College of Computer and Information Sciences
Department of Computer Science

CSC 220: Computer Organization

Unit 12
CPU programming

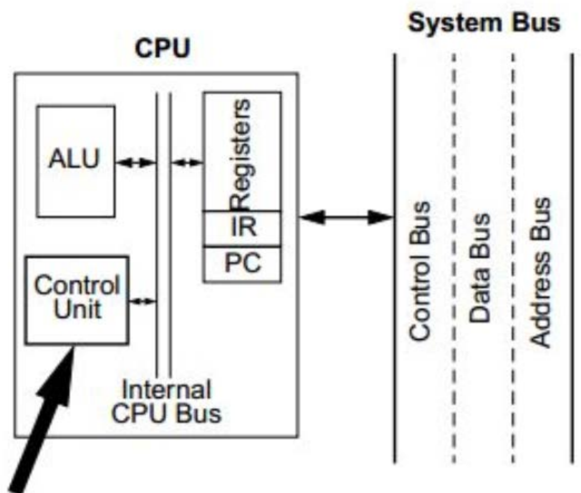
Instruction set architectures

- Last time we built a simple, but complete, datapath.
- The datapath is ultimately controlled by a programmer, so today we'll look at several aspects of this programming in more detail.
  - How programs are executed on processors
  - An introduction to instruction set architectures
  - Example instructions and programs
- Next, we'll see how programs are encoded in a processor. Following that, we'll finish our processor by designing a control unit, which converts our programs into signals for the datapath.

Programming and CPUs

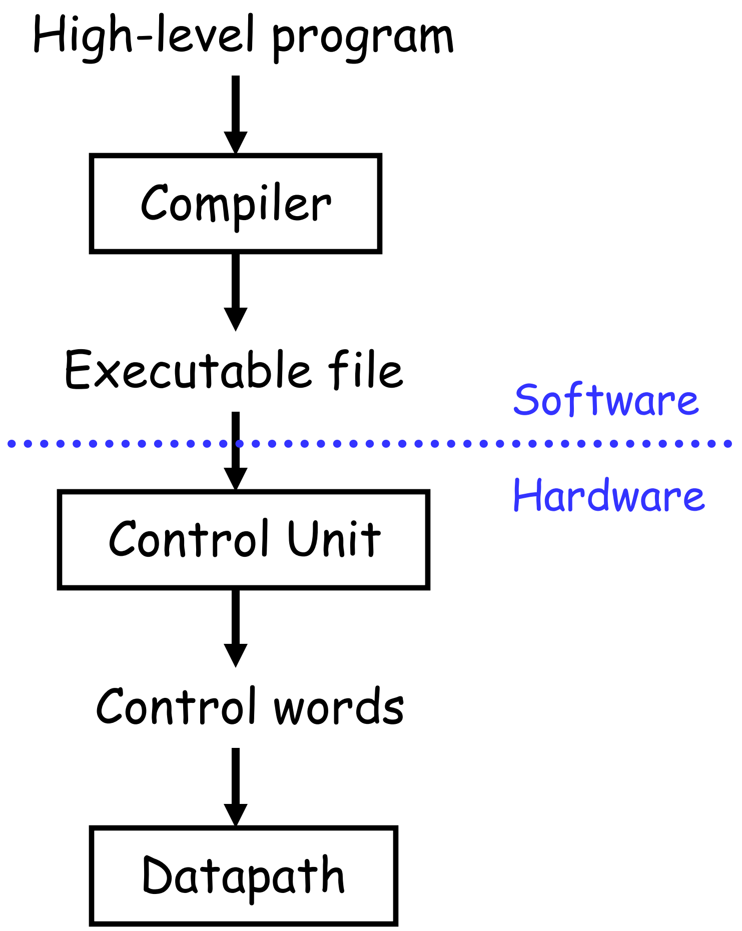
Control units

- From these examples, you can see that different actions are performed when we provide different inputs for the datapath control signals.
- The second question we had was “Who exactly decides which registers are read and written and which ALU function is executed?”
  - In real computers, the datapath actions are determined by the program that’s loaded and running.
  - A control unit is responsible for generating the correct control signals for a datapath, based on the program code.
- We’ll talk about programs and control units next week.



Programming and CPUs

- Programs written in a high-level language like C++ must be compiled to produce an executable program.
- The result is a CPU-specific machine language program. This can be loaded into memory and executed by the processor.
- CS231 focuses on stuff below the dotted blue line, but machine language serves as the interface between hardware and software.



Register transfer language

- Two-character names denote registers, such as R0, R1, DR, or SA.
- Arrows indicate data transfers. For example, we can copy the contents of source register R2 into the destination register R1 in one clock cycle.

$$R1 \leftarrow R2$$

- A conditional transfer is performed only if the Boolean condition in front of the operation is true. Below, we transfer R3 to R2 only when K = 1.

$$K\colon R2 \leftarrow R3$$

- Multiple transfers on the *same* clock cycle are separated by commas.

$$R1 \leftarrow R2,\quad K\colon R2 \leftarrow R3$$

Register transfer operations

- We can apply arithmetic operations to registers.

$$R1 \leftarrow R2 + R3$$
$$R3 \leftarrow R1 - 1$$

- Bitwise logical operations can be expressed. We use special symbols for AND and OR to prevent confusion with arithmetic operations.

$R2 \leftarrow R1 \wedge R2$	bitwise AND
$R3 \leftarrow R0 \vee R1$	bitwise OR

- Finally, we can shift values left or right by one bit. The source register is not modified, and we assume that the shift input is always 0.

$R2 \leftarrow \text{sl } R1$	left shift
$R2 \leftarrow \text{sr } R1$	right shift

Memory

- While memory transfers are similar to register transfers, we usually identify them differently. Specifically, memory to register transfers are called *read* operations, while register to memory transfers are called *write* operations. Both require specification of the memory location to be used (which can be done through a special register or a special bus) and a storage location which will hold the result of a read or which holds the data to be written.

