ECSE324 : Computer Organization

Processor Design

Textbook§Chapter 5

Christophe Dubach
Fall 2023

Revision history:
Warren Gross – 2017
Christophe Dubach – W2020, F2020, F2021, F2022, F2023
Brett H. Meyer – W2021, W2022, W2023
Some material from Hamacher, Vranesic, Zaky, and Manjikian, *Computer Organization and Embedded Systems, 6 th ed*, 2012, McGraw Hill and Patterson and Hennessy, *Computer Organization and Design, ARM Edition*, Morgan Kaufmann, 2017, and notes by A. Moshovos

Timestamp: 2023/09/25 17:07:00

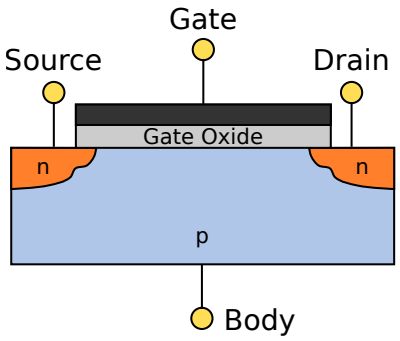
Disclaimer

It is possible (and even likely) that I will (sometimes) make mistakes and give incorrect information during the live lectures. If you have any doubts, please check the textbook, or ask for clarification online.

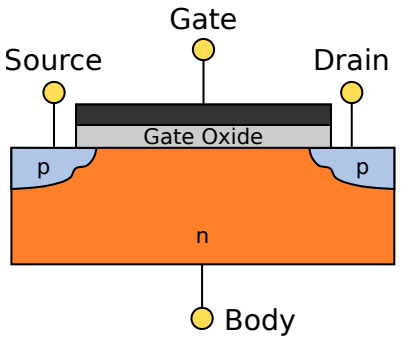
Transistors

Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET)

Cross-sectional view of a MOSFET:



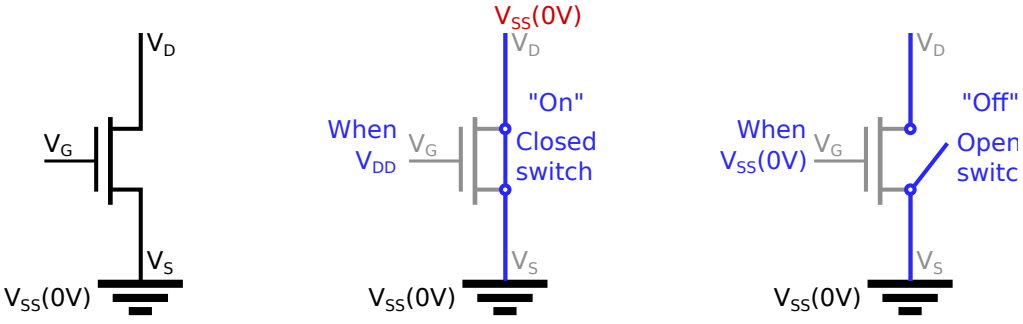
n-type



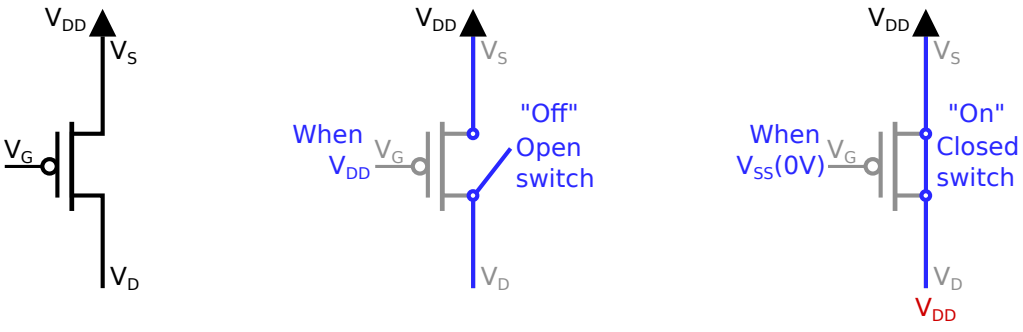
p-type

- N-type transistors take their source from the ground or another N-type transistor.
- P-type transistors take their source from the voltage supply or another P-type transitor.

Complementary MOS (CMOS)



NMOS transistor



PMOS transistor

V_{DD} = supply voltage

V_{SS} = ground (0V)

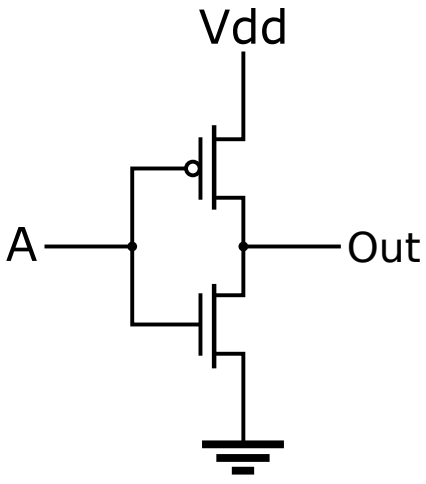
V_G = Votlage at the gate

V_S = Voltage at the source

V_D = Voltage at the drain

Logic gates with CMOS transistors

Inverter:

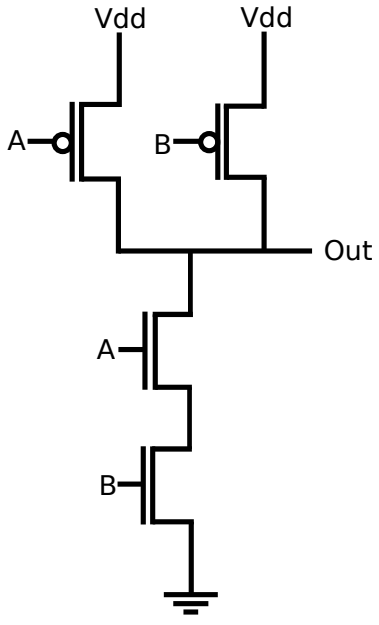


source: Public domain, https://en.wikipedia.org/wiki/File:CMOS_Inverter.svg

2 transistors:

- 1 PMOS;
- 1 NMOS.

NAND gate:



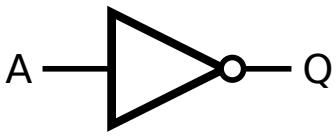
source: JustinForce, CC BY-SA 3.0, via Wikimedia Commons

4 transistors:

- 2 PMOS;
- 2 NMOS.

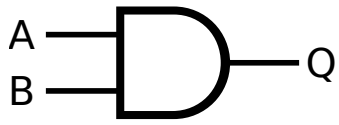
Basic digital logic components

Logic Gates



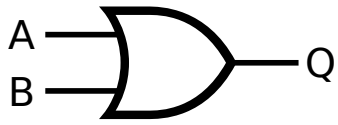
$$Q = \overline{A}$$

Not



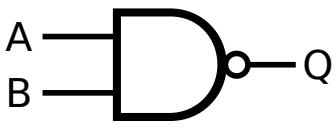
$$Q = A \cdot B$$

And



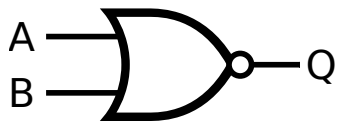
$$Q = A + B$$

Or



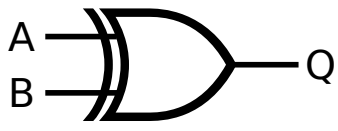
$$Q = \overline{A \cdot B}$$

Nand



$$Q = \overline{A + B}$$

Nor



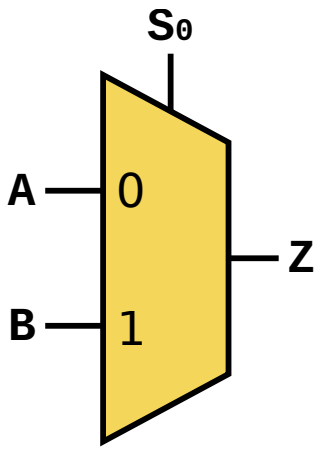
$$Q = A \otimes B$$

Xor

source: https://en.wikipedia.org/wiki/Logic_gate

Multiplexers

e.g., 2-to-1 multiplexer:

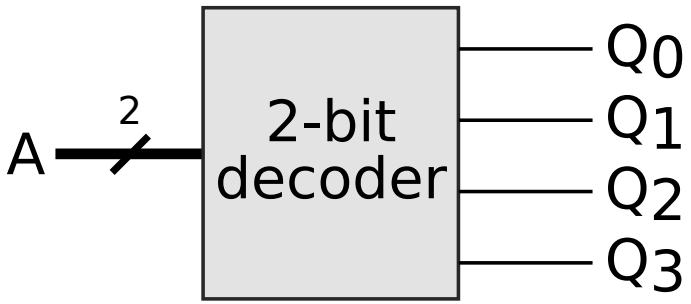


source: https://commons.wikimedia.org/wiki/File:Multiplexer_2-to-1.svg en:User:Cburnett / CC BY-SA 3.0

S	Z
0	A
1	B

Decoders

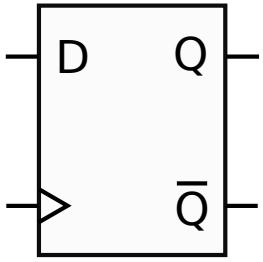
e.g., 2-bit decoder:



A	Q_0	Q_1	Q_2	Q_3
00	1	0	0	0
01	0	1	0	0
10	0	0	1	0
11	0	0	0	1

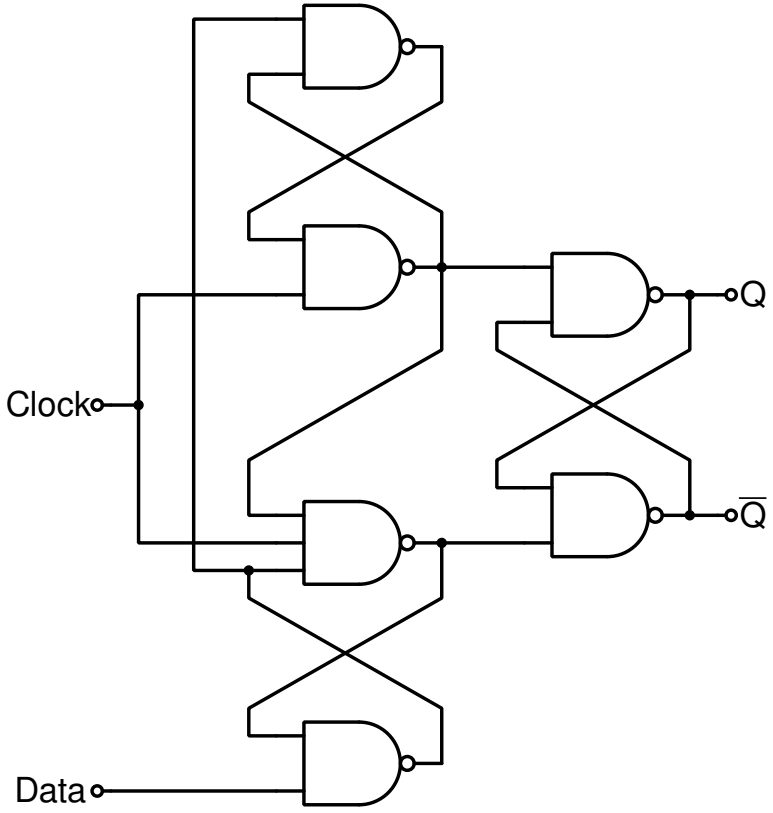
Flip-flops

D Flip-Flop



1 bit of storage

Implementation

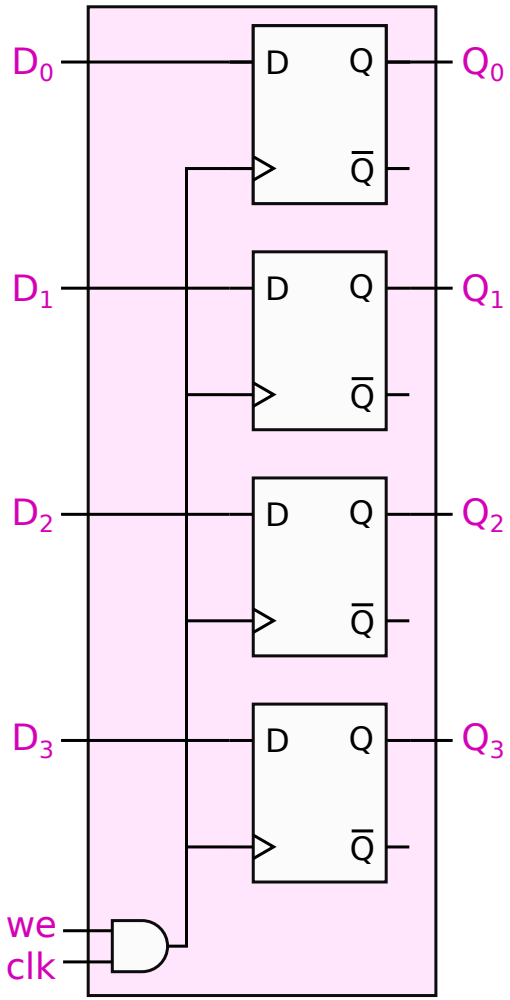


Implemented using ~ 20 transistors in CMOS.

n-bit Register

D flip-flops can combined to create n-bit registers.

e.g., 4-bit register:

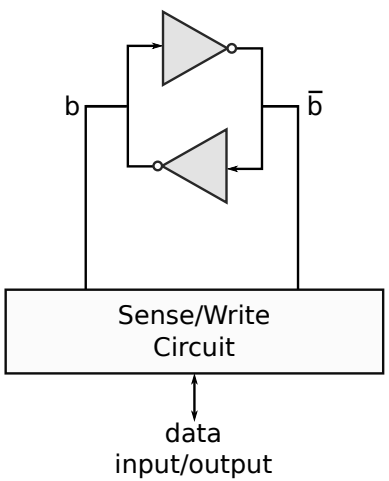


The **we** signals controls when the data is written into each flip-flop.

