*	0	0	–	1	–
9	*right shift const*	1	1	–	1	–
10	*add const*	3	3	–	1	–
11	*branch if equal*	–	0	0	–	4
12	*store*	–	2	–	–	–
13	*branch if equal*	–	0	0	–	0

The instructions are stored encoded in the ROM (machine code), but it is convenient to use a symbolic representation (assembly)

2 1001011 0010 0010 0010 0000 0000

2 *subtract* R2, R2, R2

Generic data path + controller

- In this controller, many elements that are common to a generic processor can be found:

ROM (content not encoded)

0	load	0	–	–	–	–
1	load	1	–	–	–	–
2	subtract	2	2	2	–	–
3	subtract	3	3	3	–	–
4	branch if > const	–	3	–	31	12
5	AND const	4	1	–	1	–
6	branch if diff const	–	4	–	1	8
7	add	2	2	0	–	–
8	left shift const	0	0	–	1	–
9	right shift const	1	1	–	1	–
10	add const	3	3	–	1	–
11	branch if equal	–	0	0	–	4
12	store	–	2	–	–	–
13	branch if equal	–	0	0	–	0

+1

0

1

REG

branch address

op

rd

rs1

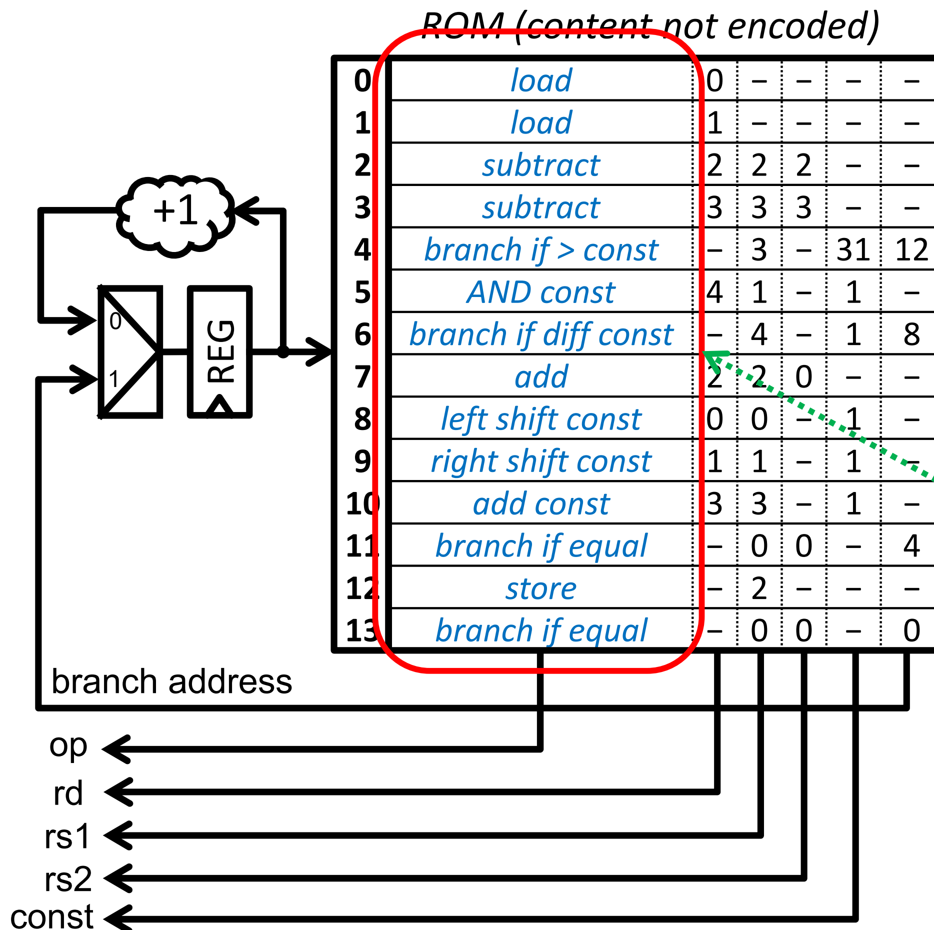
rs2

const

There are different types of instructions: arithmetic, branch, data transfer...