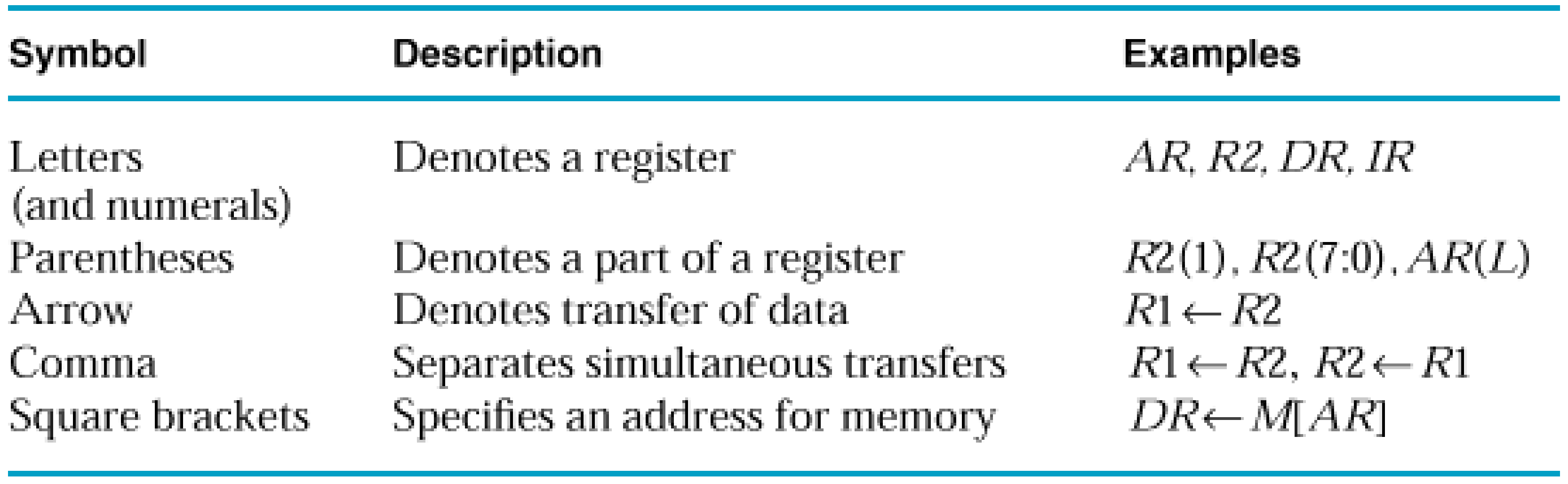
- RTL expressions for a **Read** operation, assuming the use of an address registers:

$$AR \leftarrow address$$
$$DR \leftarrow M[AR]$$

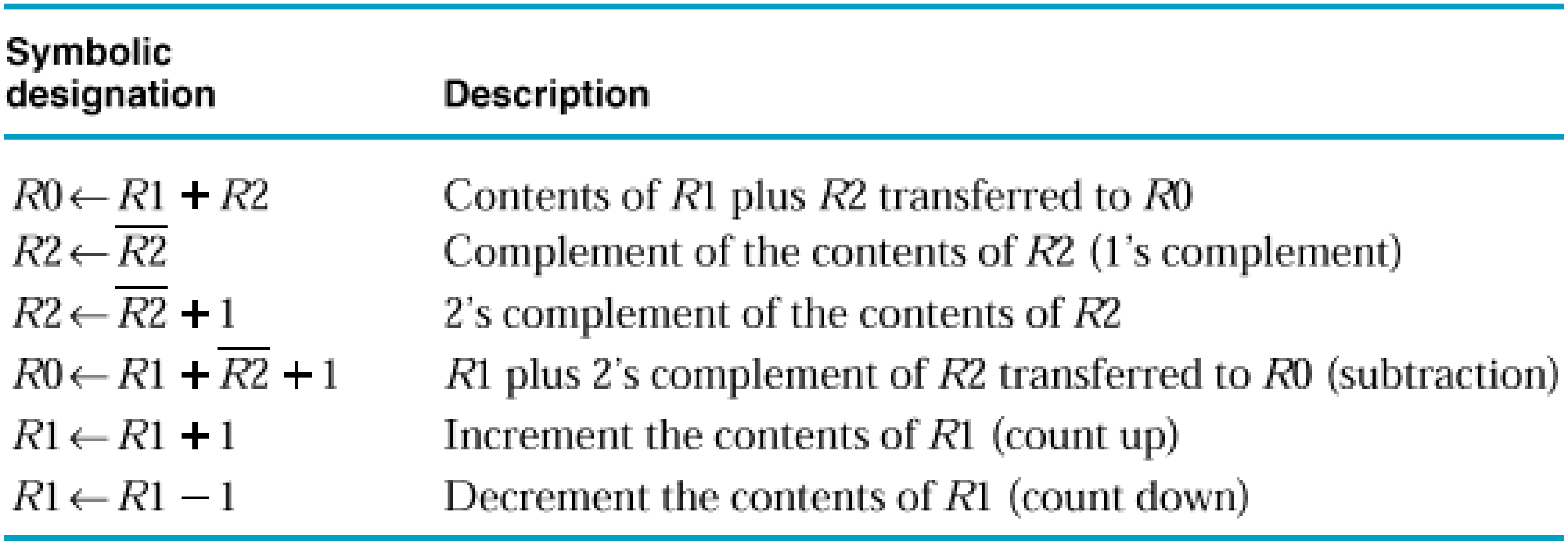
- RTL expressions for a **Write** operation, assuming use of a data register:

$$AR \leftarrow address$$
$$DR \leftarrow value$$
$$M[AR] \leftarrow DR$$

- Register to Memory Transfers are denoted using square brackets surrounding the memory address.
  - e.g. DR ← M[AR] (Read operation)
  - e.g. M[AR] ← DR (Write operation)

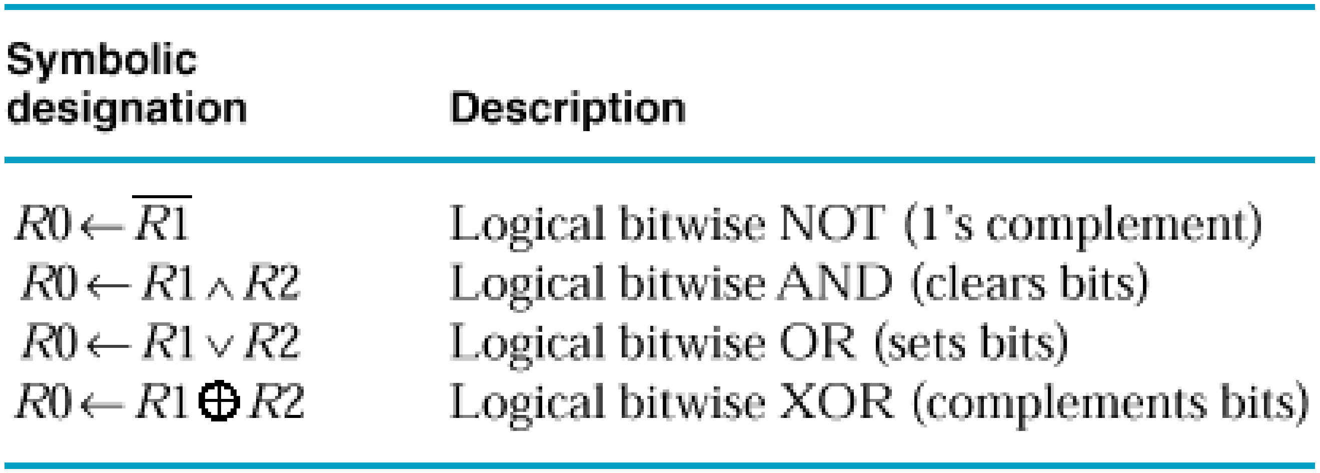


Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*AR, R2, DR, IR*
Parentheses	Denotes a part of a register	$R2(1)$, $R2(7{:}0)$, $AR(L)$
Arrow	Denotes transfer of data	$R1 \leftarrow R2$
Comma	Separates simultaneous transfers	$R1 \leftarrow R2,\ R2 \leftarrow R1$
Square brackets	Specifies an address for memory	$DR \leftarrow M[AR]$