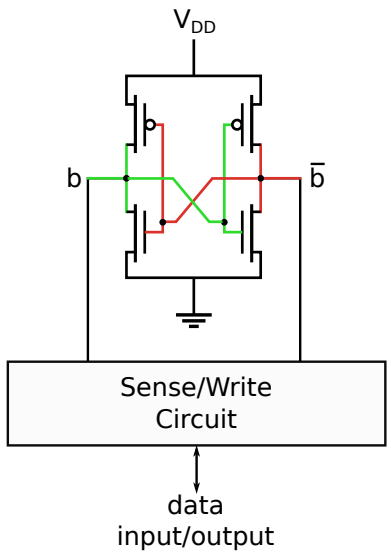
Memory Bit Cell

Using flip-flops for storing large amount of data is costly in terms of gates/transistors: ≈ 20 transistors / bit.

Far fewer transistors if we use two inverters coupled with a sense/write circuit instead:



Inverters implementation



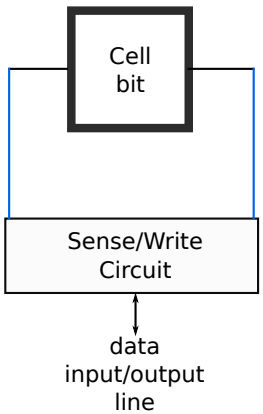
Transistors equivalent

This design only uses 4 transistors per bit.

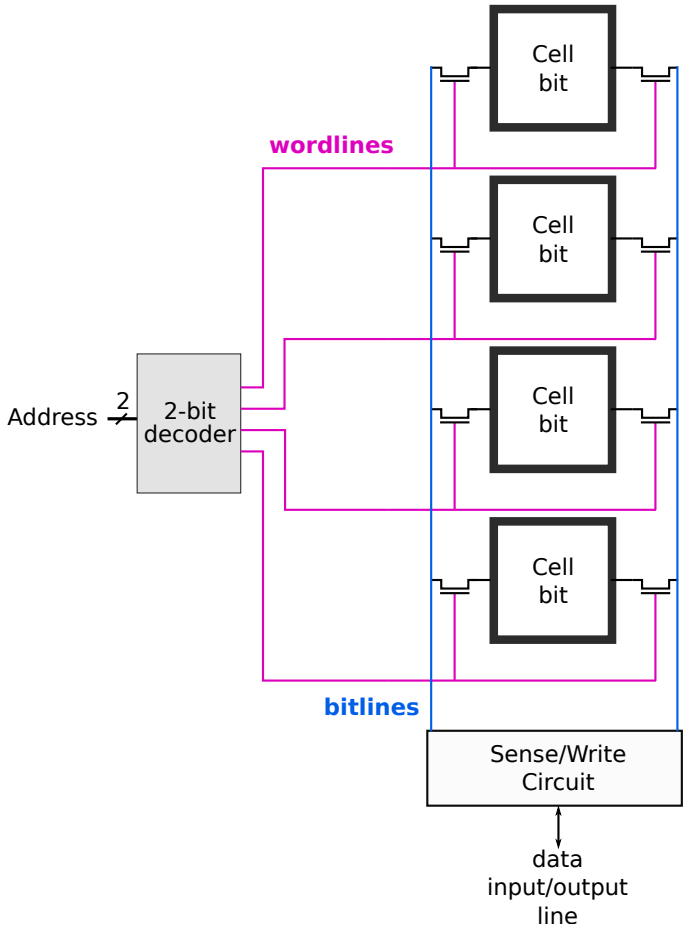
Problem: at the cost of a sense/write circuit for every bit!

Bit Cell Column

Solution: reuse the same sense/write circuit for several cells!



Bit cell

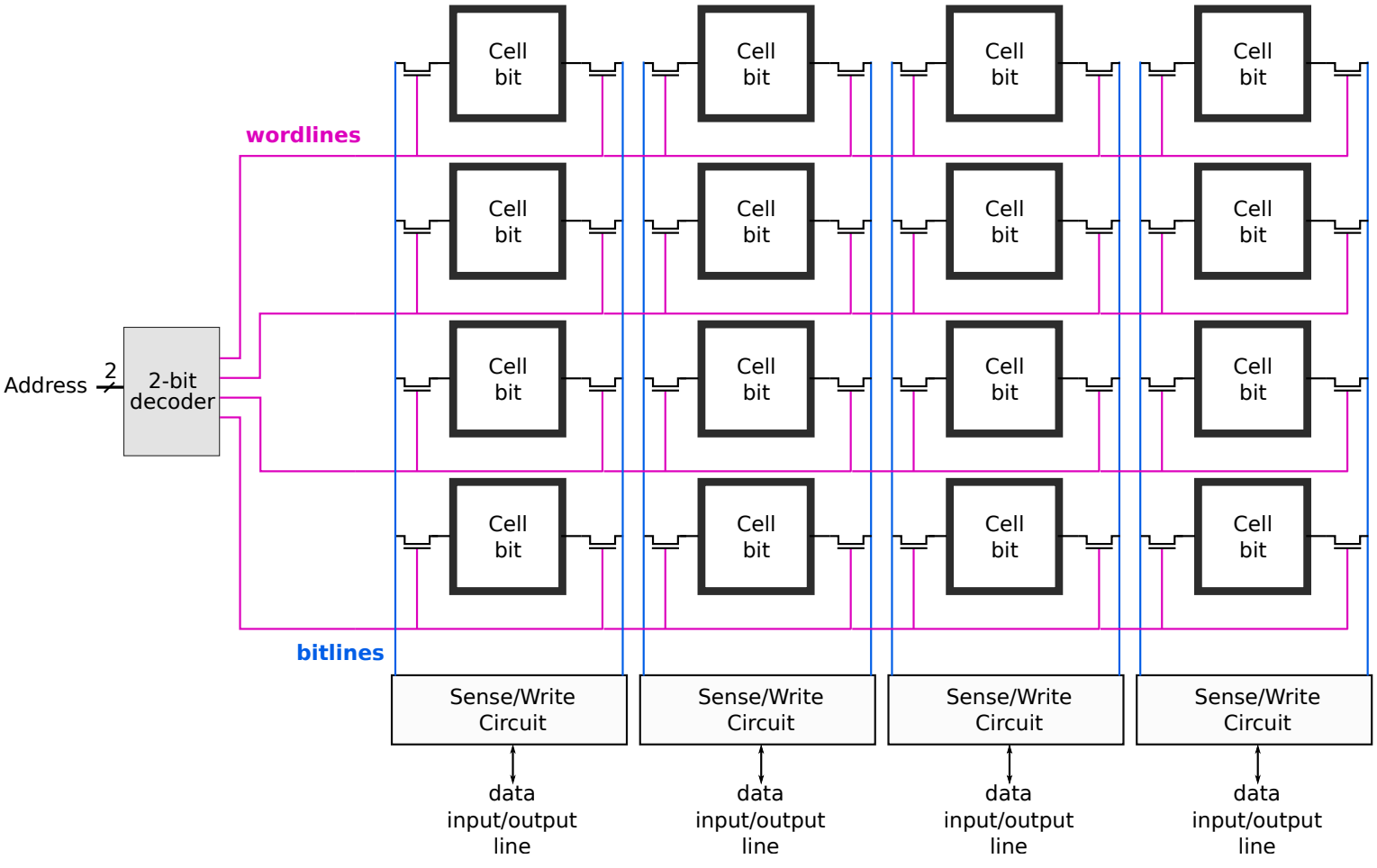


Bit cell column: 6T per cell

Problem: Each cell requires extra logic because of the decoder.

Cell Array

Solution: reuse the decoder for multiple rows of cells!



This is the basic implementation of Random Access Memory (RAM).

Random Access Memory (RAM)

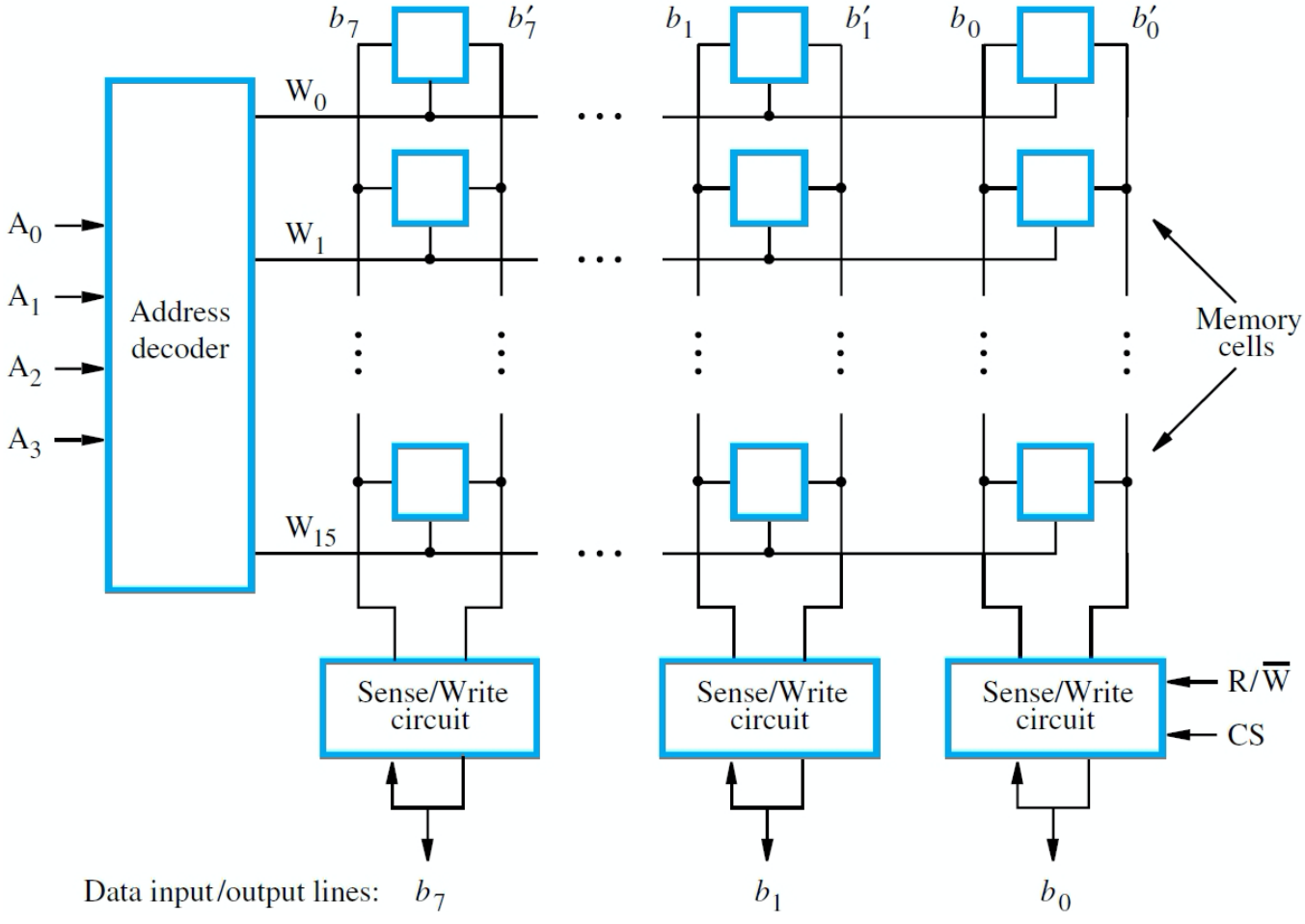
Semiconductor RAM is organized as a 2-D array of cells, each storing a single bit.

- Each row of the array stores one memory word – *note! a memory word may be different in size than a processor word!!*
- Square-ish shape for the cell array are preferred to reduce latency. Why?

Exercise: consider a 16x8 RAM with an 8-bit word size and 16 words.

- How many bits does this memory store?
- How many bits are needed for the memory address?

Example: 16x8 RAM



source: Copyright Carl Hamacher, Zvonko Vranesic, Safwat Zaky, Naraig Manjikian, Computer Organization and Embedded Systems, 2011.

