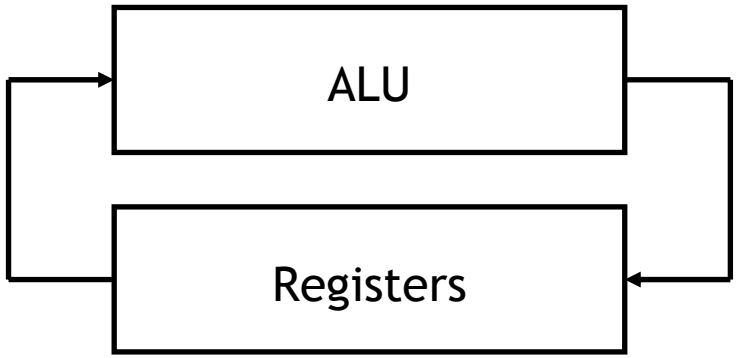
Datapaths

- The rest of the semester focuses on computer architecture—combining combinational and sequential circuit elements to produce a computer.
- We'll spend most of our time on the central processor unit or CPU.
  - The datapath does all of the actual data processing.
  - A control unit tells the datapath what to do and when to do it.
  - An instruction set is the programmer's interface to the CPU.
- Today will be devoted to the datapath.

Keep it simple!

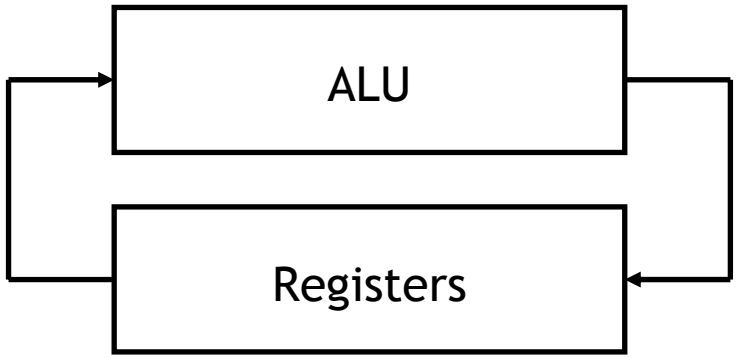
- *Abstraction* is very helpful in understanding processors.
  - Although we studied how devices like registers and muxes are built, we don't need that level of detail here.
  - You should focus more on *what* these component devices are doing, and less on *how* they work.
- Otherwise it's easy to get bogged down in the details, and datapath and control units can be a little intimidating.

A basic CPU model



- A processor is just one big sequential circuit.
  - Some registers are used to store values, which form the state.
  - An ALU performs various operations on the stored data.
- CS375 and other theory courses talk more about state machines and how they can model computers.

Register transfers



- The processor just moves data between registers, possibly with some ALU computations.
- To describe this data movement and computation more precisely, we'll introduce a register transfer language.
  - The objects in the language are *registers*.
  - The basic operations are *transfers*, in which data is copied from one register to another.
  - We can also perform arithmetic operations on data being transferred.

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.

$$K: R2 \leftarrow R3$$

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

$$R1 \leftarrow R2,\ \ K: R2 \leftarrow R3$$

Register transfer operations

- We can apply arithmetic operations to registers.

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

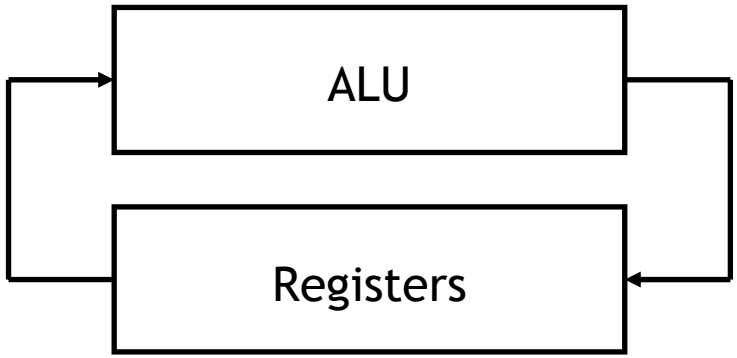
- Bitwise logical operations can be expressed. We use special symbols for AND and OR to prevent confusion with arithmetic operations.

$R2 \leftarrow R1 \wedge R2$	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.

$R2 \leftarrow sl\ R1$	left shift
$R2 \leftarrow sr\ R1$	right shift

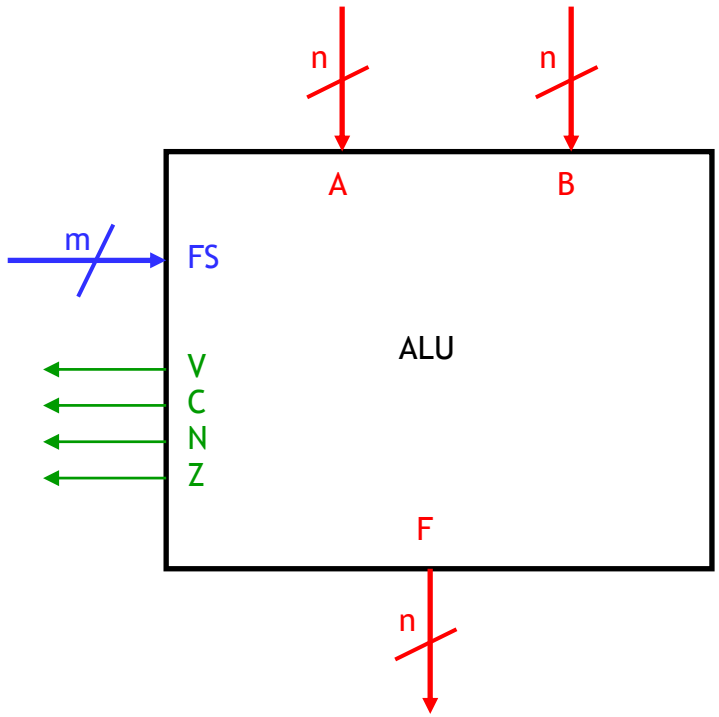
What's inside the datapath?



- Now we'll look in detail at the processor's datapath, which is responsible for doing all of the dirty work.
  - An ALU does computations, as we've seen before.
  - A limited set of registers serves as fast temporary storage.
  - A larger, but slower, random-access memory is also available.