Symbolic designation	Description
$R0 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R0$
$R2 \leftarrow \overline{R2}$	Complement of the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement of the contents of $R2$
$R0 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus 2's complement of $R2$ transferred to $R0$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ (count up)
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ (count down)

Examples of Arithmetic Microoperations



Symbolic designation	Description
$R0 \leftarrow \overline{R1}$	Logical bitwise NOT (1's complement)
$R0 \leftarrow R1 \wedge R2$	Logical bitwise AND (clears bits)
$R0 \leftarrow R1 \vee R2$	Logical bitwise OR (sets bits)
$R0 \leftarrow R1 \oplus R2$	Logical bitwise XOR (complements bits)

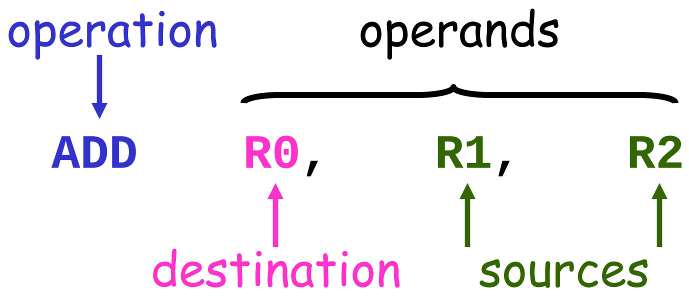
Examples of Logic Microoperations

Assembly and machine languages

- Machine language instructions are sequences of bits in a specific order.
- To make things simpler, people typically use assembly language.
  - We assign "mnemonic" names to operations and operands.
  - There is (almost) a one-to-one correspondence between these mnemonics and machine instructions, so it is very easy to convert assembly programs to machine language.
- We'll use assembly code this today to introduce the basic ideas, and switch to machine language next time.

Data manipulation instructions

- Data manipulation instructions correspond to ALU operations.
- For example, here is a possible addition instruction, and its equivalent using our register transfer notation:



Register transfer instruction:

$$R0 \leftarrow R1 + R2$$

- This is similar to a high-level programming statement like

```
R0 = R1 + R2
```

- Here, all of the operands are registers.

More data manipulation instructions

- Here are some other kinds of data manipulation instructions.

```
NOT   R0, R1            R0 ← R1'
ADD   R3, R3, #1        R3 ← R3 + 1
SUB   R1, R2, #5        R1 ← R2 - 5
```

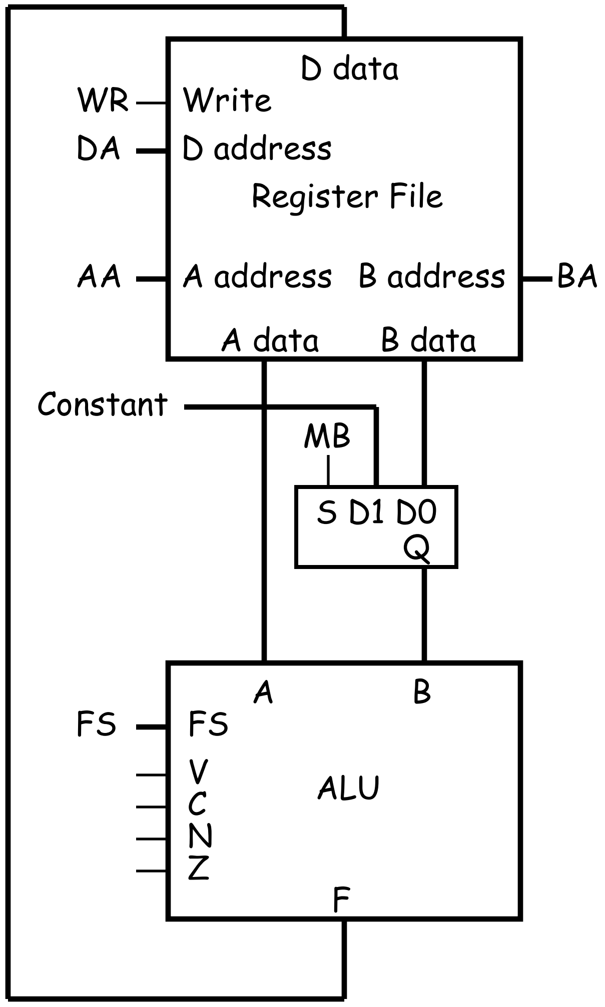
- Some instructions, like the NOT, have only one operand.
- In addition to register operands, constant operands like 1 and 5 are also possible. Constants are denoted with a hash mark in front.

Relation to the datapath

- These instructions reflect the design of our datapath from last week.
- There are at most two source operands in each instruction, since our ALU has just two inputs.
- The two sources can be two registers, or one register and one constant.
- More complex operations like