Arithmetic and Logic Operations

Arithmetic and Logic Operations

Binary Integer Arithmetic (Recap)

Textbook§1.4, 1.5

Unsigned Integers

Decimal number $D = d_{n-1}d_{n-2}\ldots d_1 d_0$ where $d_i \in \{0, 1, \ldots, 9\}$

Value in base 10 $V(D) = \sum_{i=0}^{N-1} d_i \times 10^i$

e.g., $67 = 6 * 10^1 + 7 * 10^0 = 67$

Binary number $B = b_{n-1}b_{n-2}\ldots b_1 b_0$ where $b_i \in \{0, 1\}$

Value in base 10 $V(D) = \sum_{i=0}^{N-1} d_i \times 2^i$

e.g., $0100\ 0011 = 1 \times 2^6 + 1 \times 2^1 + 1 \times 2^0 = 67$

E.g., $67_{10} = 0100\ 0011_2$

E.g., $13_{10} = 0000\ 1101_2$

The range of values depends on number of bits n: $V(D) \in [0; 2^n - 1]$.

E.g., if $n = 8$ bits, the maximum value is
$2^8 - 1 = 255_{10} = 1111\ 1111_2$.

Binary Addition

Decimal addition

```
   67
+  13
----
    1
   80
```

Binary addition

```
  0100 0011
+ 0000 1101
-----------
       1111
  0101 0000
```

Watch out for overflow!

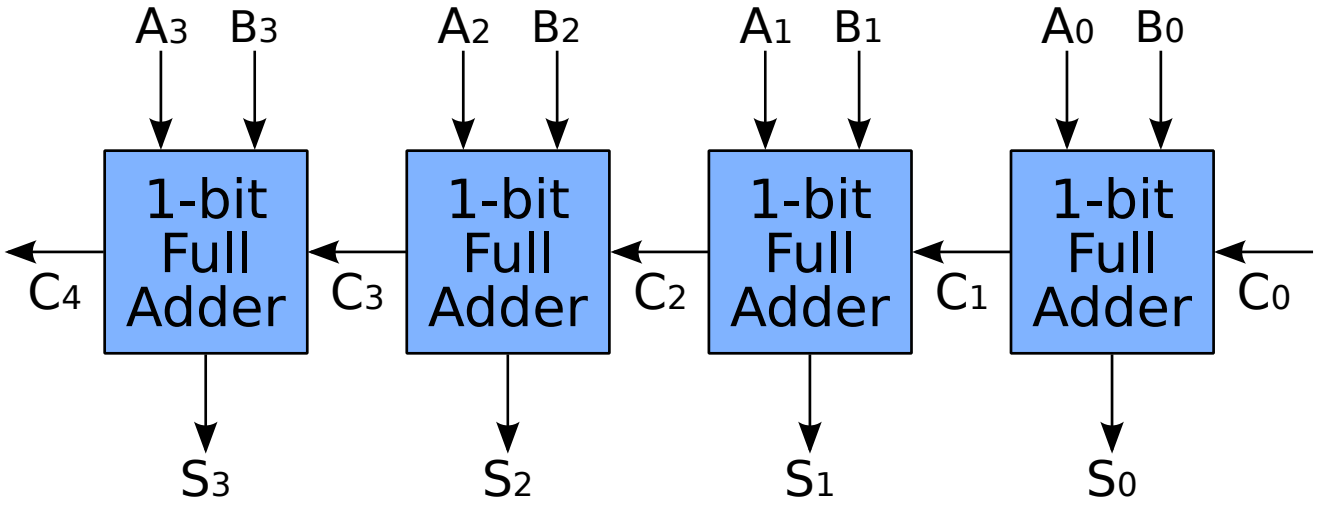
```
   195
+  141
----
    1
   336
```

```
  1100 0011
+ 1000 1101
-----------
1      1111
  0101 0000
```

336 is larger than the maximum value (255) we can represent with 8 bits. The carry-out indicates the overflow.

Binary Addition in Hardware

Ripple carry adder: $S = A + B$



source: https://commons.wikimedia.org/wiki/File:4-bit_ripple_carry_adder.svg en:User:Cburnett / CC BY-SA

Signed Integers: Sign-and-magnitude Representation

We need to encode the sign in the representation of signed binary numbers.

Sign-and-magnitude is the simplest approach: use the leftmost bit (MSB) to represent the sign, and the remaining bits to represent the magnitude (*i.e.*, absolute value). Example for 8 bits:

$$\texttt{MSB} = \texttt{0} \Rightarrow \texttt{positive} \qquad +13 = \texttt{00001101}$$
$$\texttt{MSB} = \texttt{1} \Rightarrow \texttt{negative} \qquad -13 = \texttt{10001101}$$

Problems with sign-and-magnitude:

- Two representations for zero = 0000 0000 = 1000 0000
- We need extra hardware to handle the addition of a positive number and a negative one (we cannot simply add the numbers together)

Signed Integers: 1's-complement Representation

To get a negative value: invert each bit of the corresponding positive representation, and vice-versa.

This representation has the advantage that signed and unsigned arithmetic can use the same hardware.

$$
\begin{aligned}
+13 &= 00001101 \\
-13 &= 11110010
\end{aligned}
$$

$$
\begin{array}{rll}
 & 0001\,0000 & = (16_{10}) \\
+ & 1111\,0010 & = (-13_{10}) \\
\hline
 & {\scriptstyle 1\,1\,1\,1}\;0000\,0010 & = (2_{10})
\end{array}
$$

This result is off by one;
carry out, but no overflow.

Overflow

Overflow occurs when the result of an arithmetic operation does not fit into the range of the n-bit representation used, *e.g.*, $[-2^{n-1}, 2^{n-1} - 1]$ when a bit is used to represent the sign.

If there is a carry out during *unsigned* arithmetic, overflow has occurred.

$$\begin{array}{rll} & \texttt{0001 0000} & = (16_{10}) \\ + & \texttt{1111 0010} & = (242_{10}) \\ \hline & \texttt{0000 0010} & = (2_{10}) \end{array}$$

$\texttt{1111}$

Here, the result is off by 256; the carry out (2^8) indicates overflow.

In *signed* arithmetic, overflow must be detected differently.

Back to 1's-complement Representation

$$+13 \quad = \texttt{00001101}$$
$$-13 \quad = \texttt{11110010}$$

$$
\begin{array}{rrl}
 & 0001\,0000 & = (16_{10}) \\
+ & 1111\,0010 & = (-13_{10}) \\
\hline
 & {\scriptstyle 1\,1\,1\,1} & \\
 & 0000\,0010 & = (2_{10})
\end{array}
$$

This result is off by one;
carry out, but no overflow.

Problems:

- Still two representations for zero = 0000 0000 = 1111 1111
- Need to add 1 to the result when an operand is negative (try as an exercise with $(-2)_{10} + (-2)_{10}$)
- Need a way to identify overflow

Signed Integers: 2's-complement Representation

For integer arithmetic, computers use 2's-complement representations.

To get a negative value: invert each bit of the corresponding positive representation and add one (works in reverse as well).

$$
\begin{aligned}
+13 &= 0000\ 1101 \\
-13 &= 1111\ 0010 + 1 \\
&= 1111\ 0011
\end{aligned}
$$

$$
\begin{array}{rll}
 & 0001\ 0000 & = (16_{10}) \\
+ & 1111\ 0011 & = (-13_{10}) \\
\hline
 & {\scriptstyle 1\ 1\ 1\ 1}\ 0000\ 0011 & = (3_{10})
\end{array}
$$

Correct value; however, carry out without actual overflow again!

Problem:

- Still need a way to identify overflow

Overflow in 2's Complement Addition

Recall that overflow occurs when the answer does not fit into the representable range of numbers.

Observations:

- With signed addition, the carry-out does not indicate overflow.
- Overflow can only happen if both numbers have the same sign.

Rule: Overflow only occurs if both summands have the same sign, and the sum has a different sign than that of the summands.

$$\begin{array}{rll} & 0110 & = (+6_{10}) \\ + & 0100 & = (+4_{10}) \\ \hline & 1010 & = (+10_{10}) \end{array}$$

No carry out, different sign
$\Rightarrow$ overflow!

$$\begin{array}{rll} & 1110 & = (-2_{10}) \\ + & 1001 & = (-7_{10}) \\ \hline {\scriptstyle 1}\,\, & 0111 & = (9_{10}) \end{array}$$

Carry out, different sign
$\Rightarrow$ overflow!

Ranges

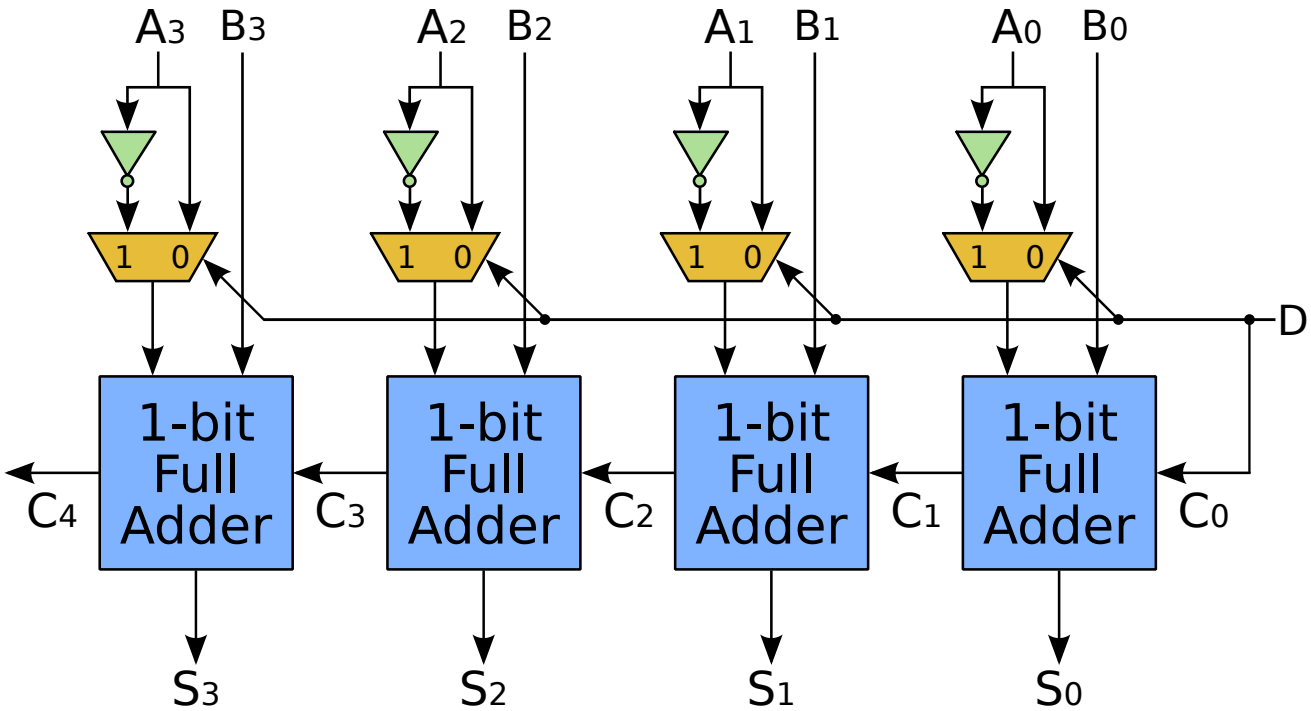
Integer representations, assuming $n = 4$ bits:

Binary	Decimal Value		
	Sign and Magnitude	1's Complement	2's Complement
1000	-0	-7	-8
1001	-1	-6	-7
1010	-2	-5	-6
1011	-3	-4	-5
1100	-4	-3	-4
1101	-5	-2	-3
1110	-6	-1	-2
1111	-7	-0	-1
0000	+0	+0	+0
0001	+1	+1	+1
0010	+2	+2	+2
0011	+3	+3	+3
0100	+4	+4	+4
0101	+5	+5	+5
0110	+6	+6	+6
0111	+7	+7	+7
Range:	$[-7; +7]$	$[-7; +7]$	$[-8; +7]$
	$[-2^{n-1}+1; 2^{n-1}-1]$	$[-2^{n-1}+1; 2^{n-1}-1]$	$[-2^{n-1}; 2^{n-1}-1]$

Subtraction

$B - A = B + (-A)$: form the 2's complement inverse of A and add to B.

In hardware, invert the bits and add one using the carry in signal C_0.
The signal D selects between addition and subtraction.



source: https://commons.wikimedia.org/wiki/File:4-bit_ripple_carry_adder-subtracter.svg en:User:Cburnett / CC BY-SA

Sign Extension

Sometimes you will want to convert an `n`-bit number to an `m`-bit number, where `m` $>$ `n`.

The rule for 2's complement numbers is to replicate (*extend*) the sign bit.

4-bit value	8-bit value	
`0010`	`0000 0010`	$= (2_{10})$
`1110`	`1111 1110`	$= (-2_{10})$

Sign-extension is important if (when) we store numbers in memory using fewer bits than our processor uses for its operations.
E.g., we may use 8-bit numbers for color or sound, but do math on such numbers using a 32-bit adder.

Arithmetic and Logic Operations

Logical Operations

Bitwise boolean operators

Besides arithmetic operations such as a adding or subtracting, a computer must be able to perform logical operations such as **AND**, **OR**, **NOR**, ...