The all-important ALU

- The main job of a central processing unit is to "process," or to perform computations... Remember the ALU from a few weeks ago?
- We'll use the following general block symbol for the ALU.
  - A and B are two *n*-bit numeric inputs.
  - FS is an *m*-bit function select code, which picks one of 2^m functions.
  - The *n*-bit result is called F.
  - Several status bits provide more information about the output F.
    - V = 1 for signed overflow.
    - C is the carry out.
    - N = 1 if the result is negative.
    - Z = 1 if the result is 0.
- This should all look familiar from MP2!



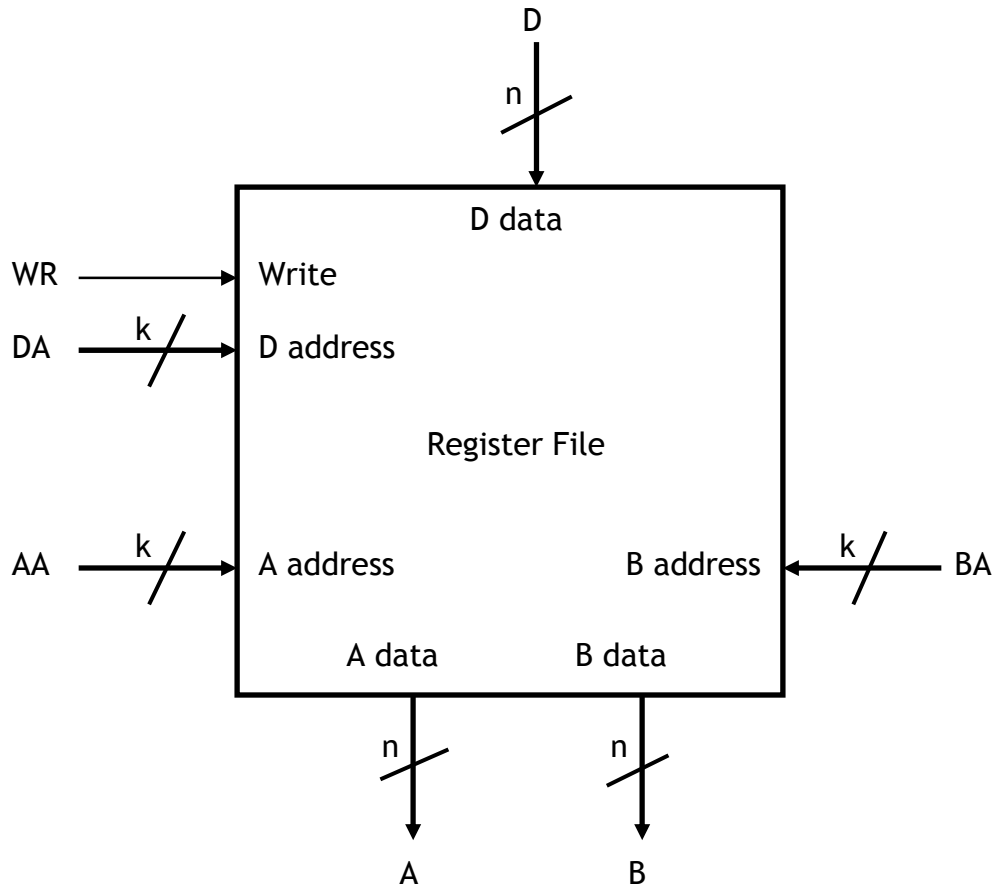
ALU functions

- For concrete examples, we'll use the ALU as it's presented in the textbook.
- The table of operations on the right is taken from page 373.
- The function select code FS is 5 bits long, but there are only 15 different functions here.
- We use different symbols for AND and OR to avoid confusion with arithmetic operations.

FS	Operation
00000	F = A
00001	F = A + 1
00010	F = A + B
00011	F = A + B + 1
00100	F = A + B'
00101	F = A + B' + 1
00110	F = A − 1
00111	F = A
01000	F = A $\wedge$ B (AND)
01010	F = A $\vee$ B (OR)
01100	F = A $\oplus$ B
01110	F = A'
10000	F = B
10100	F = sr B (shift right)
11000	F = sl B (shift left)

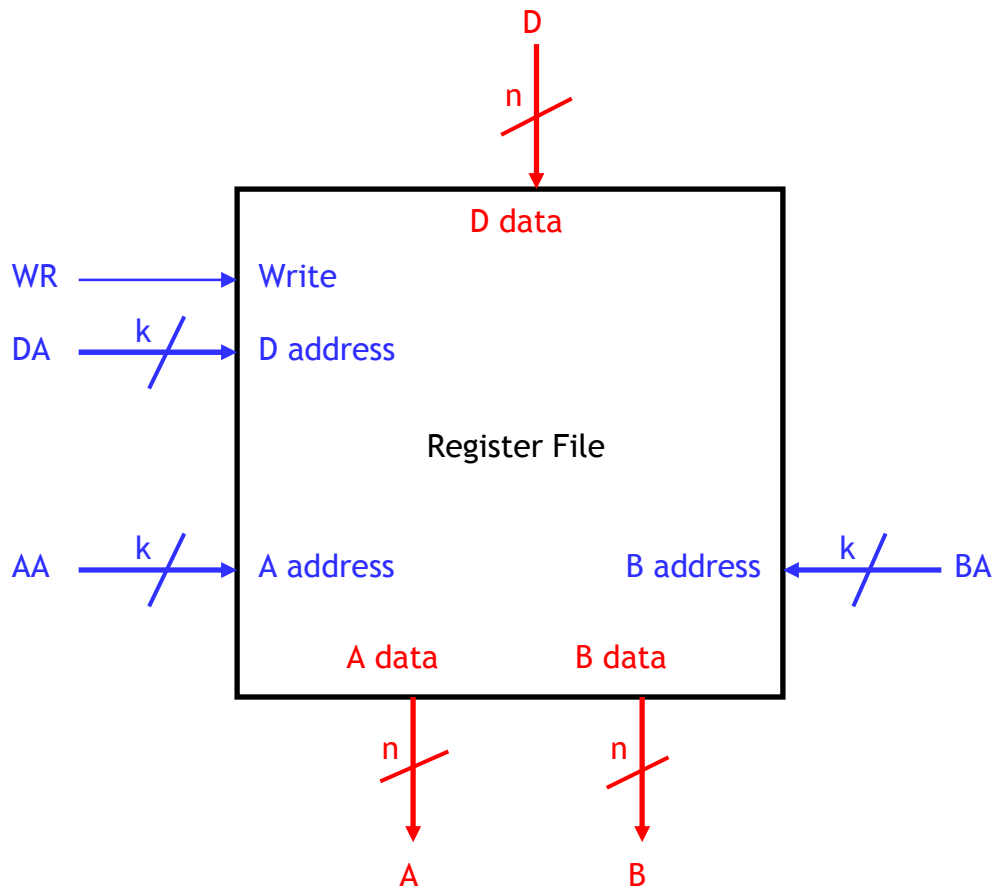
Register files

- As we mentioned already, the ALU inputs and outputs will come from and go to a set of registers.
- Here is a block symbol for a $2^k \times n$ register file.
  - There are 2^k registers, so each register address is k bits long.
  - Each register contains an n-bit value, so the data inputs and outputs have to be n bits wide.