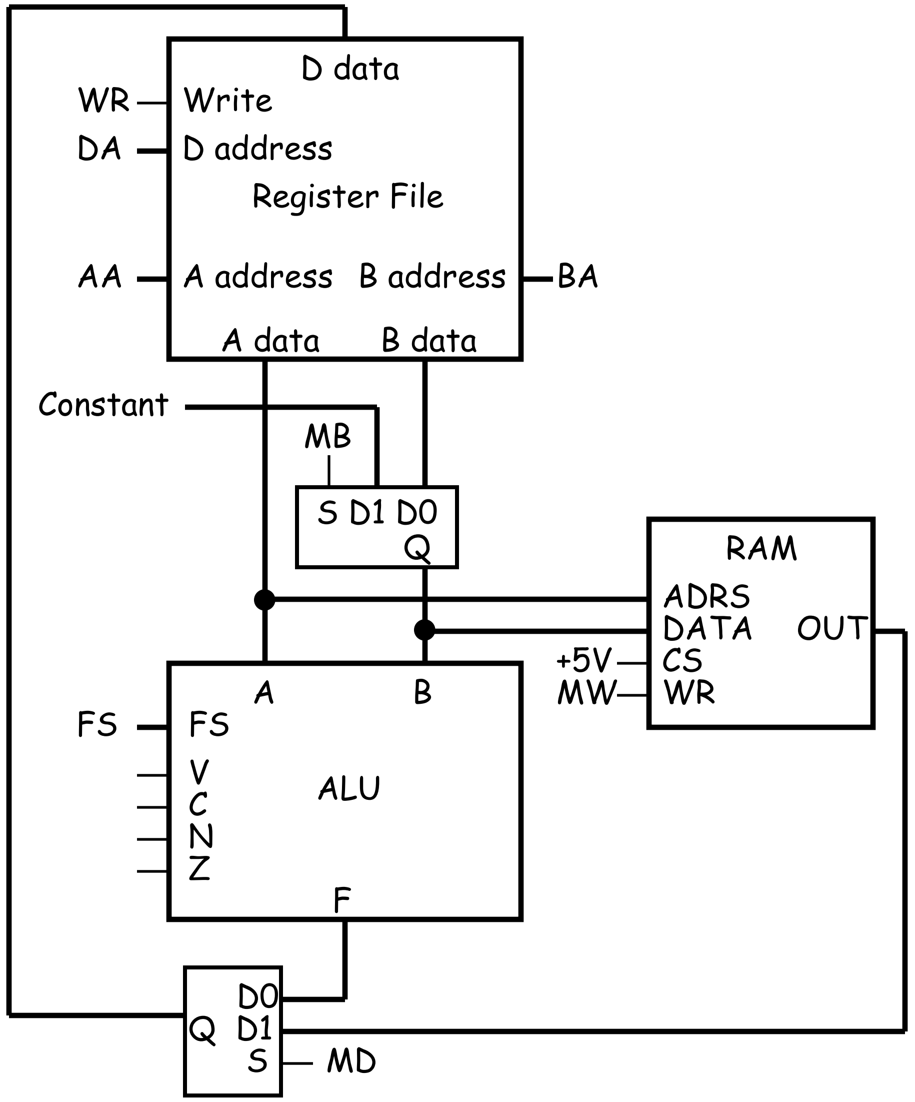
```
R0 ← R1 + R2 - 3
```

  must be broken down into several lower-level instructions.
- Instructions have just one destination operand, which must be a register.



What about RAM?

- Recall that our ALU has direct access only to the register file.
- RAM contents must be copied to the registers before they can be used as ALU operands.
- Similarly, ALU results must go through the registers before they can be stored into memory.
- We rely on data movement instructions to transfer data between the RAM and the register file.

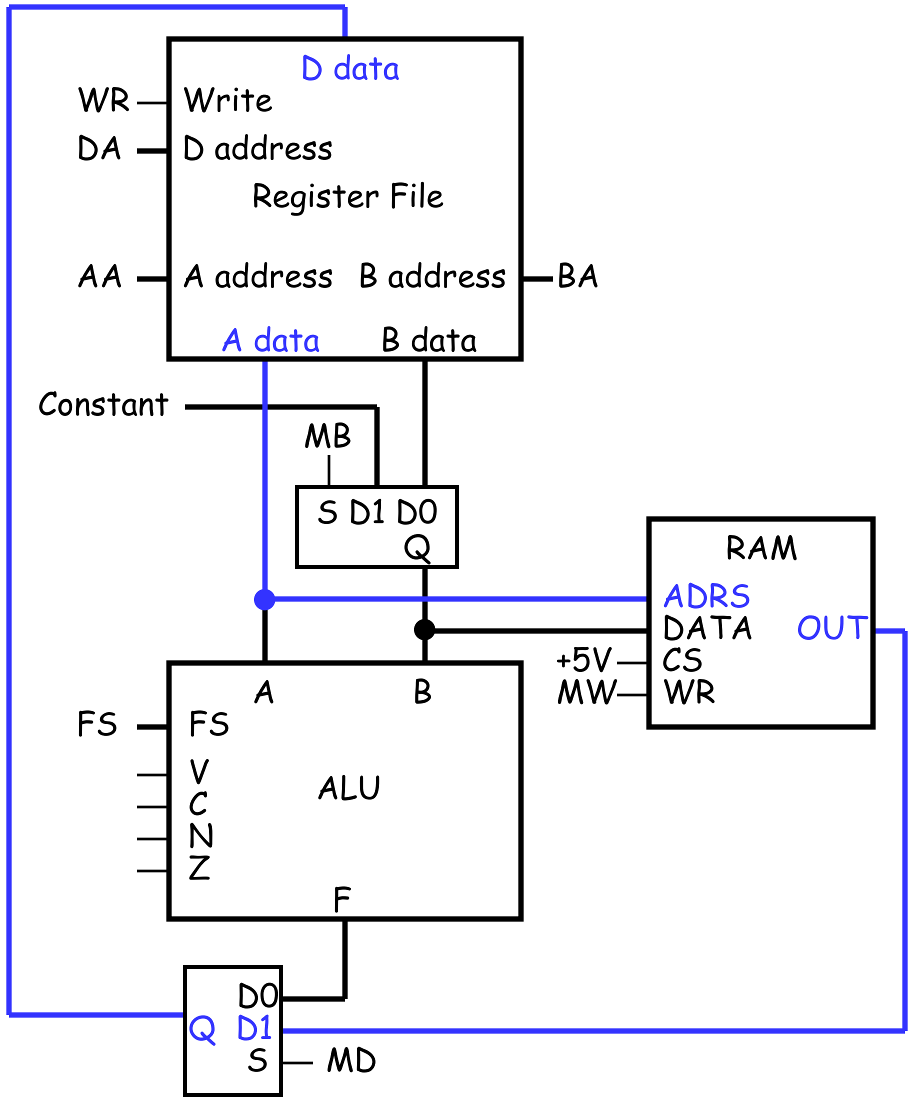


Loading a register from RAM

- A load instruction copies data *from* a RAM address *to* one of the registers.

  `LD R1,(R3)` **`R1 ← M[R3]`**

- Remember in our datapath, the RAM address must come from one of the registers—in the example above, R3.
- The parentheses help show which register operand holds the memory address.