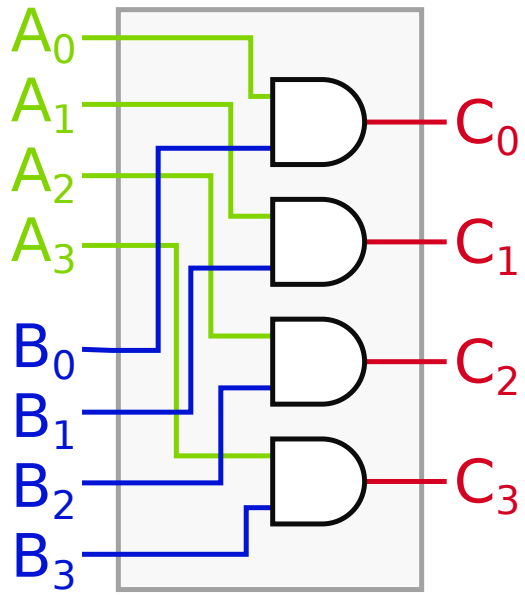
Since a processor typically process data at the granularity of a word (*e.g.*, 32 bits), the logical operator will be applied to all the bits of their input independently.

For instance, for a 4-bit machine:

```
     0011
AND  1010
-----------
     0010
```

```
     0011
OR   1010
-----------
     1011
```

Hardware implementation of a 4-bit wide AND:



Shifting operators

Shift change the positions of bits, moving them left or right.

Computer typically perform three kind of shifting operations:

- Logical Shift Left (LSL) $\ll$
- Logical Shift Right (LSR) $\ggg$
- Arithmetic Shift Right (ASR) $\gg$

Logical vs Arithmetic shift:

- Logical $\Rightarrow$ pad with 0s
- Arithmetic $\Rightarrow$ extend sign bit

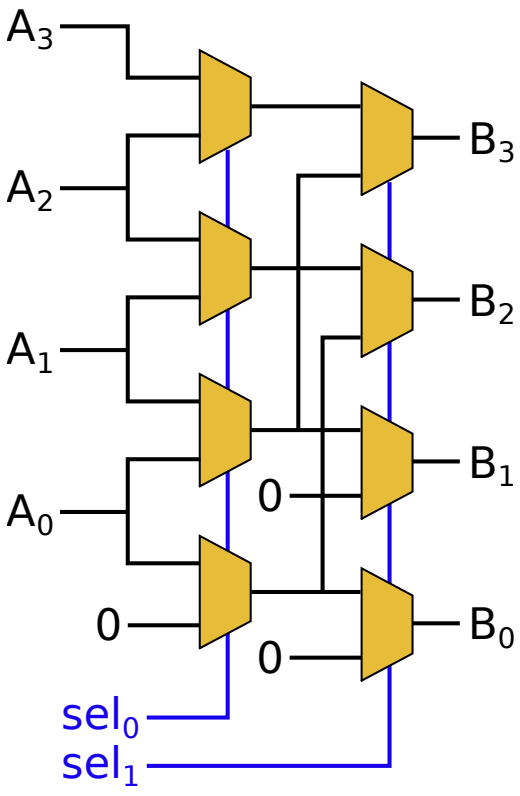
Example: `1100` $\gg$ `0010` = ?

These operators are useful to implement multiplication in software:

- Left shift by one $=$ multiplication by 2;
- Right shift by one $=$ division by 2.

You will see more about this in the labs/tutorials.

Hardware implementation of a 4-bit Barrel Shifter for **LSL**:



$\texttt{sel}_0$ shifts left by 1 if 1

$\texttt{sel}_1$ shifts left by 2 if 1

source: By Aaron Logan, from http://www.lightmatter.net/gallery/albums.php, CC-BY

To support larger bit-width, simply add extra stages to shift by 4, 8, ...

Question:

- How do you shift left by 3?
- How would you update this circuit to shift right?
- What about arithmetic shift?

Arithmetic and Logic Operations

ALU

Arithmetic and Logic Unit (ALU)

Computing is (mostly) about performing arithmetic and boolean operations.

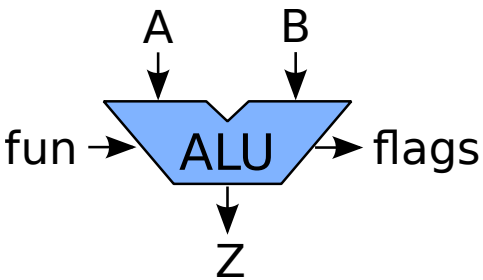
At the core of any computer lies the ALU. It performs integer operations such as adding integers, and bit-wise logical operations such as OR on a word (*e.g.*, 32 bits).

The ALU inputs are:

- Two values, A and B;
- A function determining the operation to perform (*e.g.*, ADD, OR, LSL).

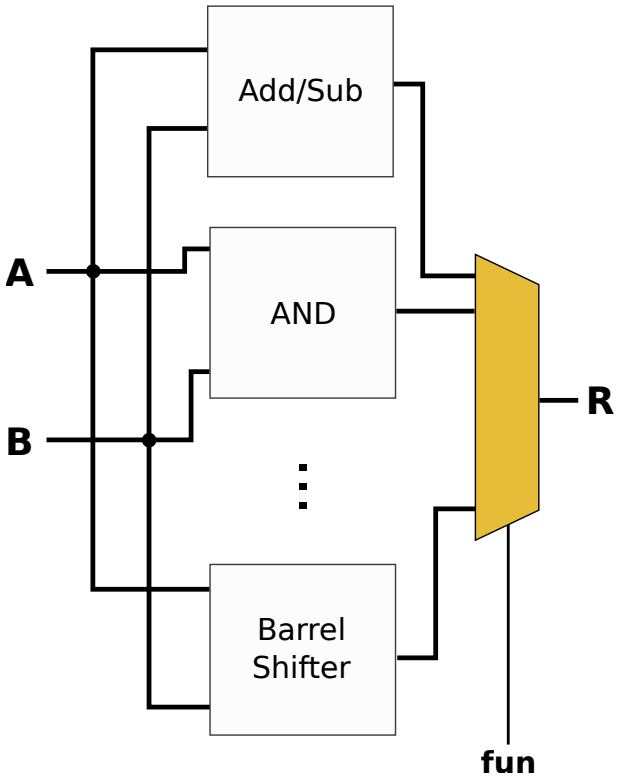
The output of the ALU are:

- The result Z of the operation;
- A set of flags that indicate, for instance, if an overflow has occurred.



ALU Implementation

The ALU can be simply implemented by multiplexing the ADD/SUB unit seen earlier, the logical bitwise operator and the Barrel shifter.



(flags omitted)

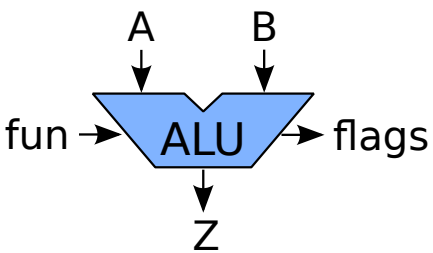
The `fun` select signal of the multiplexer controls which operation is performed by the ALU.

Programming the ALU

An ALU is in fact a *programmable* piece of hardware since its function can be decided at runtime.

For instance, an ALU could be designed to perform the following operations controlled by the `fun` signal:

Function	fun	Description	Output
ADD	000	signed integer addition	A+B
SUB	001	signed integer subtraction	A-B
AND	010	bitwise AND	A and B
OR	011	bitwise OR	A or B
NOR	100	bitwise NOR	A nor B
LSL	101	logical shift left	A $\ll$ B[3-0]
LSR	110	logical shift right	A $\ggg$ B[3-0]
ASR	111	arithmetic shift right	A $\gg$ B[3-0]



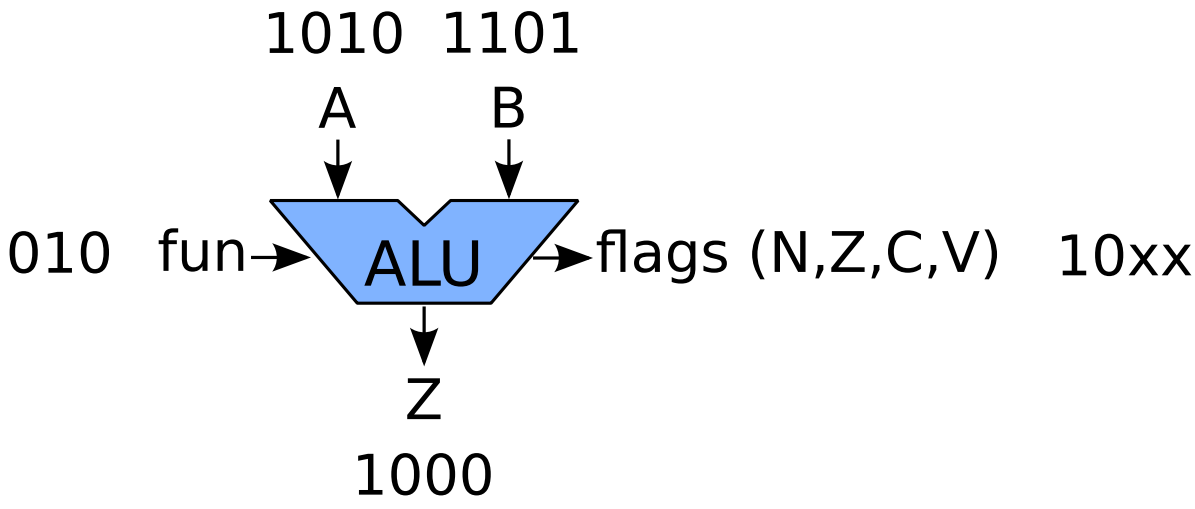
ALU Flags

The ALU also outputs some flags that specifies some properties of the result. The typical flags found on most ALUs are:

- N: Negative
- Z: Zero
- C: Carry generated (when adding/subtracting)
- O: Overflow generated (when adding/subtracting)

Example

Let us use the ALU to perform a bit-wise AND between `1010` and `1101`:



What about a subtraction?

How I can tell if A smaller than B?

So far we have a piece of hardware that we can configure to do any of the arithmetic or logical operations discussed on two values.

But what if we want to perform x+y-z?

Answer:

- Perform x+y;
- Store the temporay result `tmp` somewhere;
- Perform `tmp`-z;

We need to have a storage and control our ALU sequentially.

We are going to use a sequential circuit.

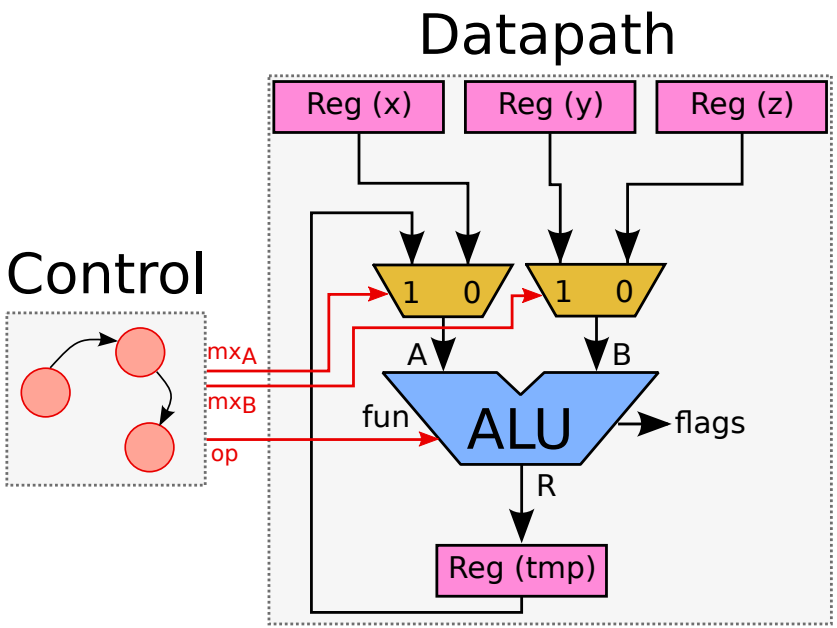
Control and Datapath

Control + Datapath

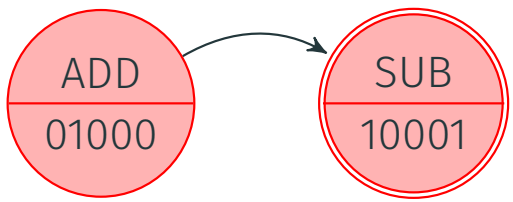
We separate the circuit in two parts:

- Control: typically a Finite State Machine (FSM);
- Datapath: registers, ALU and other functional units.