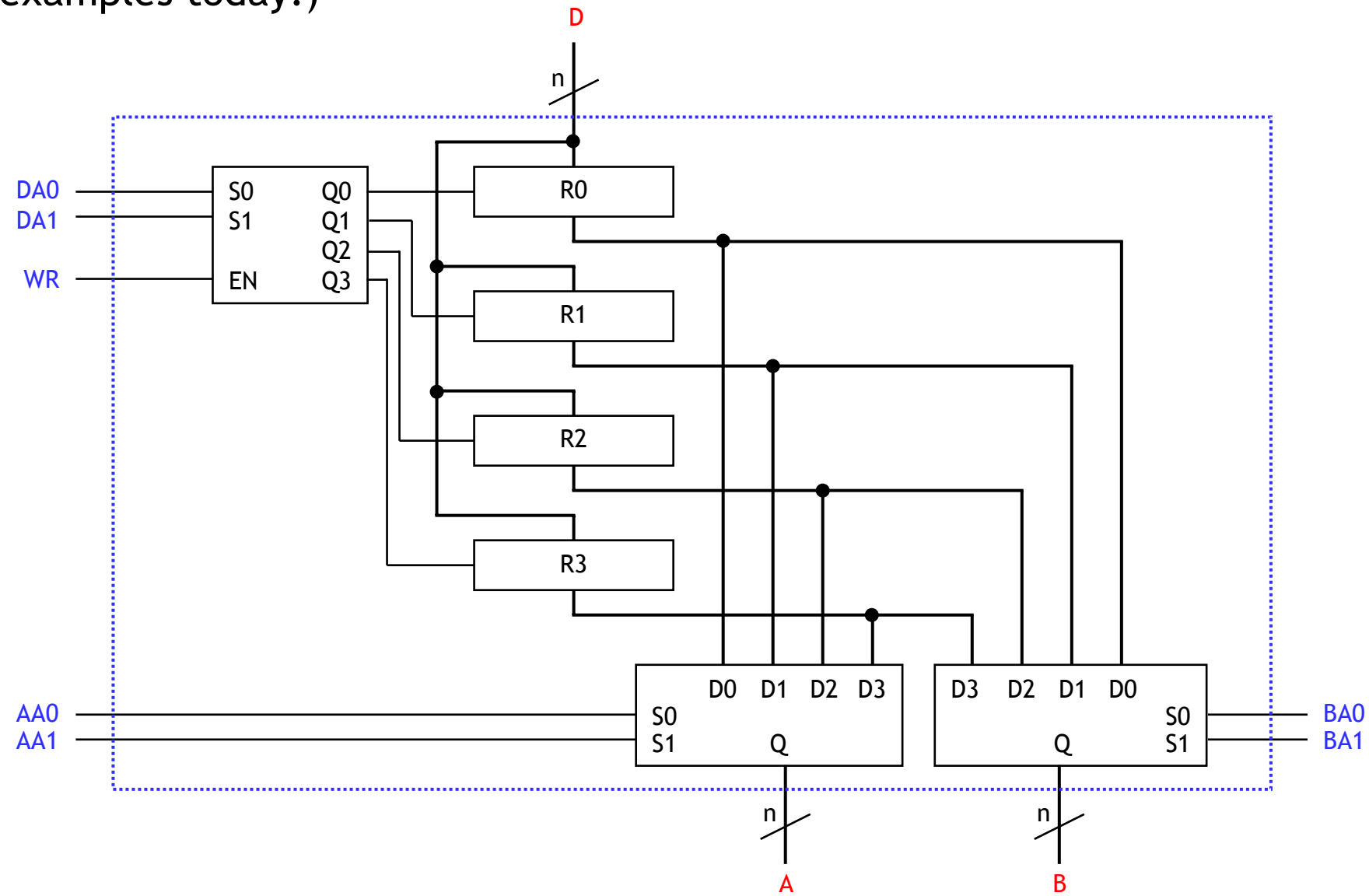
Accessing the register file

- You can read *two* registers at once by supplying the AA and BA inputs. The data appears on the A and B outputs.
- You can write to a register by using the DA and D inputs, and setting WR = 1.
- These are registers so there must be a clock signal, even though we usually don't show it in diagrams.
  - You can read from the register file at any time.
  - Data is written only on the positive edge of the clock.

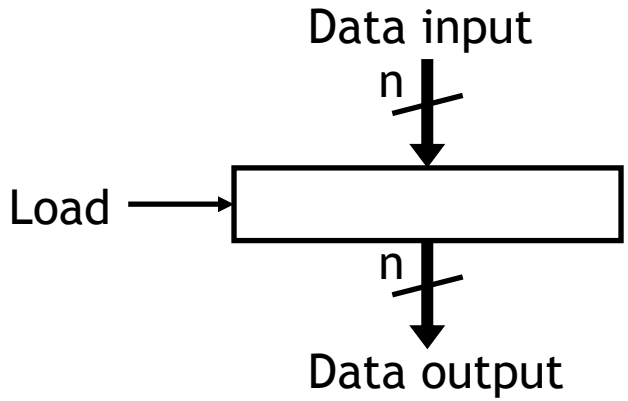


What's inside the register file

- Here's a 4 × *n* register file. (We'll assume a 4 × *n* register file for all our examples today.)