Generic data path + controller

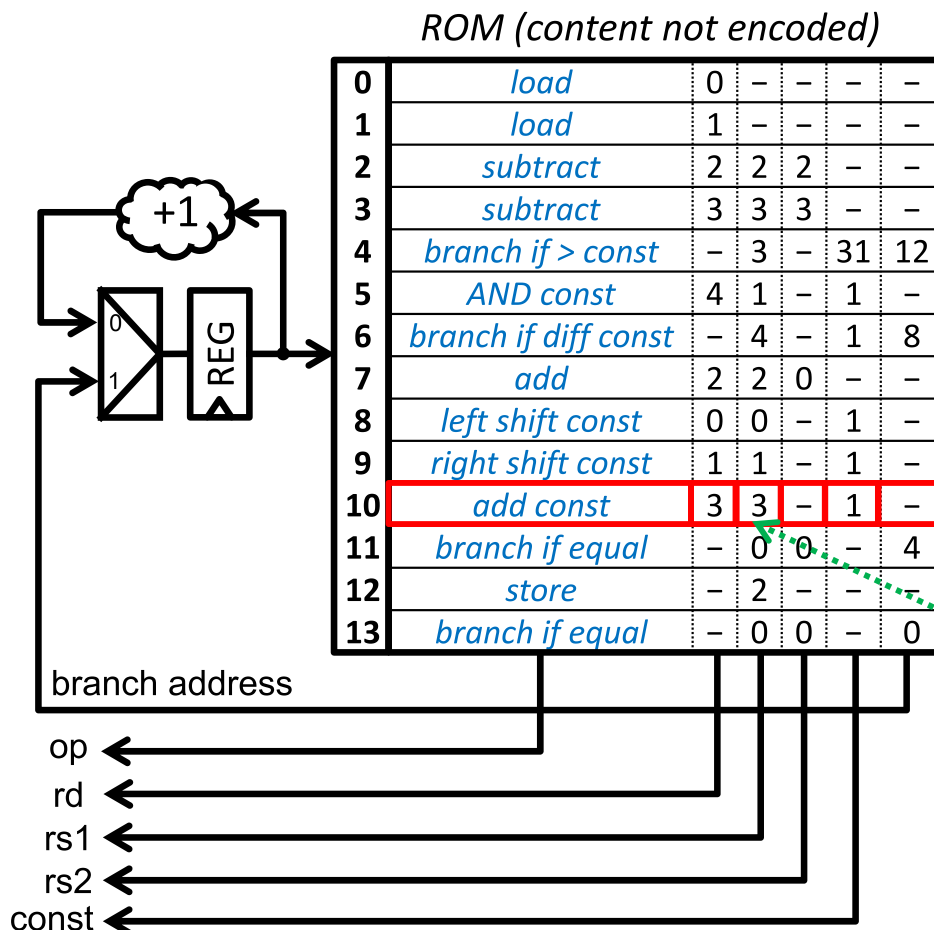
- In this controller, many elements that are common to a generic processor can be found:



All the different instructions that can be executed form the instruction set

Generic data path + controller

- In this controller, many elements that are common to a generic processor can be found:

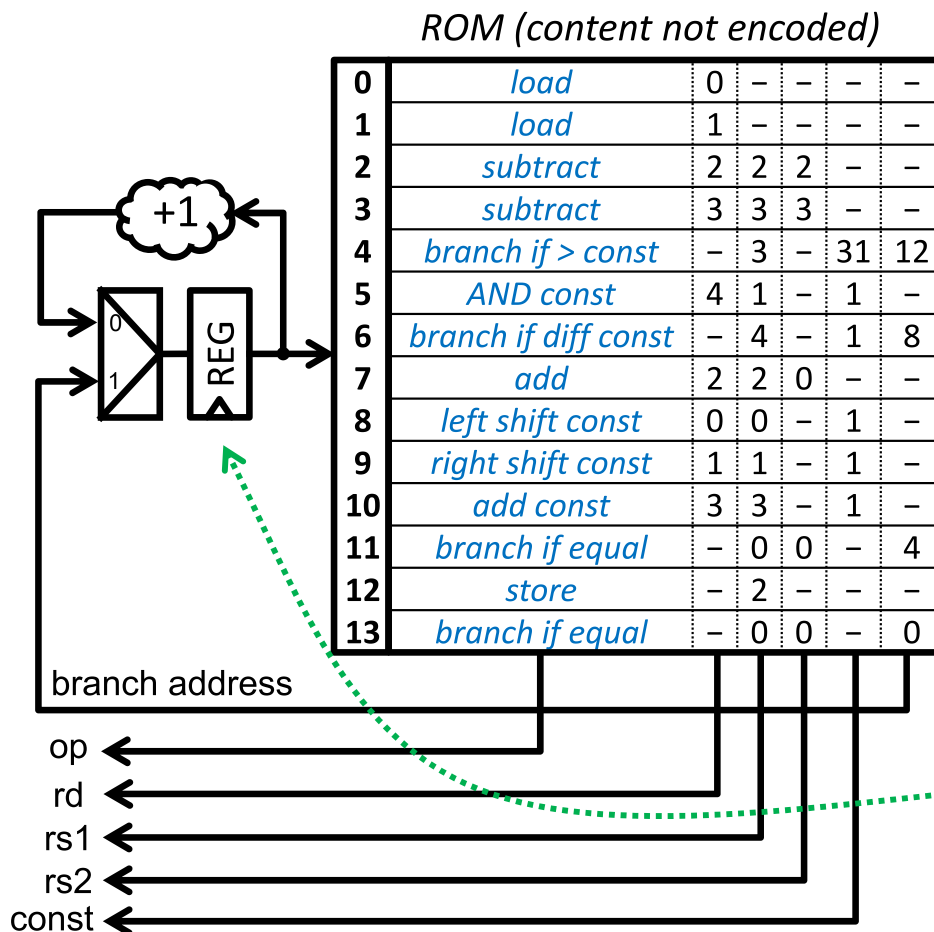


0	*load*	0	–	–	–	–
1	*load*	1	–	–	–	–
2	*subtract*	2	2	2	–	–
3	*subtract*	3	3	3	–	–
4	*branch if > const*	–	3	–	31	12
5	*AND const*	4	1	–	1	–
6	*branch if diff const*	–	4	–	1	8
7	*add*	2	2	0	–	–
8	*left shift const*	0	0	–	1	–
9	*right shift const*	1	1	–	1	–
10	*add const*	3	3	–	1	–
11	*branch if equal*	–	0	0	–	4
12	*store*	–	2	–	–	–
13	*branch if equal*	–	0	0	–	0

Instructions have a common format that places and encodes the information fields

Generic data path + controller

- In this controller, many elements that are common to a generic processor can be found:



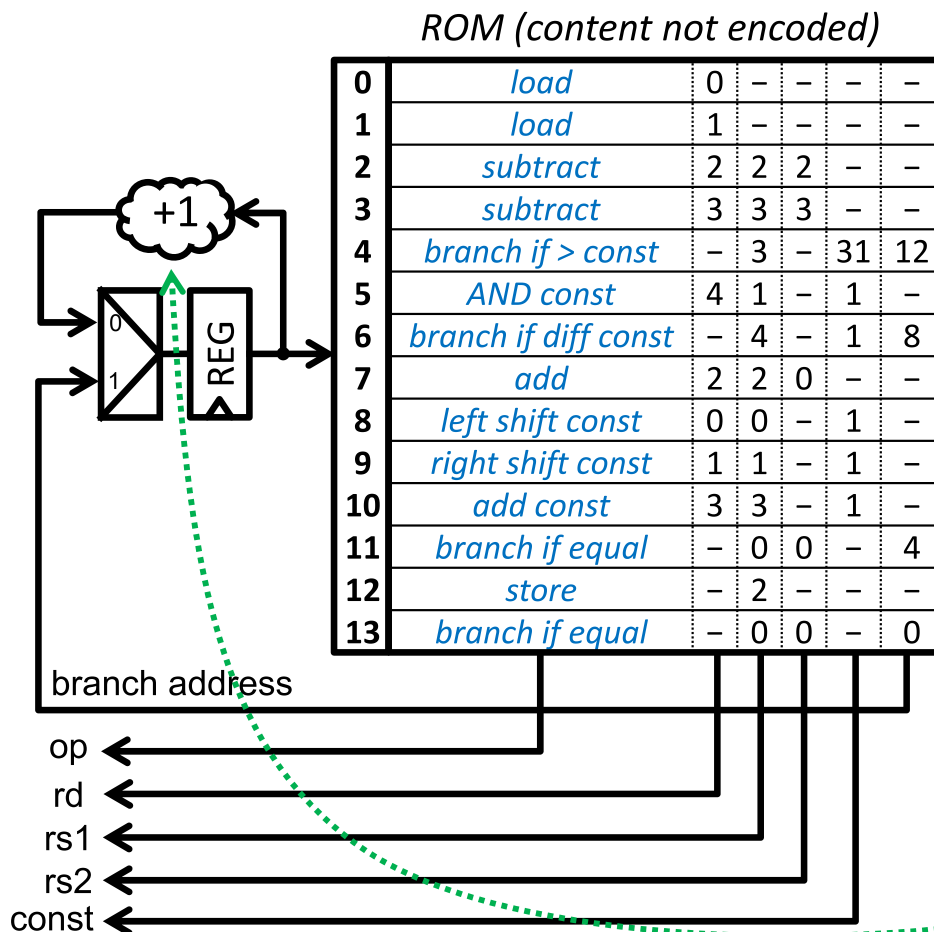
ROM (content not encoded)