Example for $x + y - z$:



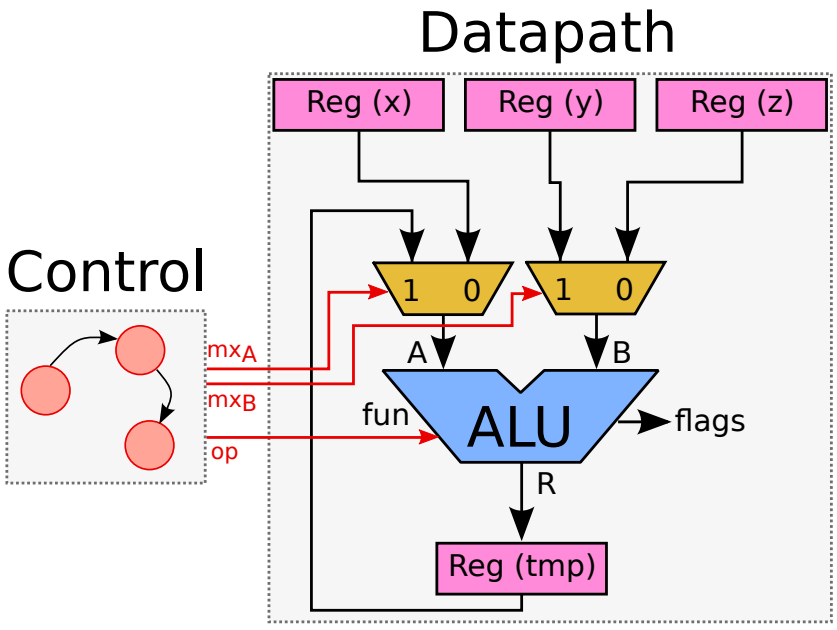
FSM for $x + y - z$:



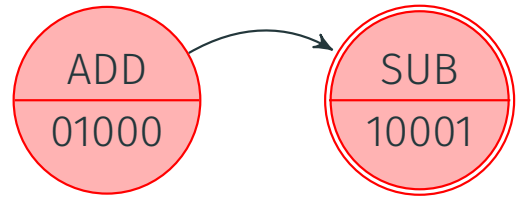
Output control signals: $\texttt{mx}_A\texttt{mx}_B\texttt{op}$

This machine executes one operation per cycle.

Example for $x + y - z$:



FSM for $x + y - z$:



Output control signals: $mx_A mx_B op$

Problem:

- What if we have more than one temporary value?
  e.g., $(x \ll z + y \ll z)$
- What if we have more than three input values?
  e.g., $(x \ll z + y \ll w)$

$\Rightarrow$ Need a more general mechanism to store values.

Register File

Register File

Constraint: we have to have a fixed number of registers in the datapath — it is hard-ware afterall.

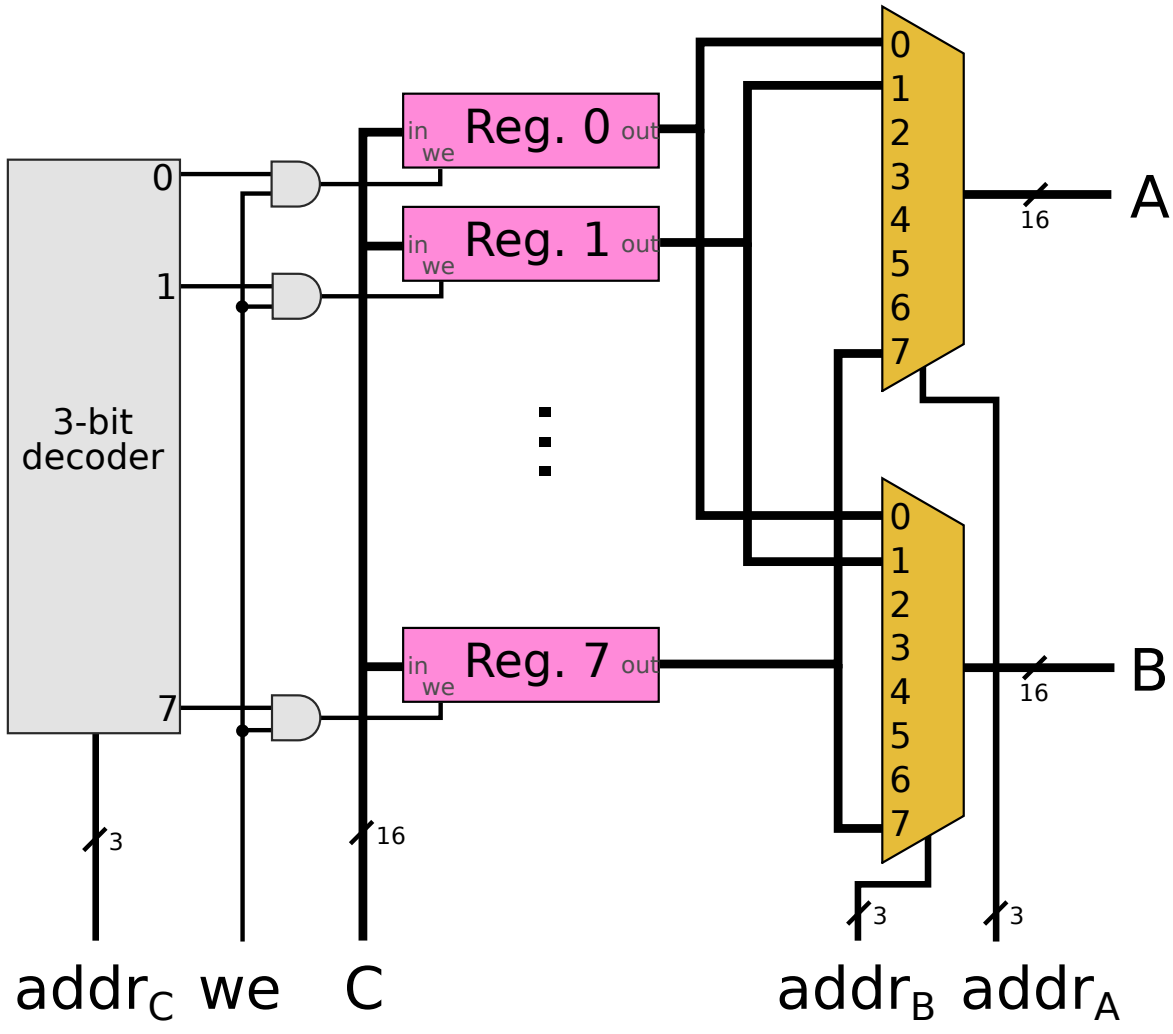
However, we can design our hardware to give us the flexibility to use any register as input, temporary, or output values.

Main idea: group a set of registers together and use one (or many) mutiplexer(s) to select which register(s) to read from, and a decoder to select which register to write to.

This is called a *Register File*.

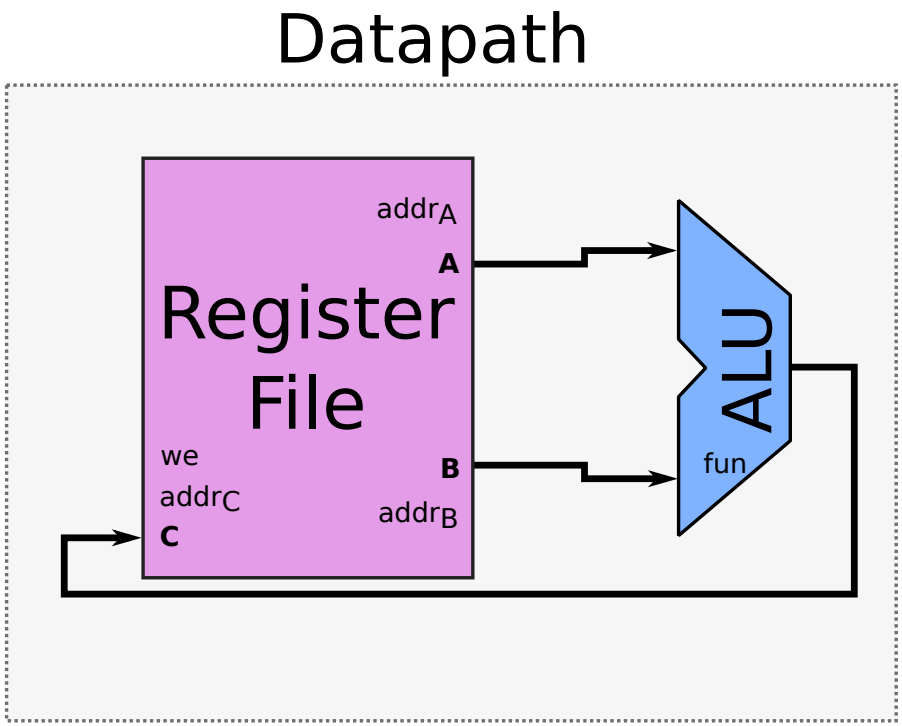
Register File Implementation

Implementation of an 8×16-bit register file with two read ports and one write port:



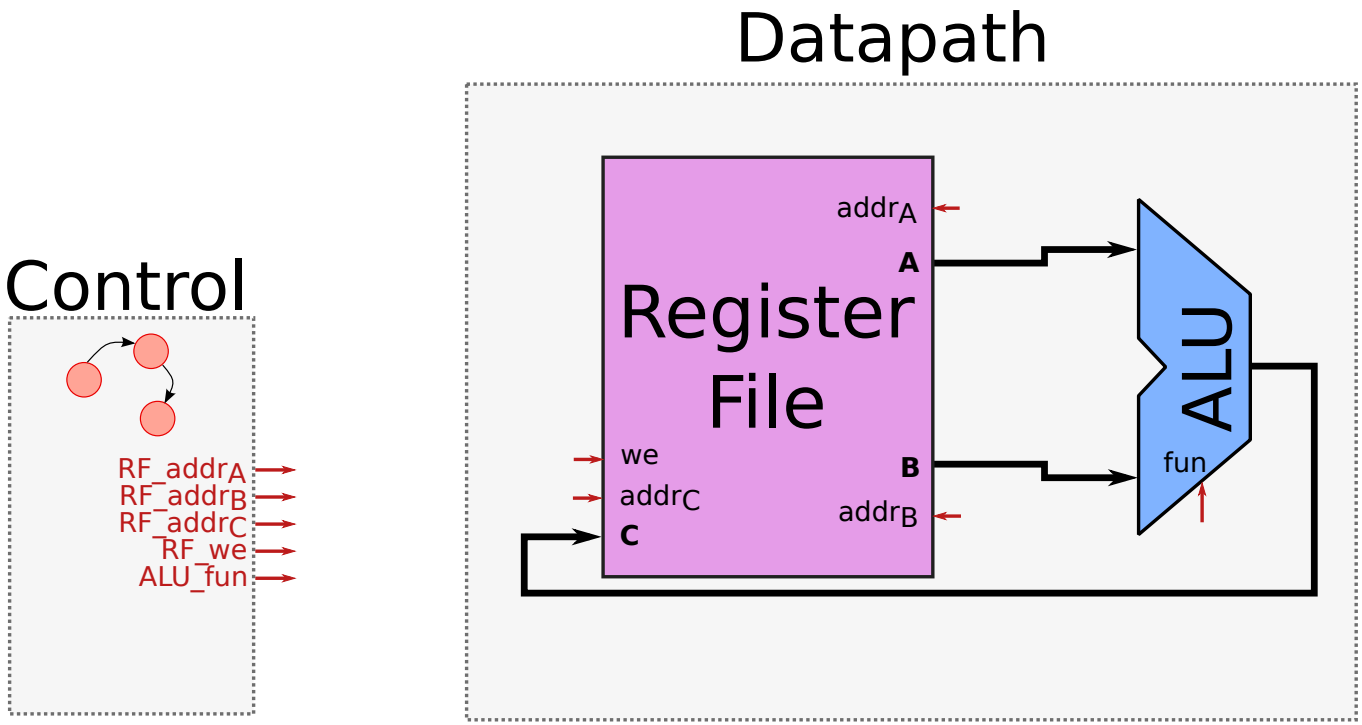
Datapath with Register File

Now we can make the ALU read its input and write its output into any register contained in the register file.



Control + Datapath with Register File

The control logic (FSM) will determine the source and destination by setting the $\texttt{addr}_A$, $\texttt{addr}_B$,and $\texttt{addr}_C$ signals accordingly.



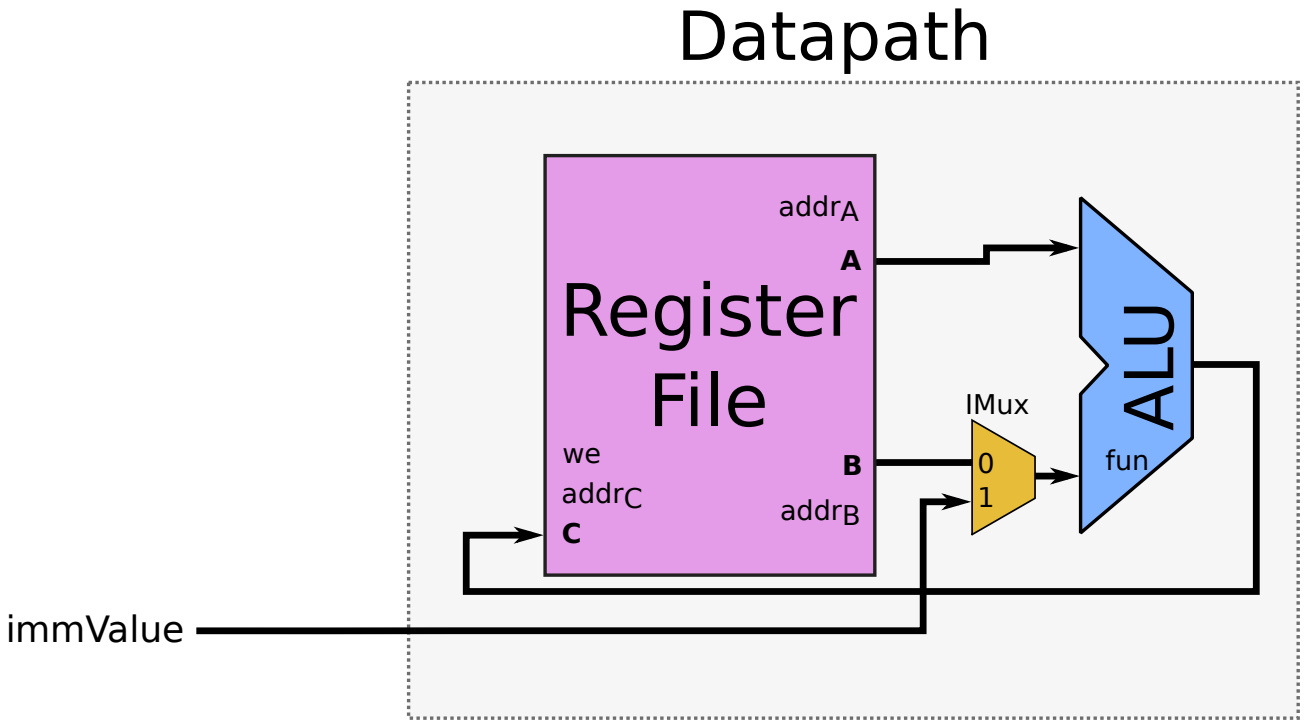
Exercise: draw FSM to perform $\mathtt{x} + \mathtt{y} - \mathtt{z}$. Assume x, y and z are respectively in `R0`, `R1` and `R2` of the register file and the result should be saved in `R3`.