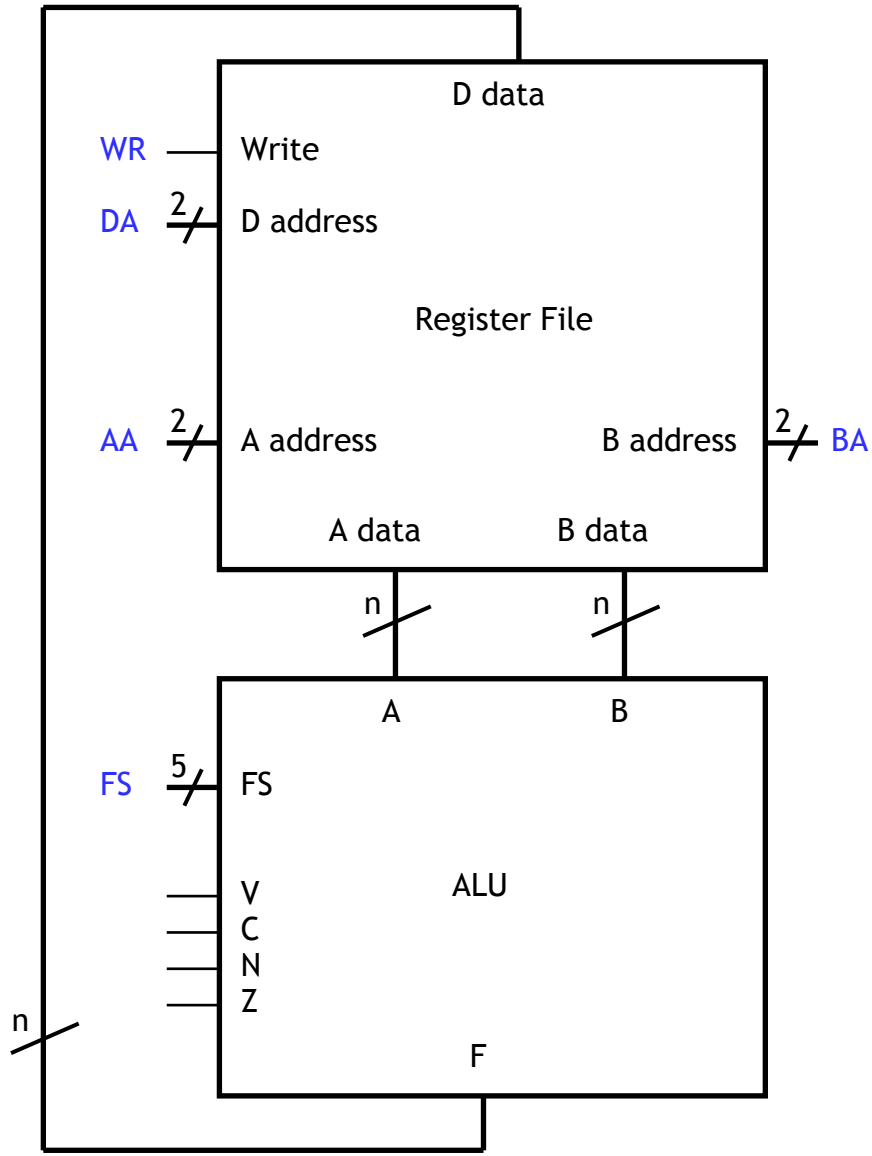
Explaining the register file



- This block diagram represents an *n*-bit register.
  - When Load = 1, the data input is stored into the register.
  - When Load = 0, the register just keeps its current value.
- The 2-to-4 decoder selects one of the four registers for writing. If WR = 1, the decoder will be enabled and one of the Load signals will be active.
- The *n*-bit 4-to-1 muxes select the two register file outputs A and B, based on the inputs AA and BA.

My first datapath

- So here is the most basic datapath.
  - The ALU's two data inputs come from the register file.
  - The ALU computes a result, which is saved back to the registers.
- WR, DA, AA, BA and FS are control signals. Their values determine the exact actions taken by the datapath—which registers are used and for what operation.
- Remember that many of the signals here are actually multi-bit values!

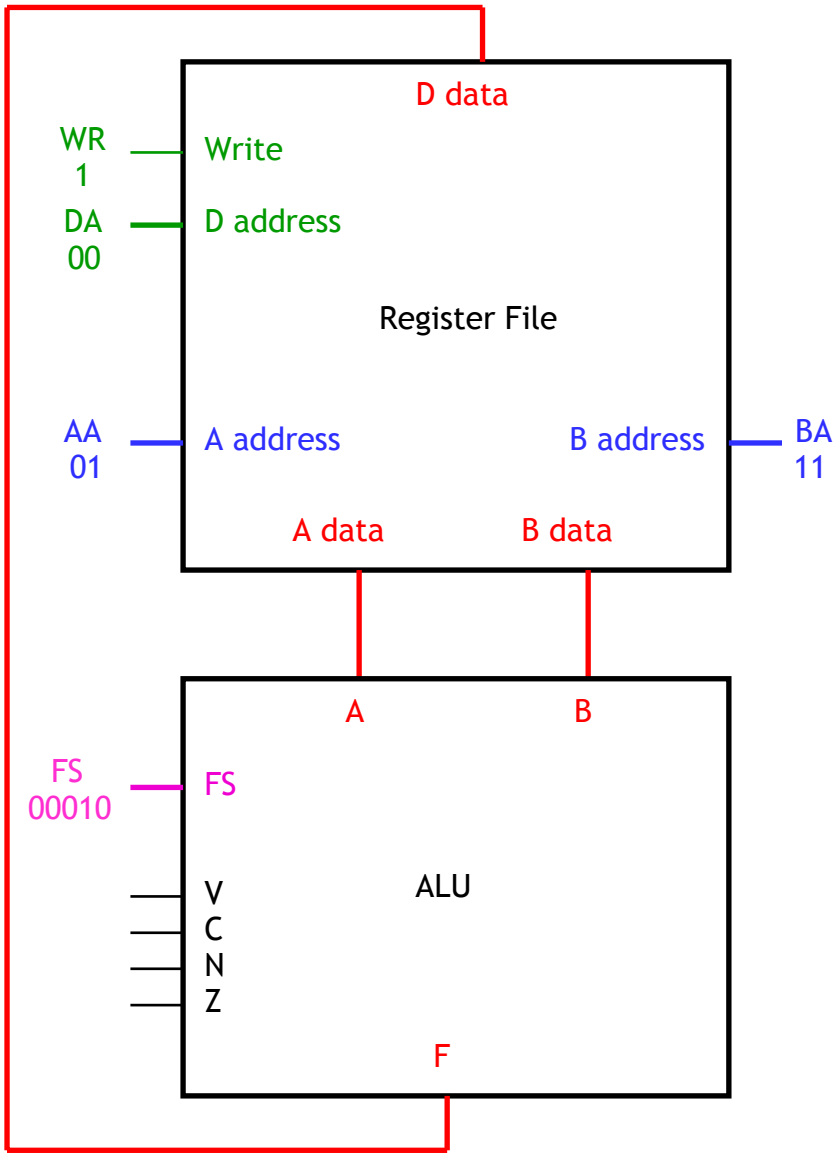


An example computation

- Let's look at the proper control signals for the operation below.