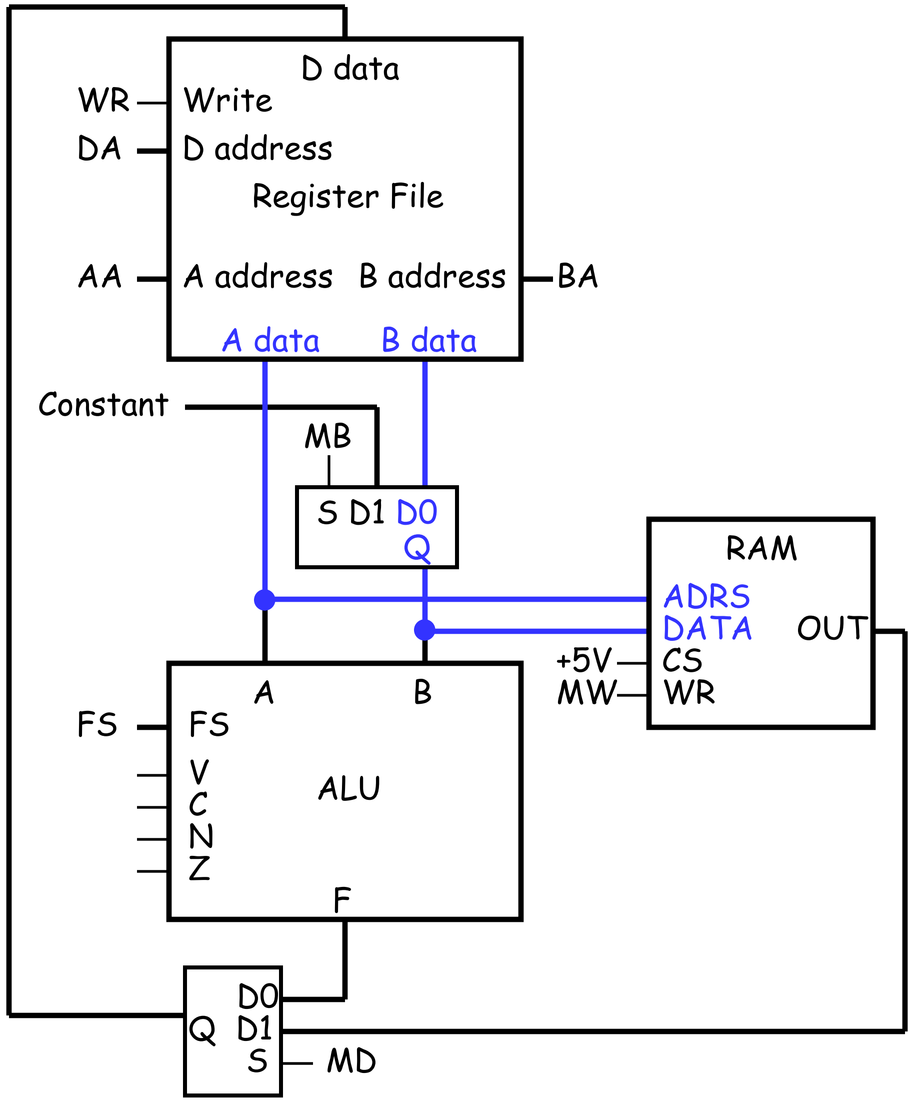


Storing a register to RAM

- A store instruction copies data *from* a register *to* an address in RAM.

  ST (R3),R1   M[R3] ← R1

- One register specifies the RAM address to write to—in the example above, R3.
- The other operand specifies the actual data to be stored into RAM—R1 above.

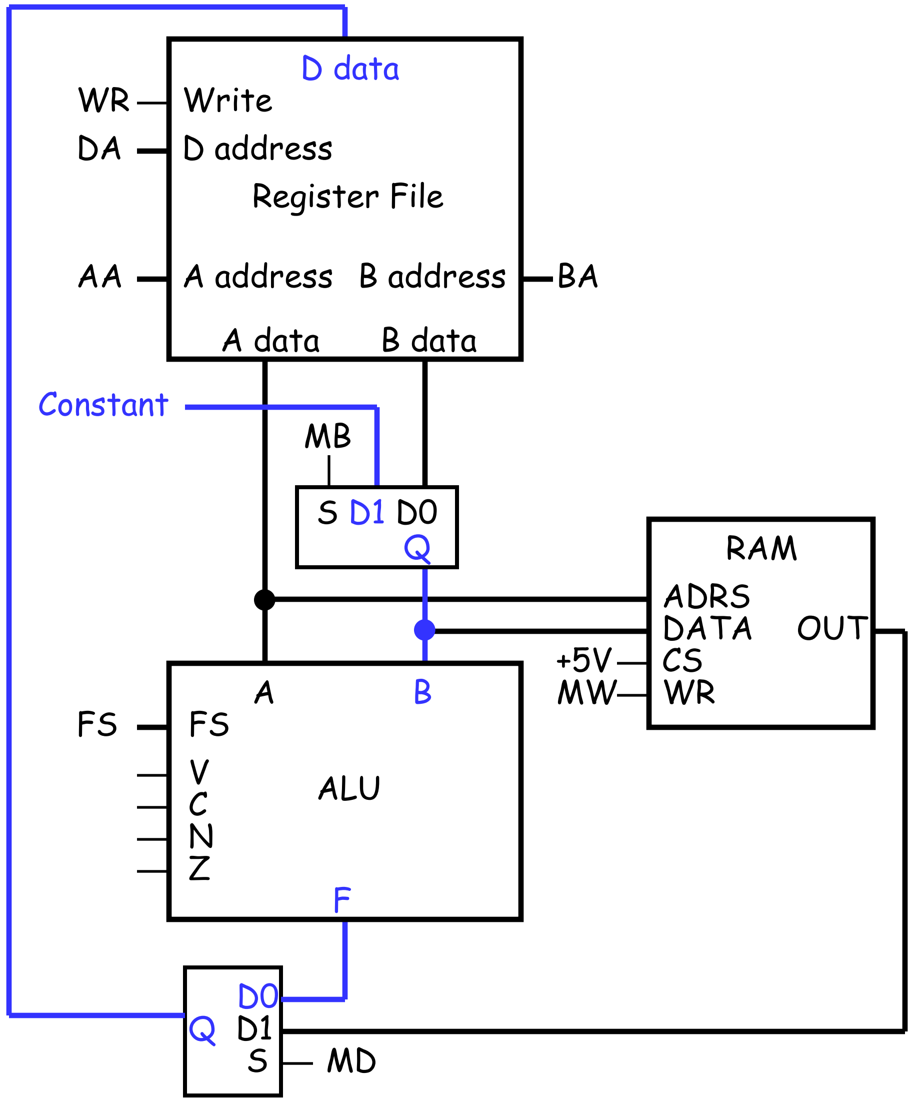


Loading a register with a constant

- With our datapath, it's also possible to load a constant into the register file:

```
LD R1, #0          R1 ← 0
```

- Our example ALU has a "transfer B" operation (FS=10000) which lets us pass a constant up to the register file.
- This gives us an easy way to initialize registers.