0	*load*	0	–	–	–	–
1	*load*	1	–	–	–	–
2	*subtract*	2	2	2	–	–
3	*subtract*	3	3	3	–	–
4	*branch if > const*	–	3	–	31	12
5	*AND const*	4	1	–	1	–
6	*branch if diff const*	–	4	–	1	8
7	*add*	2	2	0	–	–
8	*left shift const*	0	0	–	1	–
9	*right shift const*	1	1	–	1	–
10	*add const*	3	3	–	1	–
11	*branch if equal*	–	0	0	–	4
12	*store*	–	2	–	–	–
13	*branch if equal*	–	0	0	–	0

There is a "program counter (PC)" that addresses the memory

Generic data path + controller

- In this controller, many elements that are common to a generic processor can be found:



0	*load*	0	–	–	–	–
1	*load*	1	–	–	–	–
2	*subtract*	2	2	2	–	–
3	*subtract*	3	3	3	–	–
4	*branch if > const*	–	3	–	31	12
5	*AND const*	4	1	–	1	–
6	*branch if diff const*	–	4	–	1	8
7	*add*	2	2	0	–	–
8	*left shift const*	0	0	–	1	–
9	*right shift const*	1	1	–	1	–
10	*add const*	3	3	–	1	–
11	*branch if equal*	–	0	0	–	4
12	*store*	–	2	–	–	–
13	*branch if equal*	–	0	0	–	0

Usually, instructions are executed sequentially in the order in which they are stored

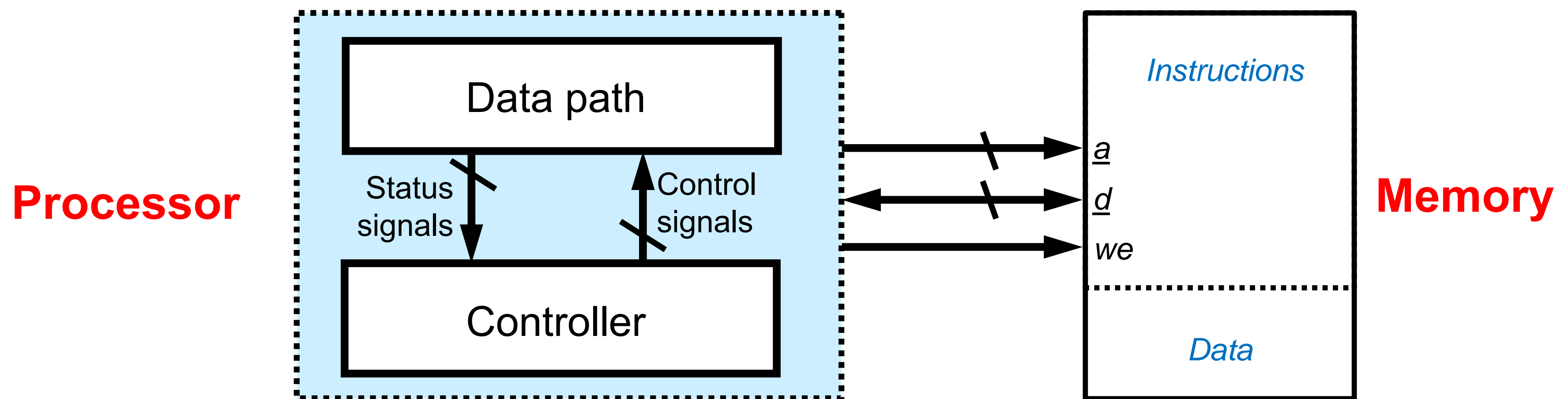
Generic data path + controller

- Just by modifying the program stored in the ROM, the data path performs a different algorithm.
  - The ENIAC (1946) was programmed by rewiring the computer

source: https://alltogether.swe.org

General purpose circuit: computer

- It is possible to design a circuit even more generic to avoid redesigning it each time the algorithm that is performed is changed:
  - Replacing the ROM with a RAM to facilitate the change of program.
  - Replacing the external input/output of the data path with a connection to the RAM to read/write data stored in it.