What about: $(\mathtt{x} \ll 2 + \mathtt{y} \ll 3)$? What is the problem?

Dealing With Constants

Constant are also known as immediate value.

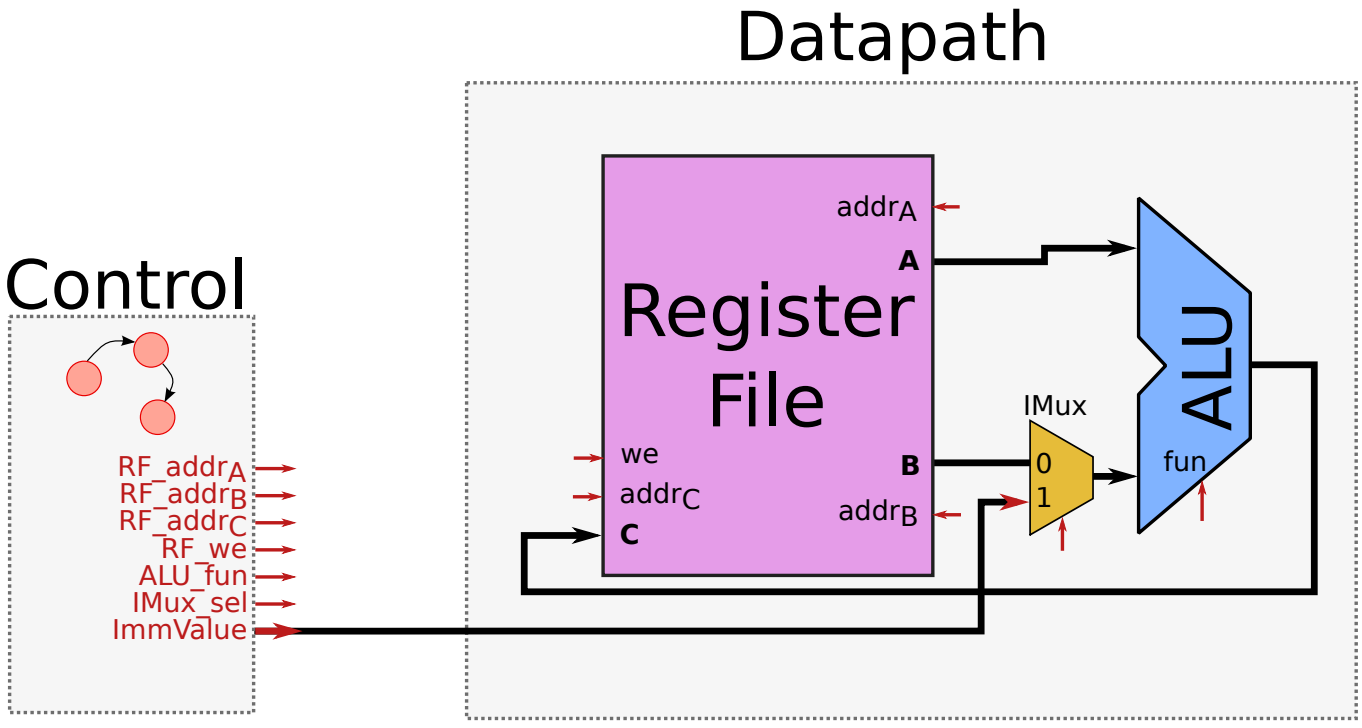
We can modify the datapath and add a multiplexer in front of one of the inputs of the ALU.



Where is the second IMux input coming from?

Dealing With Constants

The control logic will produce the immediate value. It is encoded in one of the states of the FSM.



Exercise: draw the FSM for $(x \ll 2 + y \ll 3)$ and the corresponding output signals that control the datapath. Assume x and y are stored in `R0` and `R1` respectively and the result should be saved in `R7`.

Main Memory

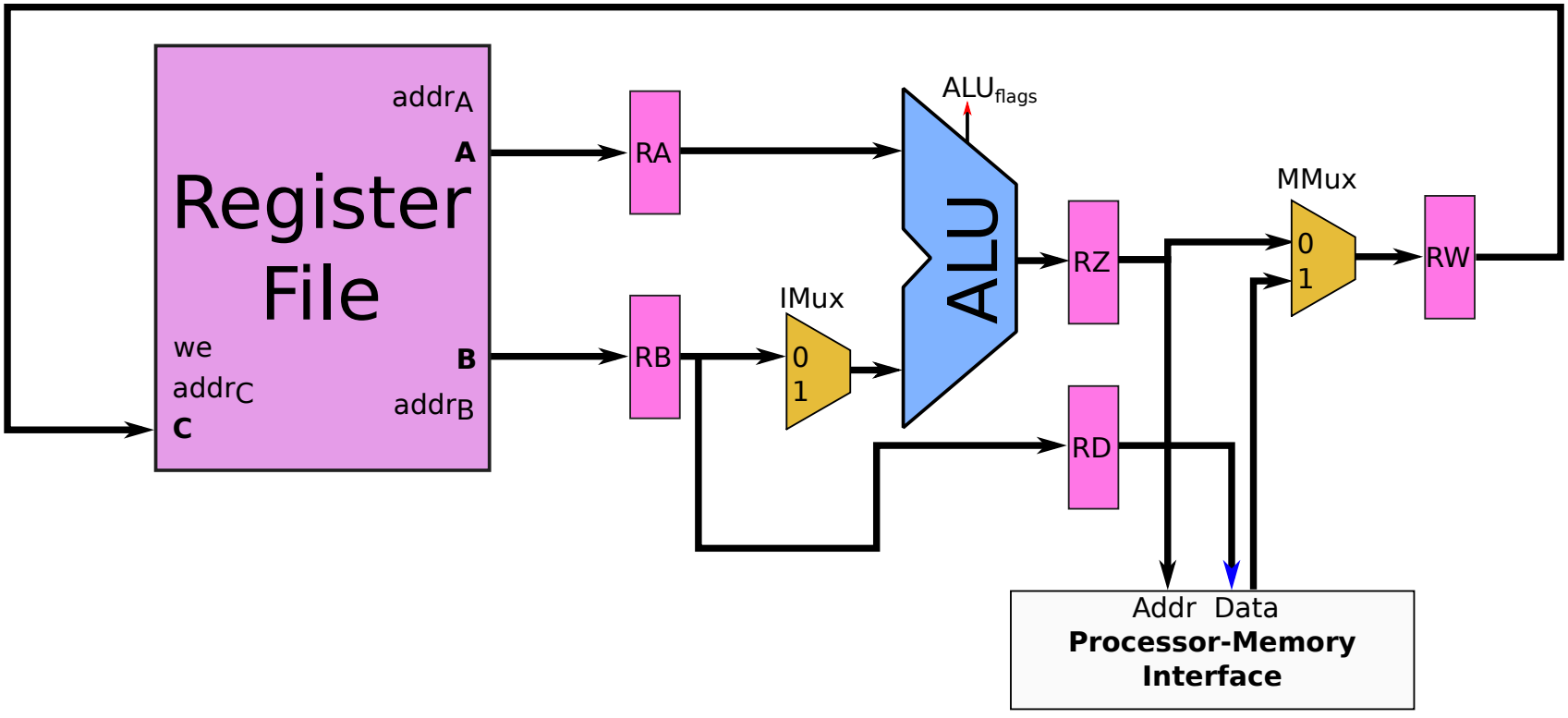
Interfacing with main memory

So far, we have only looked at ALU operations, but the processor also needs to access the main memory.

In a Load/Store architecture, the processor has two dedicated classes of operations to communicate with memory:

- Load: Reads a value from memory at a given adress and saves it into a register.
- Store: Writes the value from a register at a given address in memory.

Full Datapath



To enable Load/Store instruction, we connect our datapath to the processor memory-interface.

- The `Addr` signal, coming out of the ALU, contains the address to load/store data from/to the memory.
- The `Data` signal, is the data coming from / going to the memory.

Notice how we have also added extra registers in the datapath: `RA`,`RB`,`RZ`,`RD`, and `RW`.

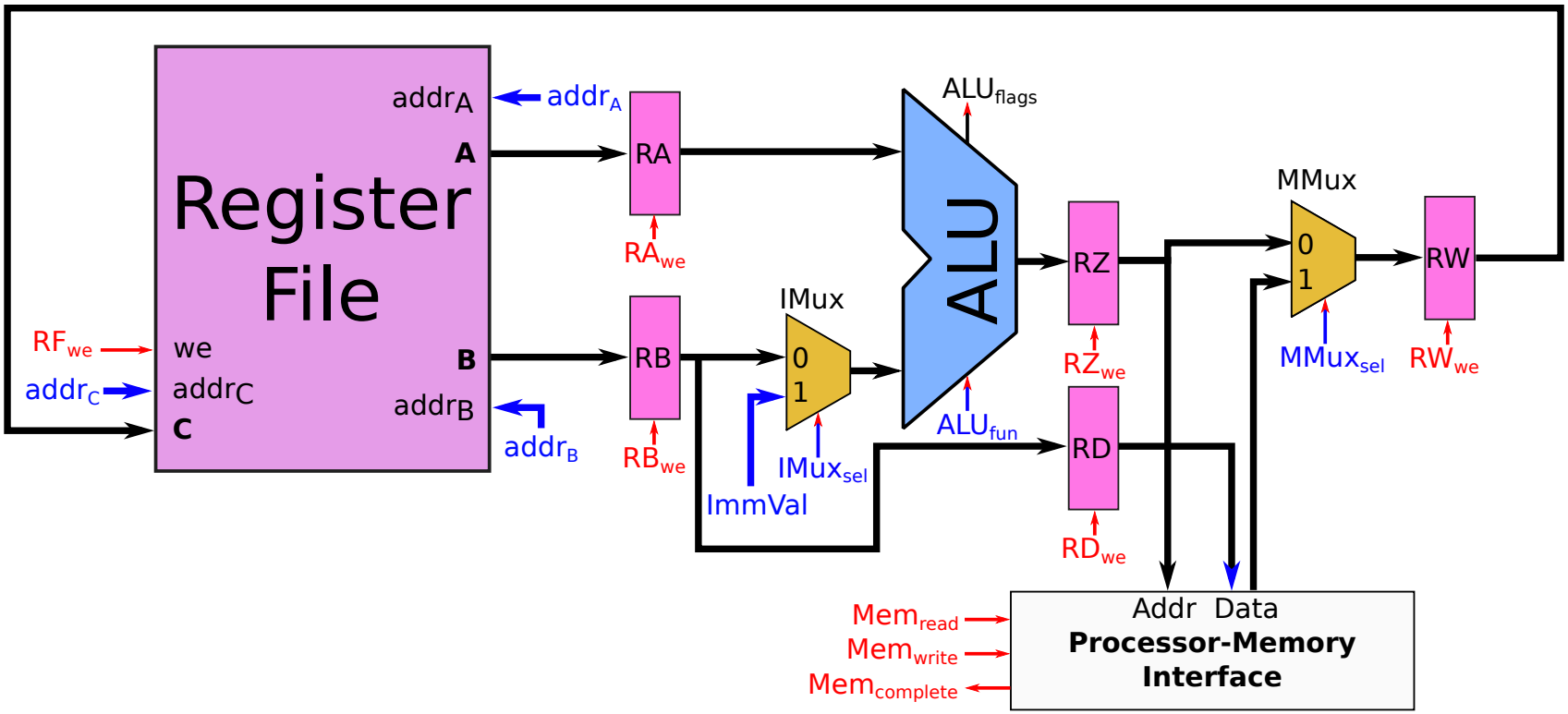
These allows us to “separate” the different steps that takes place in the machine:

- Once the register file has been read, `RA` and `RB` are updated.
- Once the ALU has been used, `RZ` and `RD` are updated.
- Once the memory has performed its operation, if any (*e.g.*, `Load`), `RW` is updated.
- Finally, at the very end the result of `RW` is written into the register file if needed.