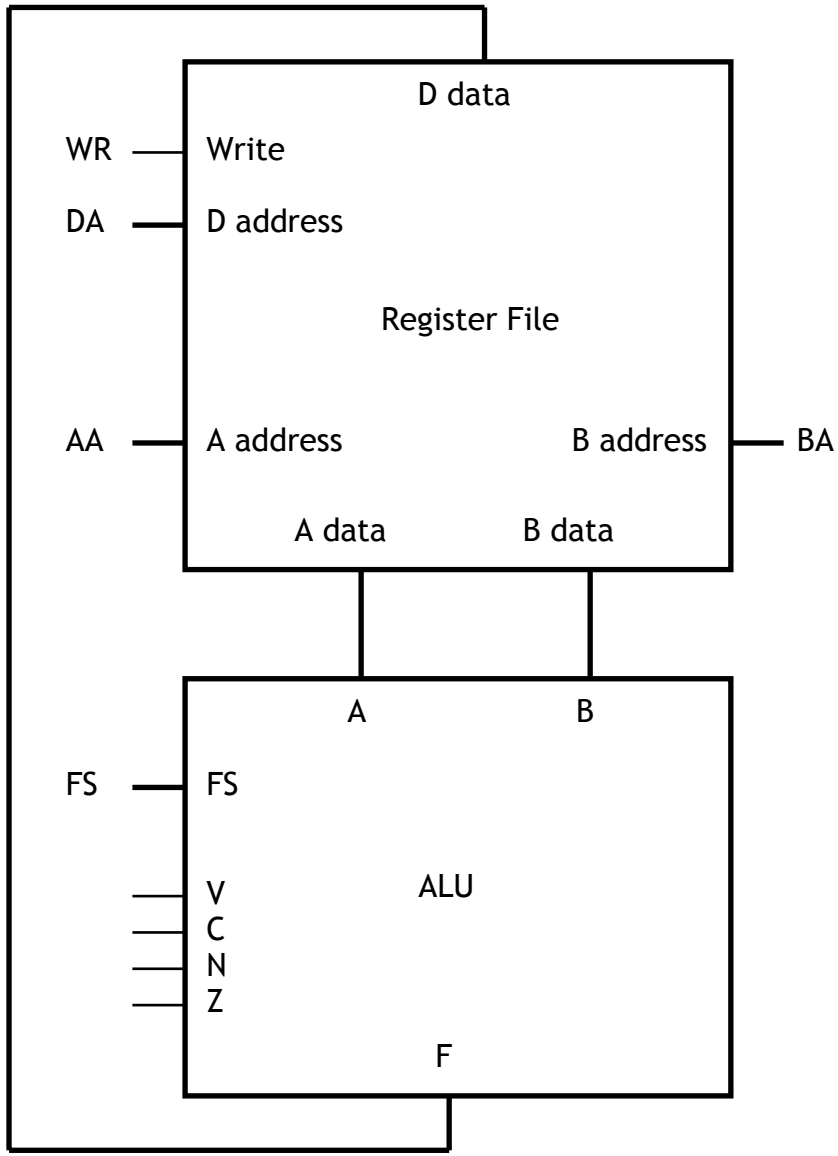
$$R0 \leftarrow R1 + R3$$

- Set AA = 01 and BA = 11. This causes the value in R1 to appear at A data, and the value in R3 to appear at the output B data.
- Set the ALU function select input FS = 00010, to perform the addition A + B.
- Set DA = 00 and WR = 1. On the next positive clock edge, the ALU's result (R1 + R3) will be stored in R0.