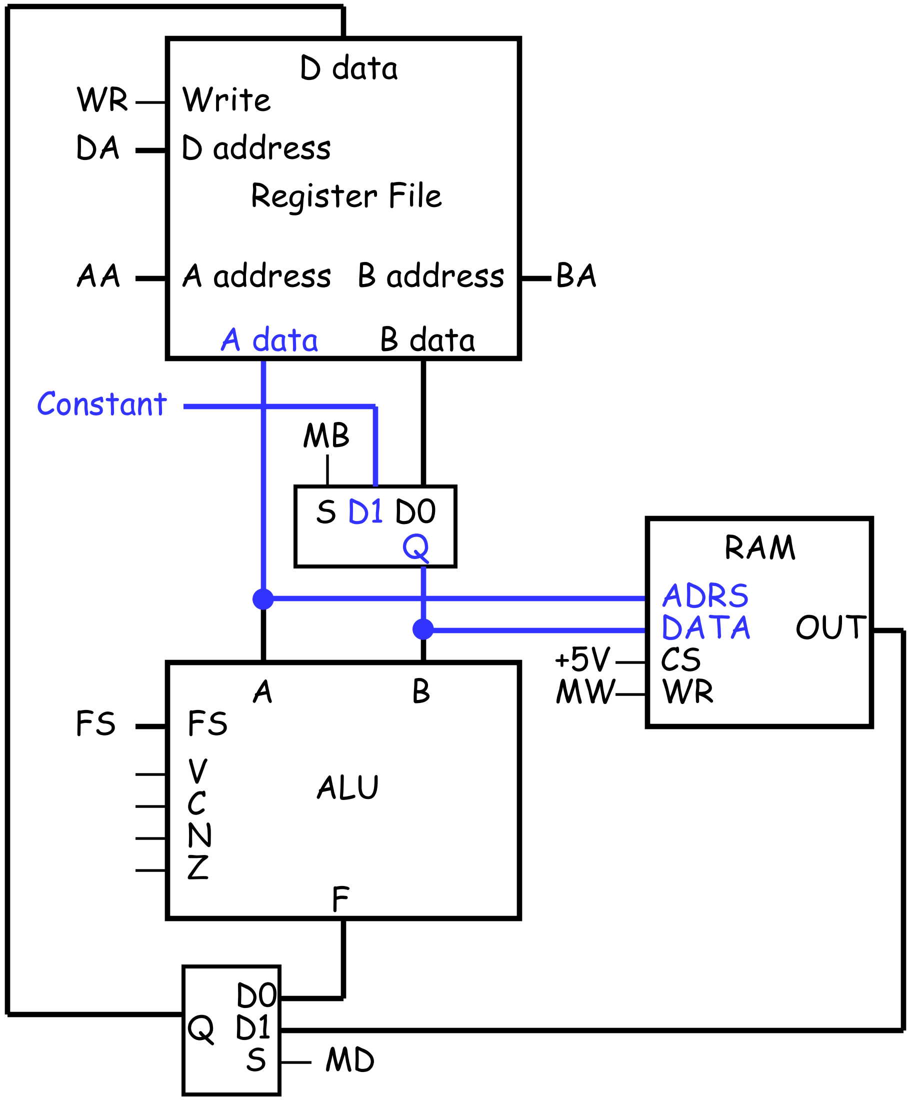


Storing a constant to RAM

- And you can store a constant value directly to RAM too:

  ST (R3), #0   M[R3] ← 0

- This provides an easy way to initialize memory contents.



The # and ( ) are important!

- We've seen several statements containing the # or ( ) symbols. These are ways of specifying different addressing modes.
- The addressing mode we use determines which data are actually used as operands:

```
LD   R0, #1000              // R0 ← 1000
LD   R0,  1000              // R0 ← M[1000]
```

```
LD   R3, R0                 // R3 ← R0
LD   R3, (R0)               // R3 ← M[R0]
```

- The design of our datapath determines which addressing modes we can use.
  - The second example above wouldn't work in our datapath. Why not?
- We'll talk about addressing modes in more detail next lecture.

A small example

- Here's an example register-transfer operation.

```
M[1000] ← M[1000] + 1
```

- This is the assembly-language equivalent:

```
LD  R0, #1000        // R0 ← 1000
LD  R3, (R0)         // R3 ← M[1000]
ADD R3, R3, #1       // R3 ← R3 + 1
ST  (R0), R3         // M[1000] ← R3
```

- An awful lot of assembly instructions are needed!
  - For instance, we have to load the memory address 1000 into a register first, and then use that register to access the RAM.
  - This is due to our relatively simple datapath design, which only allows register and constant operands to the ALU.
  - Later on, mostly in CS232, you'll see why this can be a good thing.

Control flow instructions

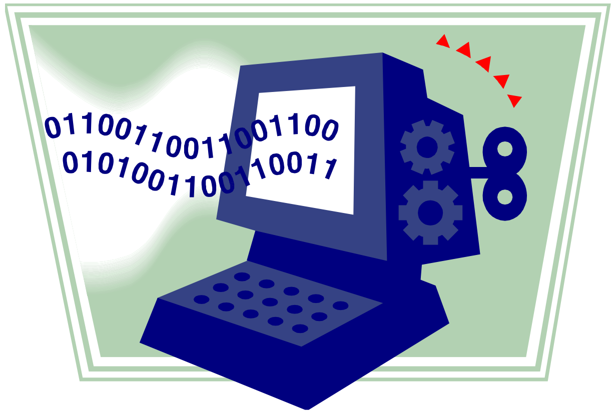
- Programs consist of a lot of sequential instructions, which are meant to be executed one after another.
- Thus, programs are stored in memory so that:
  - Each program instruction occupies one address.
  - Instructions are stored one after another.

```
768:    LD  R0, #1000      // R0 ← 1000
769:    LD  R3, (R0)       // R3 ← M[1000]
770:    ADD R3, R3, #1     // R3 ← R3 + 1
771:    ST  (R0), R3       // M[1000] ← R3
```

- A program counter (PC) keeps track of the current instruction address.
  - Ordinarily, the PC just increments after executing each instruction.
  - But sometimes we need to change this normal sequential behavior, with special control flow instructions.

Instruction encoding

- We've already seen some important aspects of processor design.
  - A datapath contains an ALU, registers and memory.
  - Programmers and compilers use instruction sets to issue commands.
- Now let's complete our processor with a control unit that converts assembly language instructions into datapath signals.
  - Today we'll see how control units fit into the big picture, and how assembly instructions can be represented in a binary format.
  - On Wednesday we'll show all of the implementation details for our sample datapath and assembly language.