These four steps are called: Decode, Execute, Memory, Writeback.

- We now need more than one clock cycle to execute an instruction;
- But this allows us to *pipeline* the datapath. More on this later in the course.

Datapath + Control

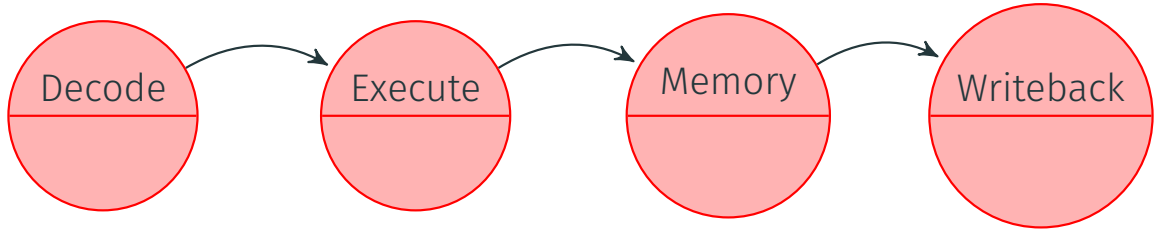


blue = data-path related signals, red = state related signals.

Memory related control signals:

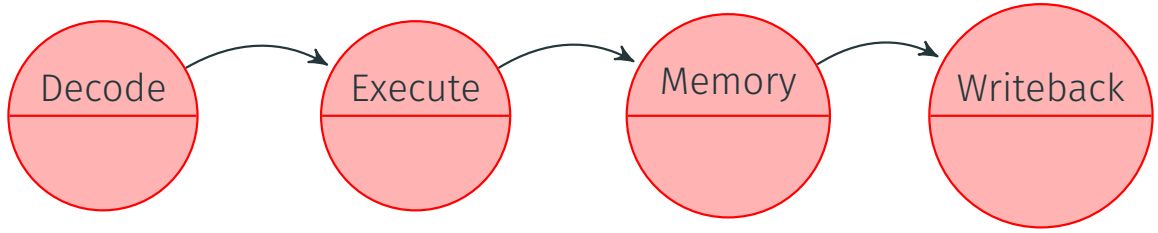
- $\texttt{Mem}_{\texttt{read}}$: load data from memory;
- $\texttt{Mem}_{\texttt{write}}$: store data to memory;
- $\texttt{Mem}_{\texttt{complete}}$: memory function has completed (data has arrived).

Example FSM for `Add, R3, R4, R5` (R3 = R4+R5)



	Decode	Execute	Memory	Write-back
addrA	4	x	x	x
addrB	5	x	x	x
addrC	x	x	x	5
ImmVal	x	x	x	x
$IMux_{sel}$	x	0	x	x
ALU_{fun}	x	ADD	x	x
$MMux_{sel}$	x	x	0	x
RA_{we}	1	0	0	0
RB_{we}	1	0	0	0
RZ_{we}	0	1	0	0
RD_{we}	0	1	0	0
Mem_{read}	0	0	0	0
Mem_{write}	0	0	0	0
RW_{we}	0	0	1	0
RF_{we}	0	0	0	1

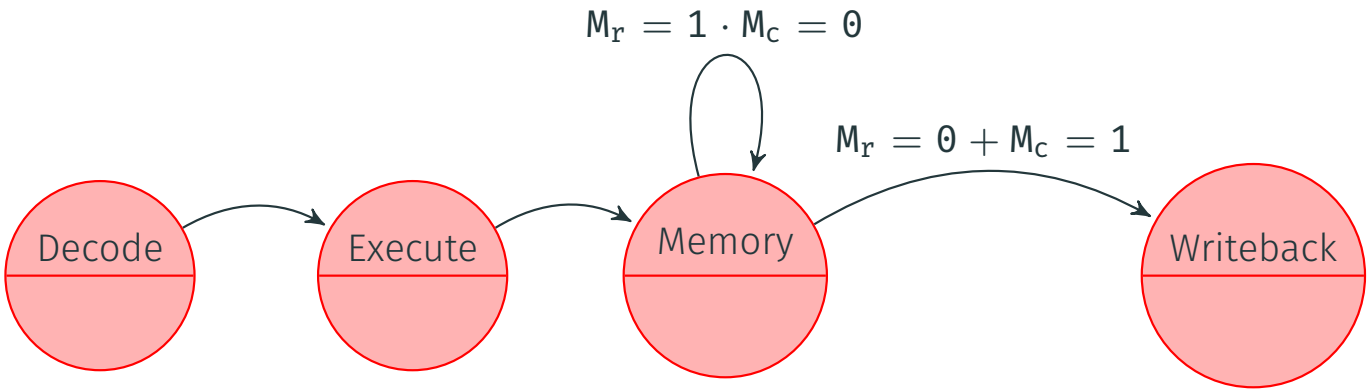
Example FSM for `Load, R2, R5` (R2 = MEM[R5])



	Decode	Execute	Memory	Write-back
addrA	5	x	x	x
addrB	x	x	x	x
addrC	x	x	x	2
ImmVal	x	0	x	x
$IMux_{sel}$	x	1	x	x
ALU_{fun}	x	ADD	x	x
$MMux_{sel}$	x	x	1	x
RA_{we}	1	0	0	0
RB_{we}	1	0	0	0
RZ_{we}	0	1	0	0
RD_{we}	0	1	0	0
Mem_{read}	0	0	1	0
Mem_{write}	0	0	0	0
RW_{we}	0	0	1	0
RF_{we}	0	0	0	1

What if the memory takes more than once clock cycle to send the data back?

We modify our FSM by using the $\texttt{Mem}_{\texttt{complete}}$ signal to only move to the Writeback state once data has been received:



$M_r = \texttt{Mem}_{\texttt{read}}, M_c = \texttt{Mem}_{\texttt{complete}}$.

One last word on memory operations.

What values of $\texttt{IMUX}_{\texttt{sel}}$ and `ImmVal` would you choose to implement:

`Store, R2, R5+4` (MEM[R5+4] = R2) ?

Producing the address through the ALU allows the machine to perform two operations at once: calculate an address and access memory.

Instructions

What if we want to change the computation?

Every time we want our machine to compute something different, we need to change the output signals in each state of the FSM.

Problem: the output of the FSM are hard-coded using combinational logic.

What we want: ability to change the output of the FSM at runtime, make it soft.

Solution: "encode" each state's output signals in a memory.

Instructions

An instruction consists of the operations the machine should perform together with the data source and destination (*i.e.*, register number or immediate value).

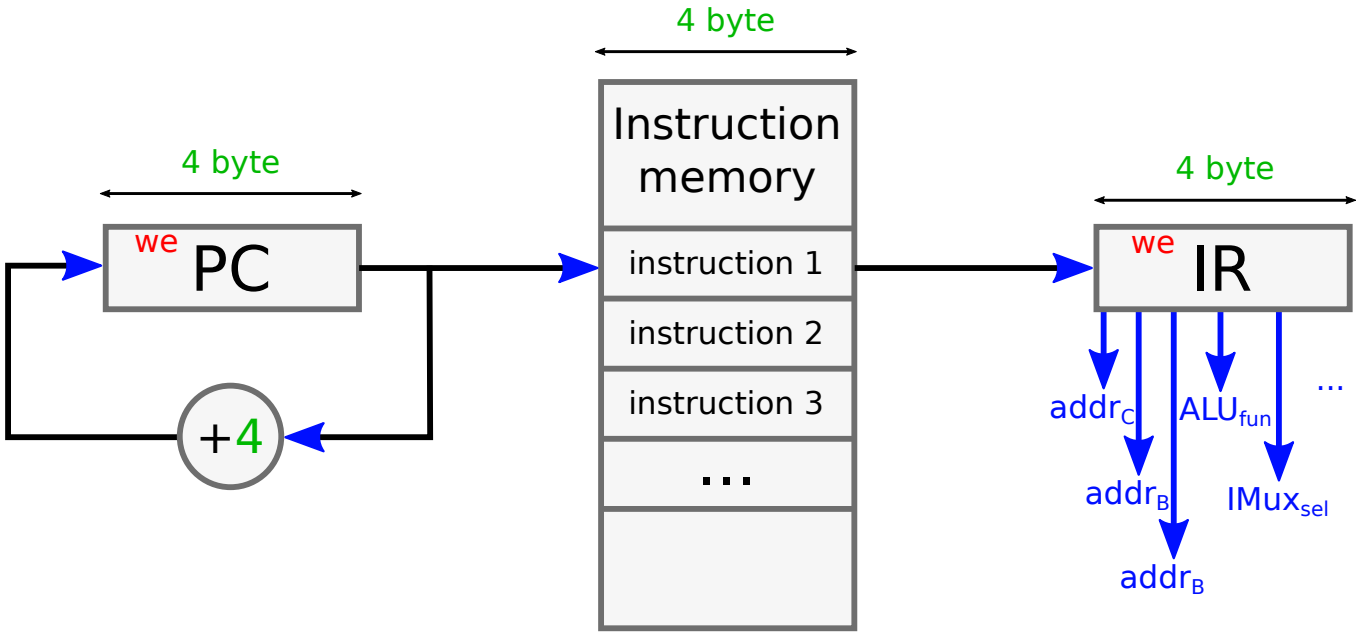
Instructions are stored in a random access memory (RAM).

A program counter (PC) is introduced to keep track of which instruction is executing. Its value is the address in memory of the instruction the machine should execute.

An instruction register (IR) is introduced to hold the currently executing instruction. The output of the `IR` will set the datapath related control signals.

Control for instruction fetching

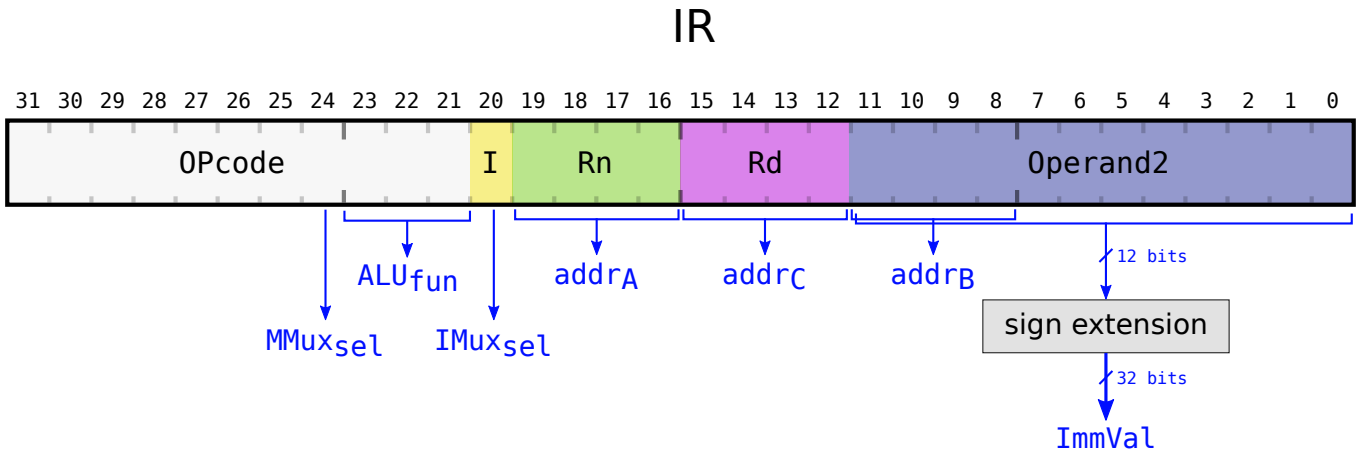
Assuming a 32-bit (4 bytes) processor:



The datapath related control signals now come out of the `IR` (Instruction Register) instead of coming out of the FSM. This allows us to “customize” the path the data takes based on data (instruction) stored in the RAM.