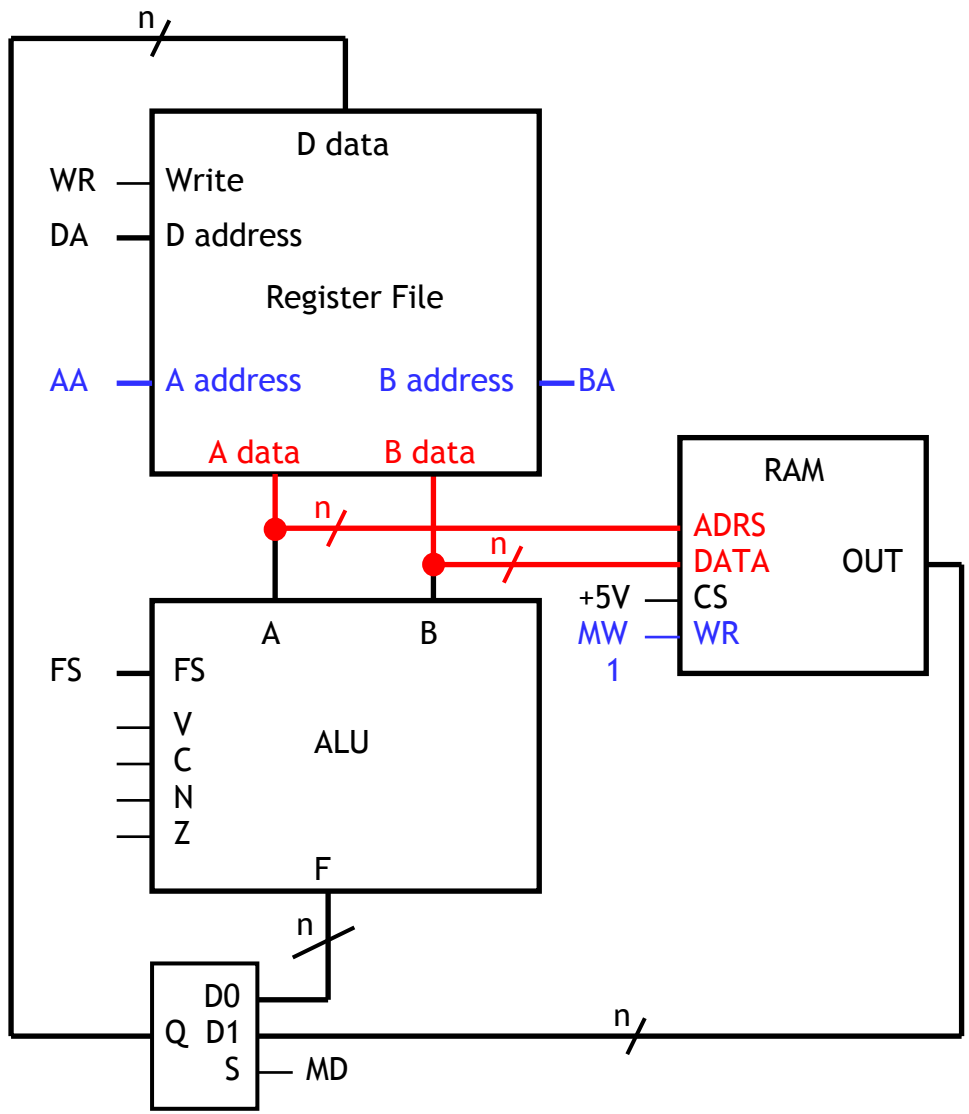
Two questions

- Four registers isn't a lot. What if we need more storage?
- Who exactly decides which registers are read and written and which ALU function is executed?



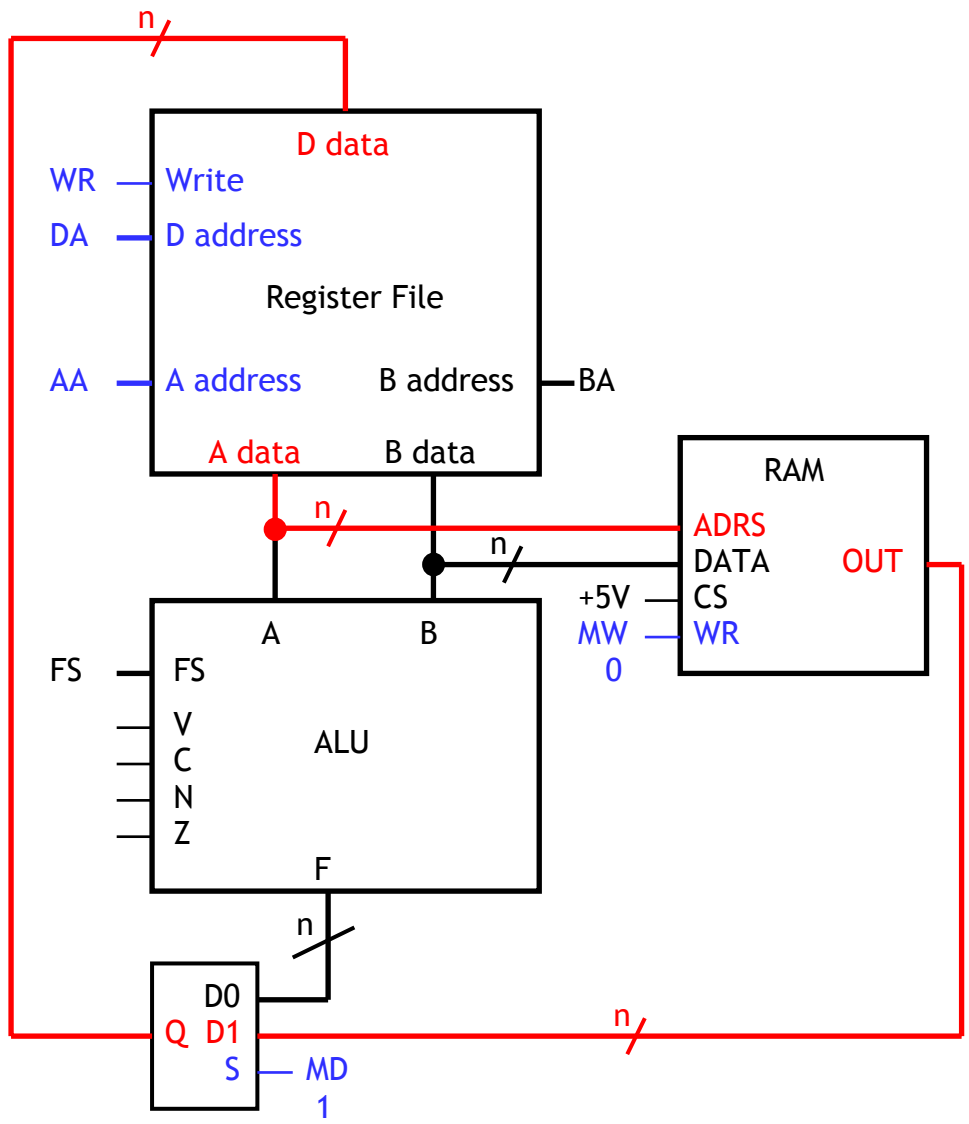
We can access RAM also

- Here's one way to connect RAM to our existing datapath.
- To *write* to RAM, we must give an address and a data value; these will come straight from the register file.
  - We connect A data to the memory ADRS input.
  - We also send B data to the memory DATA input.
- Finally, we must set MW = 1 to write to the RAM. (It's called MW because WR is already used by the register file.)