- New elements appear:
  - SW: Set of programs executed by the circuit
  - Programmer: person who develops software programs
  - Compiler: translator of algorithms into sequences of instructions

Von Neumann model (1945)

Principles

- Computer with programs stored in memory.
  - A program is a sequence of instructions and data.
    - Instructions: they rule the behavior of the computer.
    - Data: they are processed by the instructions.
- The memory is formed by words that are linearly organized.
  - All of them have the same size.
  - Each one is identified by the address that occupies in the memory.
  - It contains encoded instructions and data with no distinction.
- The program instructions are executed sequentially:
  - i.e., in the same order in which they are stored in the memory.
  - This order can only be changed after executing a branch instruction.
  - There is a program counter (PC) register that stores the address memory occupied by the instruction to execute.

Von Neumann model (1945)

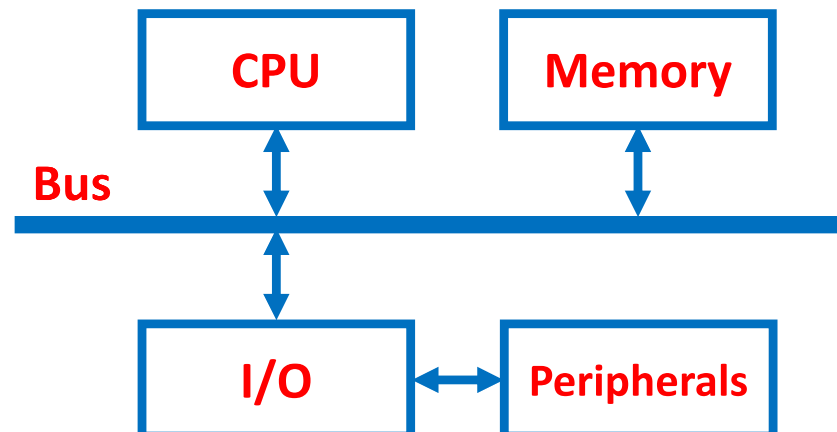
Principles

- The execution of an instruction implies:
  - Fetch the instruction from memory, whose address is contained in the PC.
  - Decode the instruction.
  - Read the source operands indicated in the instruction.
  - Perform the operation indicated in the instruction on the read operands.
  - Write back the result of the operation in the target destination indicated in the instruction.
  - Update the PC with the address of the next instruction to execute
    - By default, this would be the PC + the size of the instruction that is being executed.
    - If the instruction is a branch, the target address will be indicated by the instruction.
- This sequence of stages is known as instruction cycle.
  - A processor performs successive instruction cycles, indefinitely, one after the other.

Von Neumann model (1945)

Organization

- The main components are:
  - Processor (CPU): controls the behavior of the computer and processes data according to the instructions of a stored program
  - Memory subsystem: stores data/instructions (program)
  - Input/output (I/O) subsystem: transfers data between the computer and the external environment
  - Interconnection subsystem: provides communication channels among the processor, the memory and the I/O.



Basic concepts

Processor architecture

- The processor architecture or instruction set architecture (ISA) is the set of processor attributes that are visible by:
  - The assembly language programmer.
  - The high-level language compiler.
- It implies an agreement between HW and SW that comprises the following elements:
  - Data types supported by the instructions.
  - Memory model and organization of the information in memory.
  - Processor registers accessible by the programmer.
  - Mechanism to indicate the operand location of an instruction.
  - Set of instructions that can be executed by the processor.
  - Format and encoding of the machine instruction.
- The processor architecture abstracts the complexity of the HW design indicating what the processor does, without stating how.