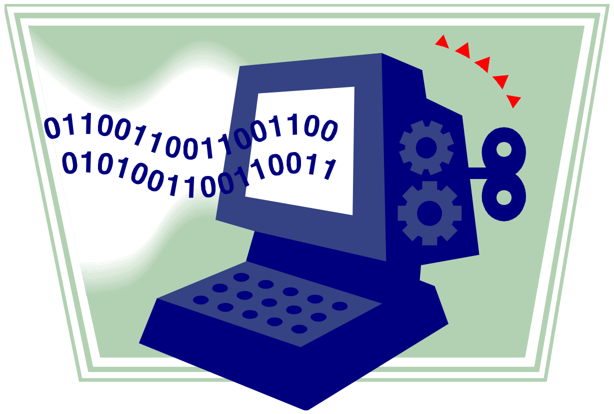


note

- Machine language is the interface between software and processors.
- High-level programs must be translated into machine language before they can be run.
- There are three main categories of instructions.
  - Data manipulation operations, such as adding or shifting
  - Data transfer operations to copy data between registers and RAM
  - Control flow instructions to change the execution order
- Instruction set architectures depend highly on the host CPU's design.
  - Today we saw instructions that would be appropriate for our datapath from last week.
  - On Monday we'll look at some other possibilities.

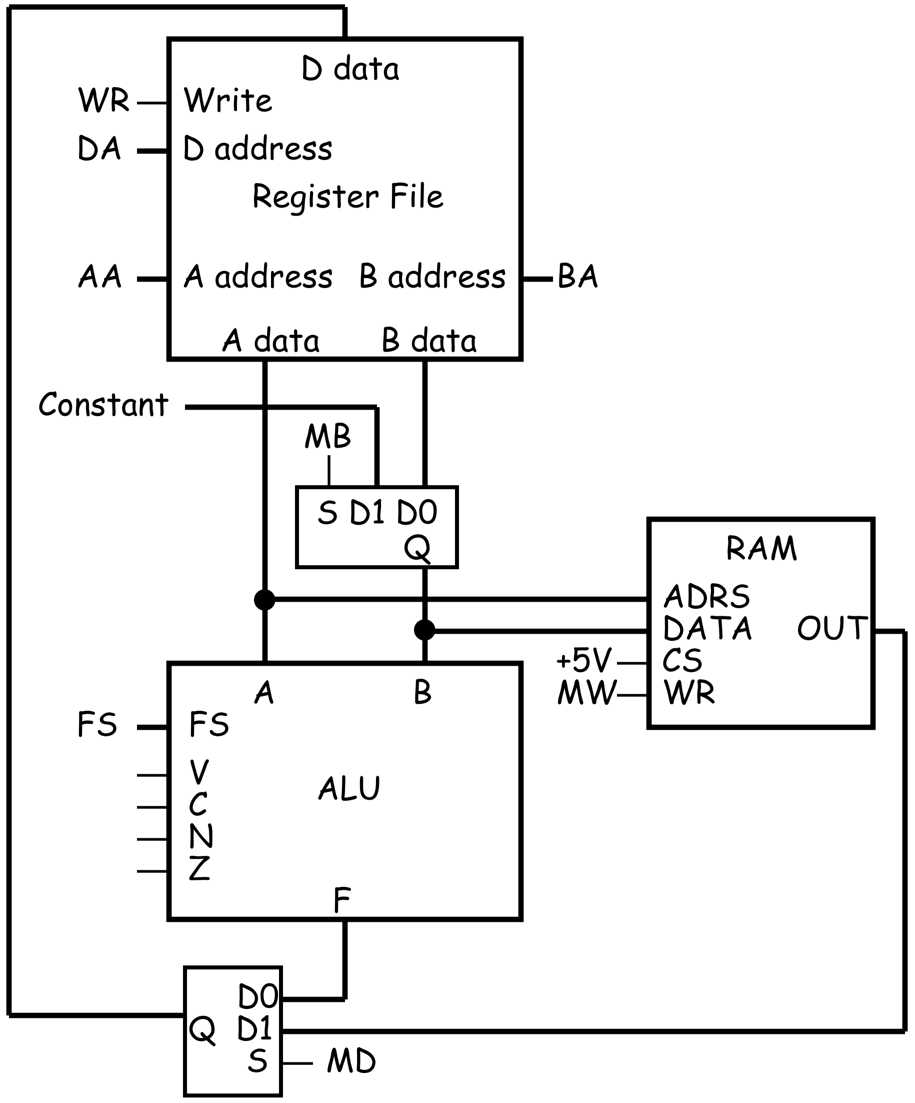
Instruction encoding

- We've already seen some important aspects of processor design.
  - A datapath contains an ALU, registers and memory.
  - Programmers and compilers use instruction sets to issue commands.
- Now let's complete our processor with a control unit that converts assembly language instructions into datapath signals.
  - Today we'll see how control units fit into the big picture, and how assembly instructions can be represented in a binary format.
  - On Wednesday we'll show all of the implementation details for our sample datapath and assembly language.

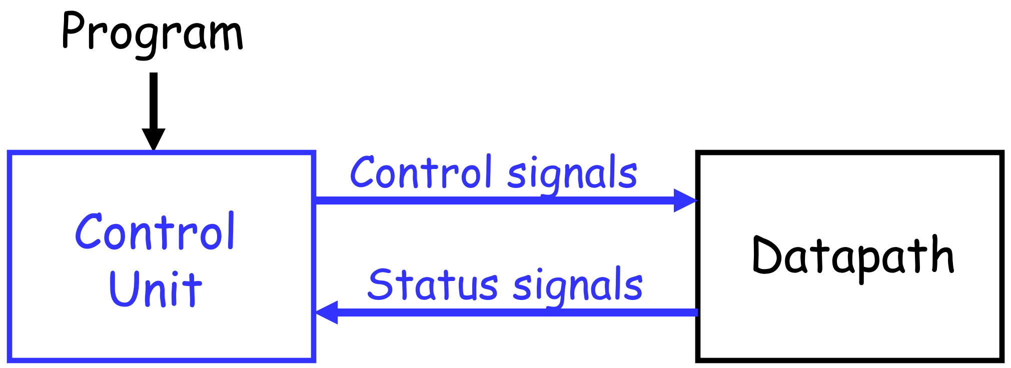


Review: Datapath

- Recall that our ALU has direct access only to the register file.
- RAM contents must be copied to the registers before they can be used as ALU operands.
- Similarly, ALU results must go through the registers before they can be stored into memory.
- We rely on data movement instructions to transfer data between the RAM and the register file.