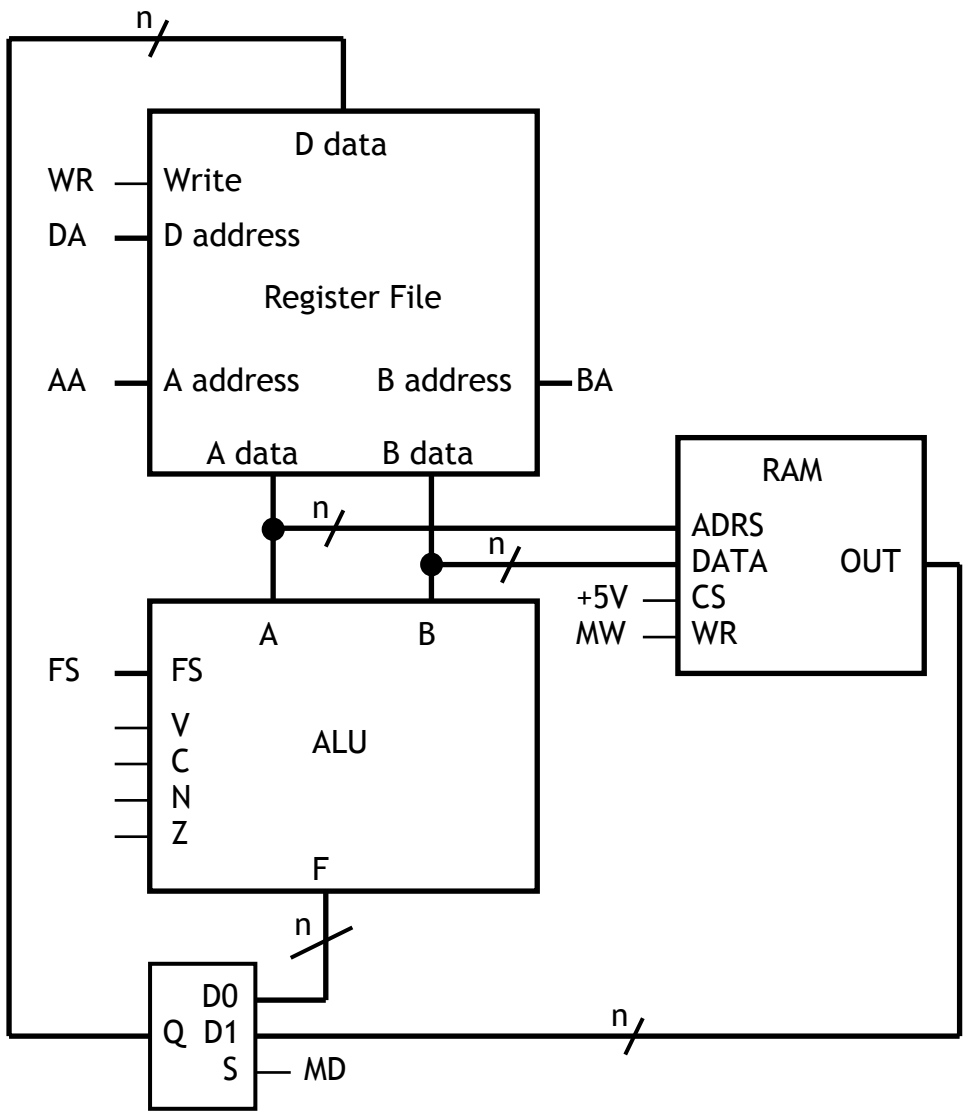
Reading from RAM

- To *read* from RAM, A data must supply an address, and we also need to ensure MW = 0.
- The RAM output should be sent to the register file for storage.
- This means that the register file's D data input could come from *either* the ALU output or the RAM.
- A new multiplexer MD selects a source for the register file.
  - If MD = 0, the ALU output can be stored in the register file.
  - If MD = 1, the RAM output is sent to the registers instead.



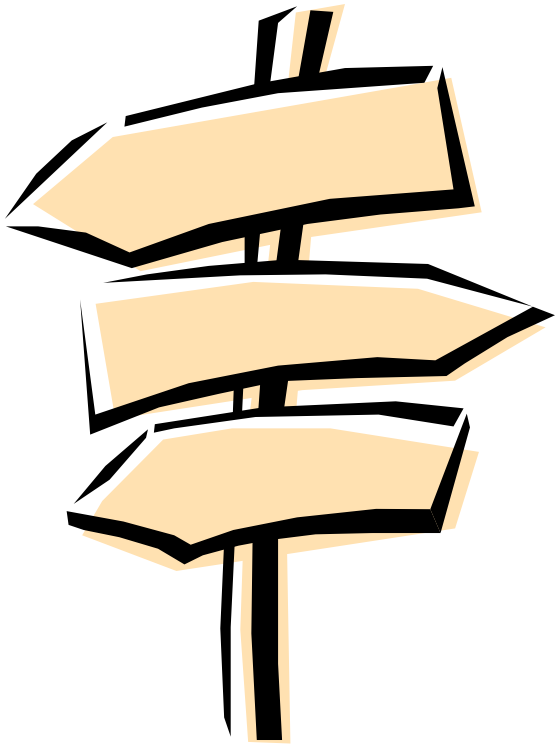
Notes about this setup

- We now have a way to copy data between our register file and the RAM.
- Notice that there's no way for the ALU to directly access the memory—RAM values *must* go through the register file first.
- Here the size of the memory is limited by the size of the registers; with *n*-bit registers, we can only use a $2^n \times n$ RAM.
- For simplicity we'll assume the RAM is at least as fast as the CPU clock, even though this is not exactly true in real life.



Memory transfer notation

- In our transfer language, the contents of random access memory address X are denoted as M[X]. Here are two examples.
  - The first word in RAM is M[0].
  - If register R1 holds an address, then M[R1] is the data at that address.
- The M[ ] notation is like a pointer dereference operation in C or C++.