Basic concepts

Processor organization

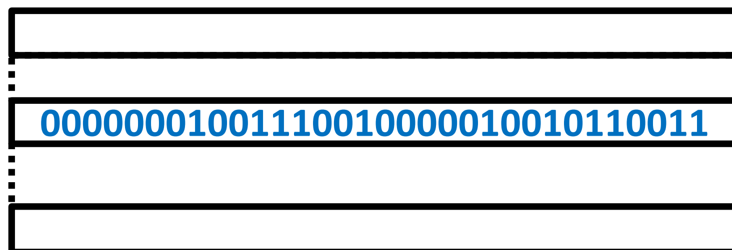
- The processor organization or microarchitecture is the organization of the HW components that form the architecture.
  - The same architecture can be implemented with different microarchitectures, which can be designed by different manufacturers.
- A family is the set of processors with the same architecture but different implementations:
  - Different technology, different performance, different price...
- There is a large number of different architecture families:
  - x86, ARM, MIPS, SPARC, PowerPC, RISC-V...
  - All the processor of the same family are compatible and can execute exactly the same programs.
  - Retrocompatibility allows the newer family members execute programs developed for the older ones.

Basic concepts

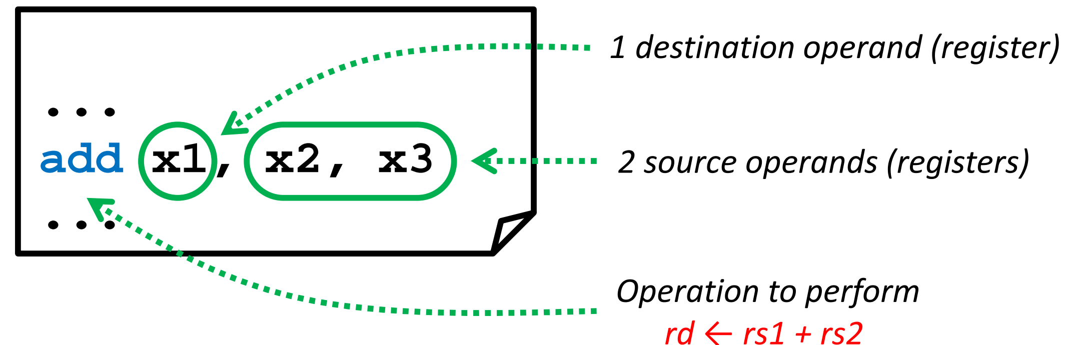
Machine code vs. assembly code

- The simplest order that a processor can execute is called instruction.
  - It defines the operation to perform and its operands.
  - The hardware of a computer only interprets and executes binary encoded instructions, i.e. machine language.
  - The assembly language is a human-readable representation of the machine language.
    - The assembly instructions use mnemonics to indicate the operation and the operands.
    - There usually is a 1:1 relation between machine and assembly instructions.

RISC-V machine code



RISC-V assembly



Basic concepts

Compiler vs. assembler

- Assembler: software that translates assembly instructions into machine code.

- Compiler: software that translates a high-level program (i.e. C/C++) into an assembly program.
  - Generally, the relation between high level and assembly instructions is 1:n.

C/C++ language

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

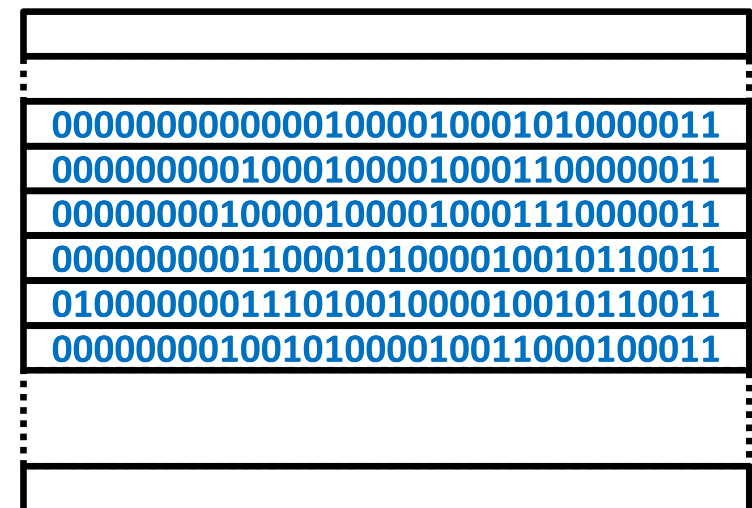
a → x5 c → x7
b → x6 d → x9

Relation of C variables with RISC-V registers

RISC-V assembly

```
...
lw   x5, 0(x8)
lw   x6, 4(x8)
lw   x7, 8(x8)
add  x9, x5, x6
sub  x9, x9, x7
sw   x9, 12(x8)
...
```

RISC-V machine code



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/