Instruction encoding

With a 32-bit processor, one possible encoding of an instruction and the corresponding datapath related control signals could be:



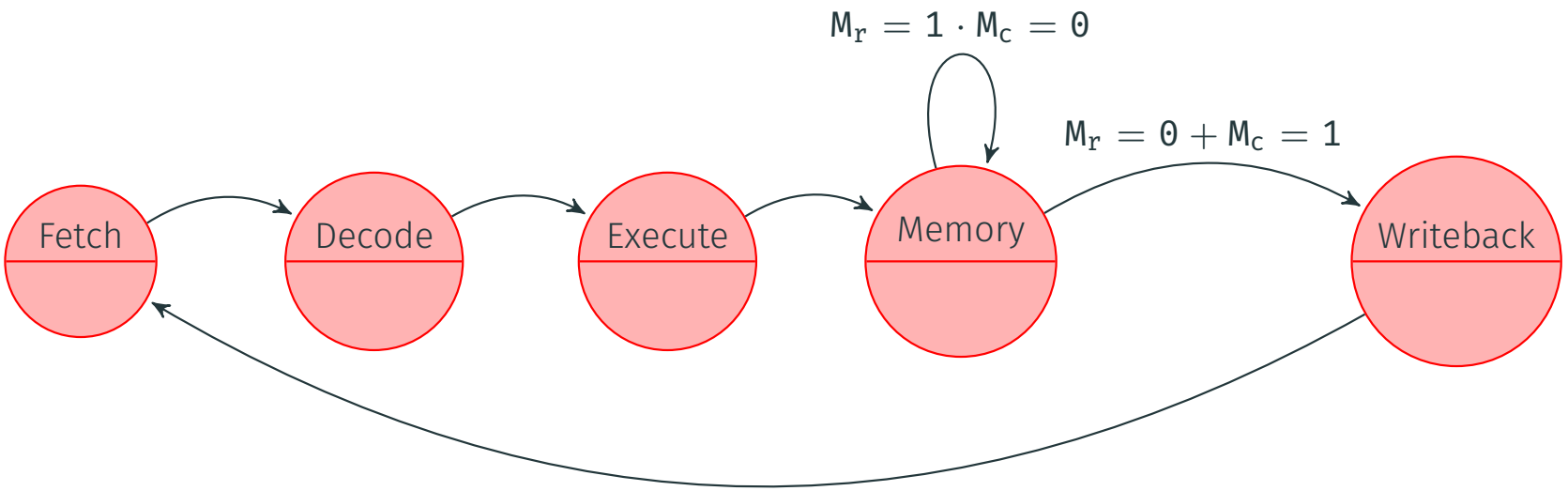
- `OPcode` specifies the type of operations, *e.g.*, ADD, Load, Store;
- `I` specifies whether `Operand2` is a register or immediate value;
- If we have an immediate value, we must sign extend it.

Exercise: What is the content of the instruction memory for the following sequence of instructions ?

```
ADD R2, R3, R4
ADD R2, R3, -3
Load R2, R3
```

Finite State Machine with Fetching

We now introduce an extra state in our FSM to specifically deal with fetching the instruction from memory and can finally “close” the loop.



These are the classical five stages of a RISC processor that repeated over and over:
Fetch, Decode, Execute, Memory and Writeback.

Control Flow

What if we want to change the behaviour of our computation based on runtime values?

For instance, consider the task of implementing a *max* function:

$$max(a, b) = \begin{cases} \mathtt{a} & \text{if } \mathtt{a} \geq \mathtt{b} \\ \mathtt{b} & \text{otherwise} \end{cases}$$

Assume the original value a and b are stored in R0 and R1 respectively, and that the result should be returned in R2.

Depending on the case taken, the processor needs to execute one of these two instructions (but not both!) to "assign" a or b to R2 :

- `ADD R2, R0, 0`
- `ADD R2, R1, 0`

One way to achieve this is to have both instructions in the instruction memory and skip one or the other depending on the case.

This means we need the ability to change the PC to an arbitrary position conditionally on the result of comparing two values.

This operation is called a branch.

Branch instruction

A branch instruction sets the ALU to perform a subtract and uses the condition flags from the output of the ALU to decide whether to update the PC value or not.

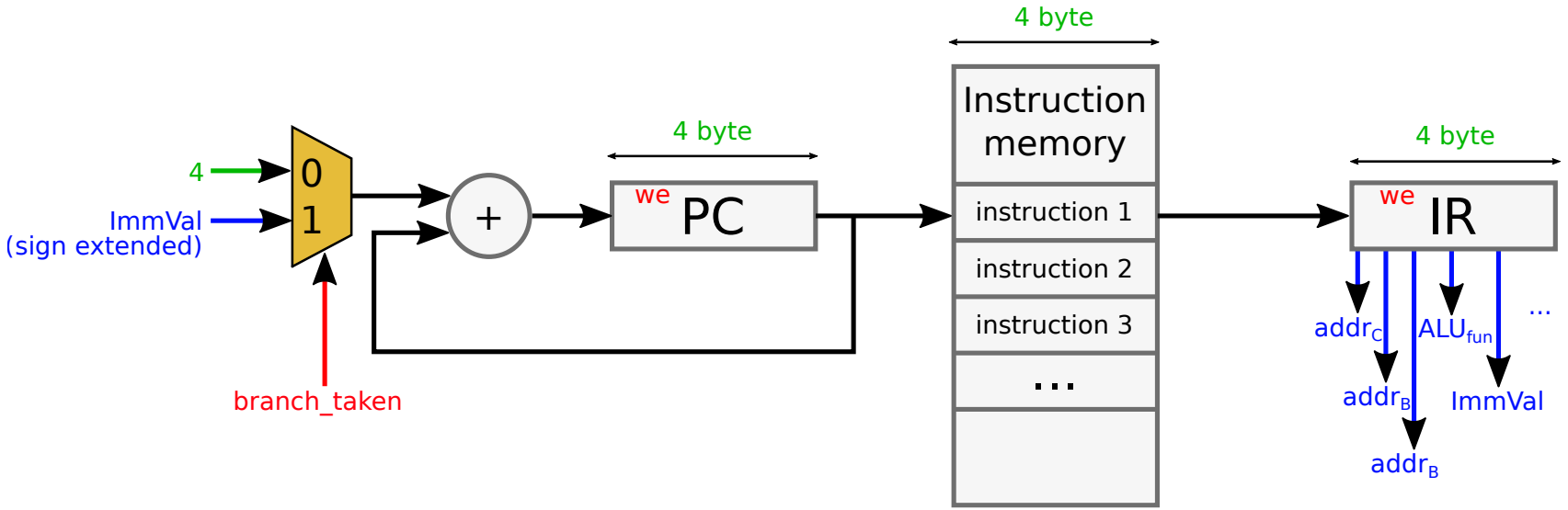
`BGE Rd, Rn, RelativeAddress` = Branch if Rd $\geq$ Rn
where `RelativeAddress` is an immediate value.

Other variants: `BEQ`, `BNE`, `BLT`, `BGT`, `BGE`, `BLE`

If need to *unconditionally* branch, simply use twice the same register with `BEQ`. *e.g.,* BEQ R0, R0, relativeAddress. The unconditional branch is often just shortened as `B`.

Control

The control logic that updates the PC now becomes:



`branch_taken` indicates a branch is taken. This happens when:

- A branch instruction executes;
- and the condition is satisfied.

ALU flags & Conditions

Condition	Bit pattern	Description	ALU flags
EQ	000	equal	Z==1
NE	001	not equal	Z==0
LT	010	signed less than	N!=V
LE	011	signed less or equal	(Z==1) or (N!=V)
GT	100	signed greater than	(Z==0) and (N==V)
GE	101	signed greater or equal	N==V

Remember, the ALU is set to perform a subtraction.

For instance, checking if $\mathtt{A} \geq \mathtt{B}$ involves performing $\mathtt{A} - \mathtt{B}$:

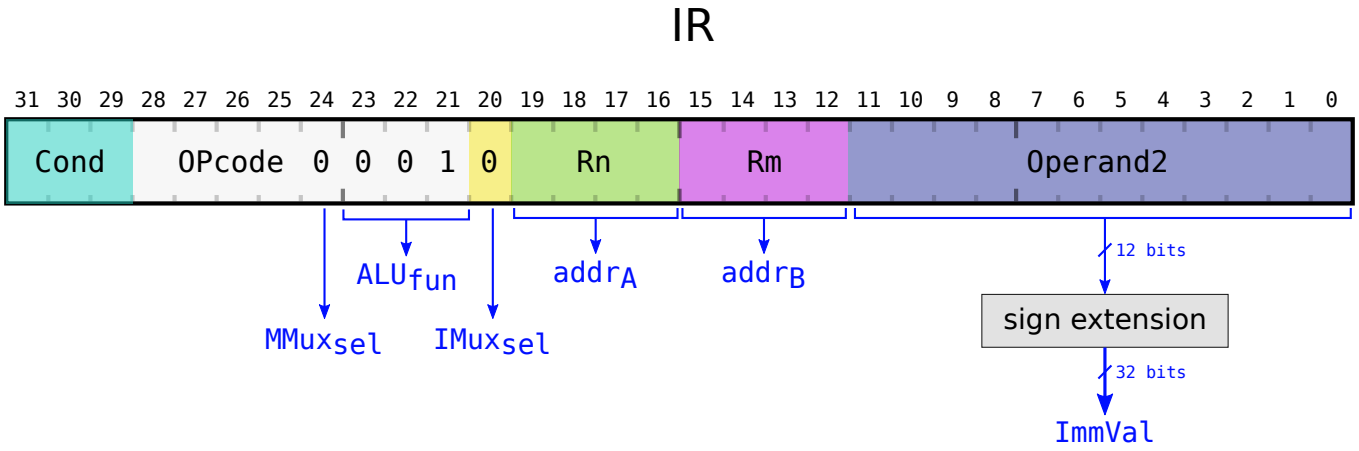
- If no overflow, $\mathtt{A} \geq \mathtt{B}$ if $\mathtt{N} = \mathtt{0}$;
- If overflow, $\mathtt{A} \geq \mathtt{B}$ if $\mathtt{N} = \mathtt{1}$.

e.g., with 4 bits: $\mathtt{A} = 6, \mathtt{B} = -7$

$\mathtt{A} - \mathtt{B} = 6 - (-7) = 6 + 7 = \mathtt{0110}_2 + \mathtt{0111}_2 = \mathtt{1101}_2$

overflow and negative! $\Rightarrow \mathtt{A} \geq \mathtt{B}$.

Branch Instruction Encoding

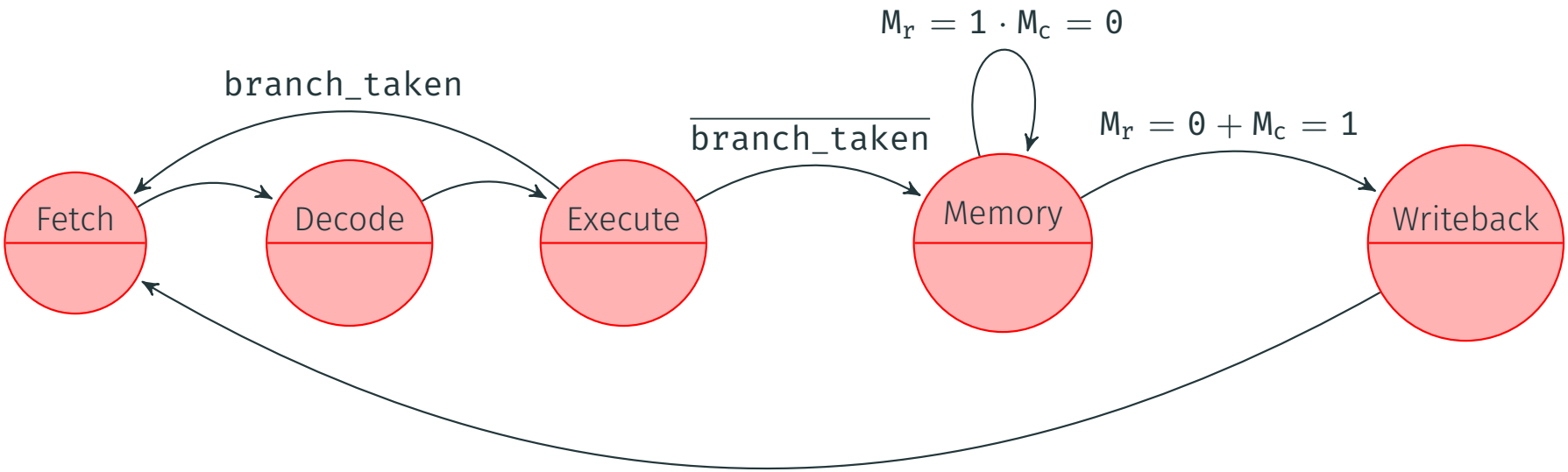


Observe how:

- The ALU is set to perform a subtraction (bits 23-21);
- Bits 31-29 encode the condition (see table from previous slide);
- $\texttt{IMUX}_{\texttt{sel}}$ is always zero since the Operand2 is used for the PC relative address;
- $\texttt{MMUX}_{\texttt{sel}}$ is 0 since this is not a memory operation;
- Bits 15-12 are now used for $\texttt{addr}_{\texttt{B}}$.

Finite State Machine with Branching

Our FSM is now updated as follows:



$\texttt{branch\_taken} = \texttt{BranchInstruction} \cdot \texttt{ConditionTrue}$

`ConditionTrue` is true when the output flags of the ALU matches with the condition expected.

Use of branch instruction

Back to our example:

$$max(a, b) = \begin{cases} \mathtt{a} & \text{if } \mathtt{a} \geq \mathtt{b} \\ \mathtt{b} & \text{otherwise} \end{cases}$$

Assuming `a`, `b` in `R0`,`R1`, and result produced in `R2`. We can use the branch instruction to implement the max function:

```
BLT R0, R1, +8
ADD R2, R0, 0
B +4
ADD R2, R1, 0
```

Why +8,+4?

Final words: Instruction versus Data Memory

In this lecture we have seen that instructions and data use different memories. This is called a Harvard architecture.

It is also possible to design the CPU where both instructions and data lives in the same memory, this is called a von Neumann architecture.

Most modern computers use a single memory for everything, as far the programmer is concerned. Internally, they use two separate memories, known as caches (more on this later in the course).

Conclusion

This lecture has:

- Reviewed the basic digital logic components and;
- Presented the datapath and control of a simple 5-stage RISC machine;

The next lecture will:

- Present the instructions supported by a real processor (ARMv7);
- Show how to write real assembly programs in much more details.