Block diagram of a processor

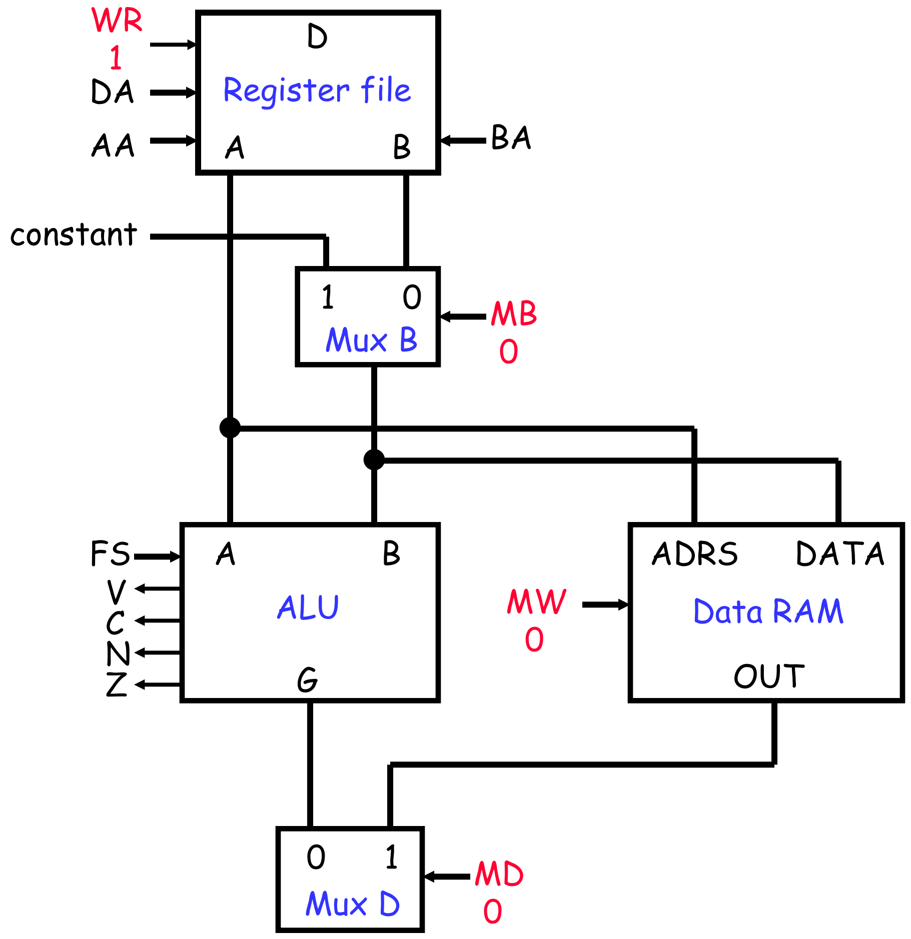


- The control unit connects programs with the datapath.
  - It converts program instructions into control words for the datapath, including **signals WR, DA, AA, BA, MB, FS, MW, MD**.
  - It executes program instructions in the correct sequence.
  - It generates the "constant" input for the datapath.
- The datapath also sends information back to the control unit. For instance, the ALU status bits V, C, N, Z can be inspected by branch instructions to alter a program's control flow.

Register format ALU operations

ADD R1, R2, R3

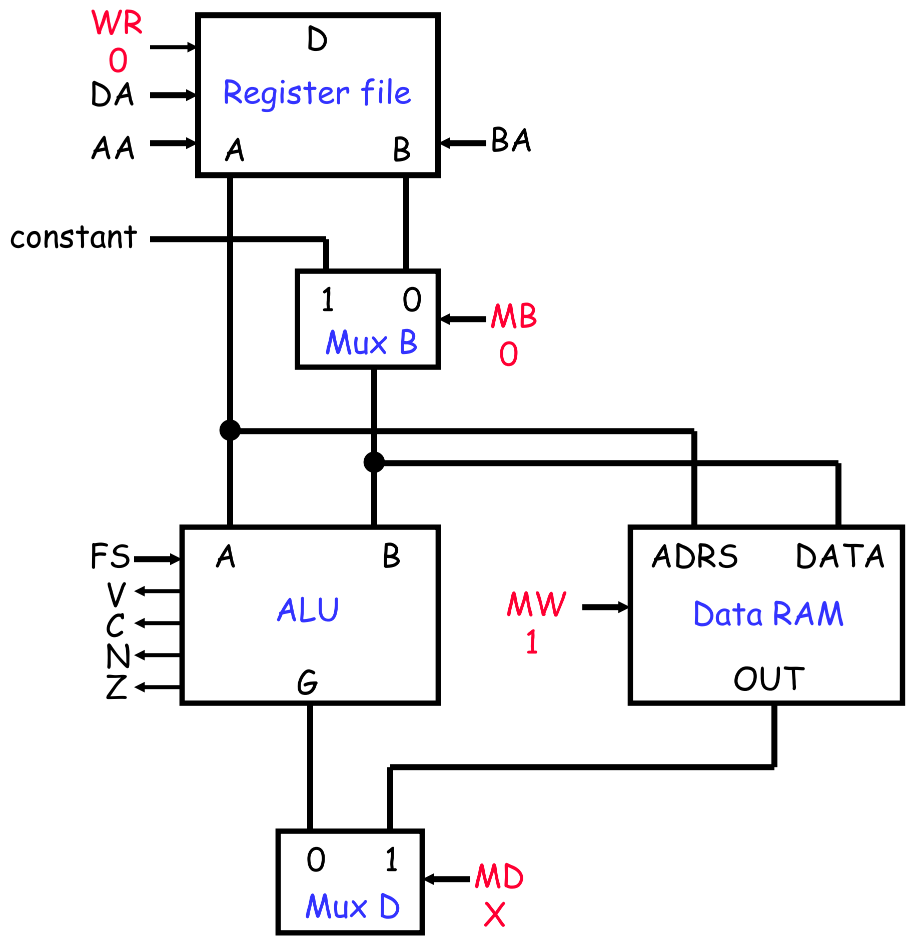
- All register format ALU operations need the same values for the following control signals:
- MB = 0, because all operands come from the register file.
- MD = 0 and WR = 1, to save the ALU result back into a register.
- MW = 0 since RAM is not modified.



Memory write operations

`ST (R0), R1`

- All memory write operations need the same values for the following control signals:
- MB = 0, because the data to write comes from the register file.
- MD = X and WR = 0, since none of the registers are changed.
- MW = 1, to update RAM.


WR
0
DA
AA
D
Register file
A
B
BA
constant
1
0
Mux B
MB
0
FS
V
C
N
Z
A
B
ALU
G
MW
1
ADRS
DATA
Data RAM
OUT
0
1
Mux D
MD
X