Example sequence of operations

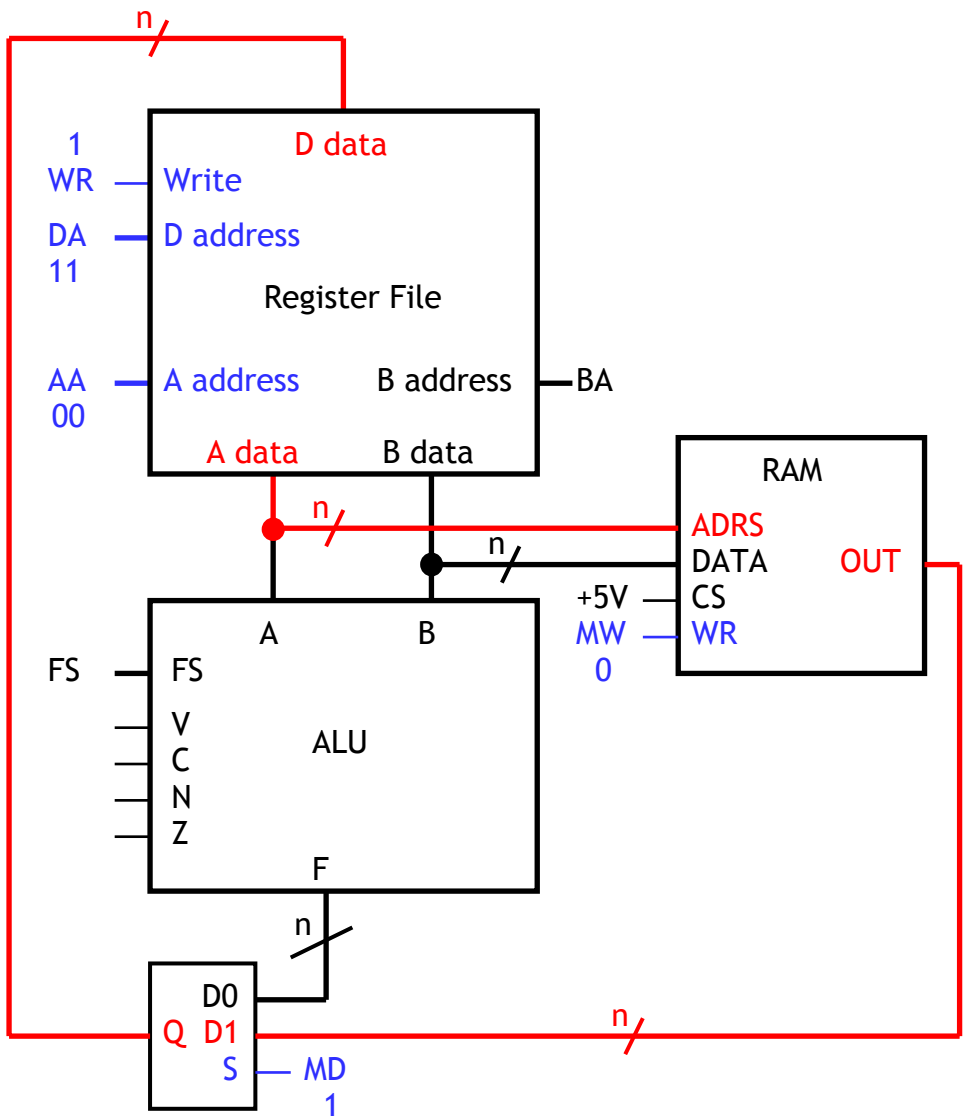
- Here is a simple series of register transfer instructions.

$$R3 \leftarrow M[R0]$$
$$R3 \leftarrow R3 + 1$$
$$M[R0] \leftarrow R3$$

- This just increments the value stored at address R0 in RAM.
  - R0 is the first register in our register file. We'll assume it contains a valid memory address.
  - Again, our ALU only operates on registers, so the RAM contents must first be loaded into a register, and then saved back to RAM.
- We can perform these three register transfer operations in our datapath over three consecutive clock cycles.

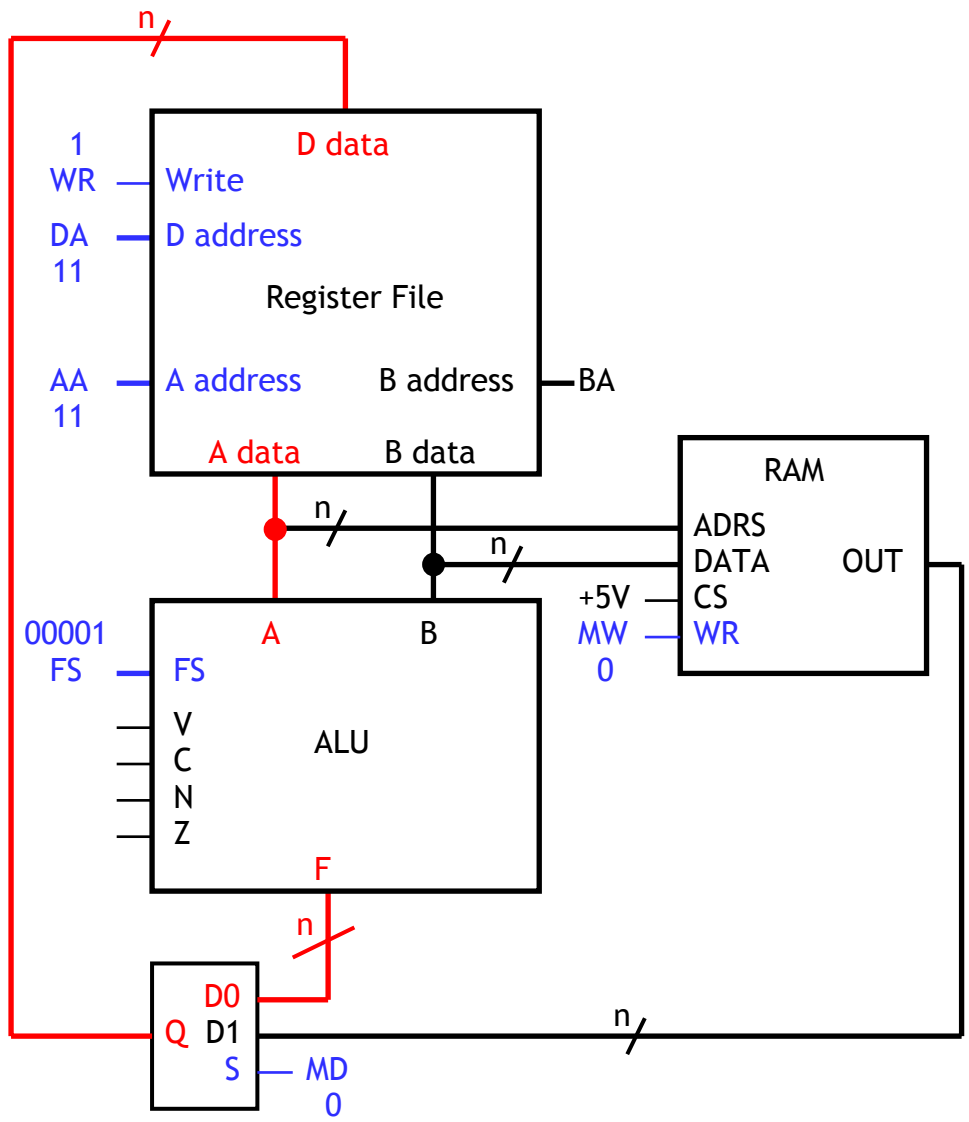
R3 ← M[R0]

- AA should be set to 00, to read register R0.
- The value in R0 will be sent to the RAM address input, so M[R0] appears as the RAM output OUT.
- MD must be 1, so the RAM output goes to the register file.
- To store something into R3, we need to set DA = 11 and WR = 1.
- MW should be 0, so nothing is accidentally changed in RAM.
- We did not use the ALU or the second register file output for this operation, so FS and BA can be don't care conditions.



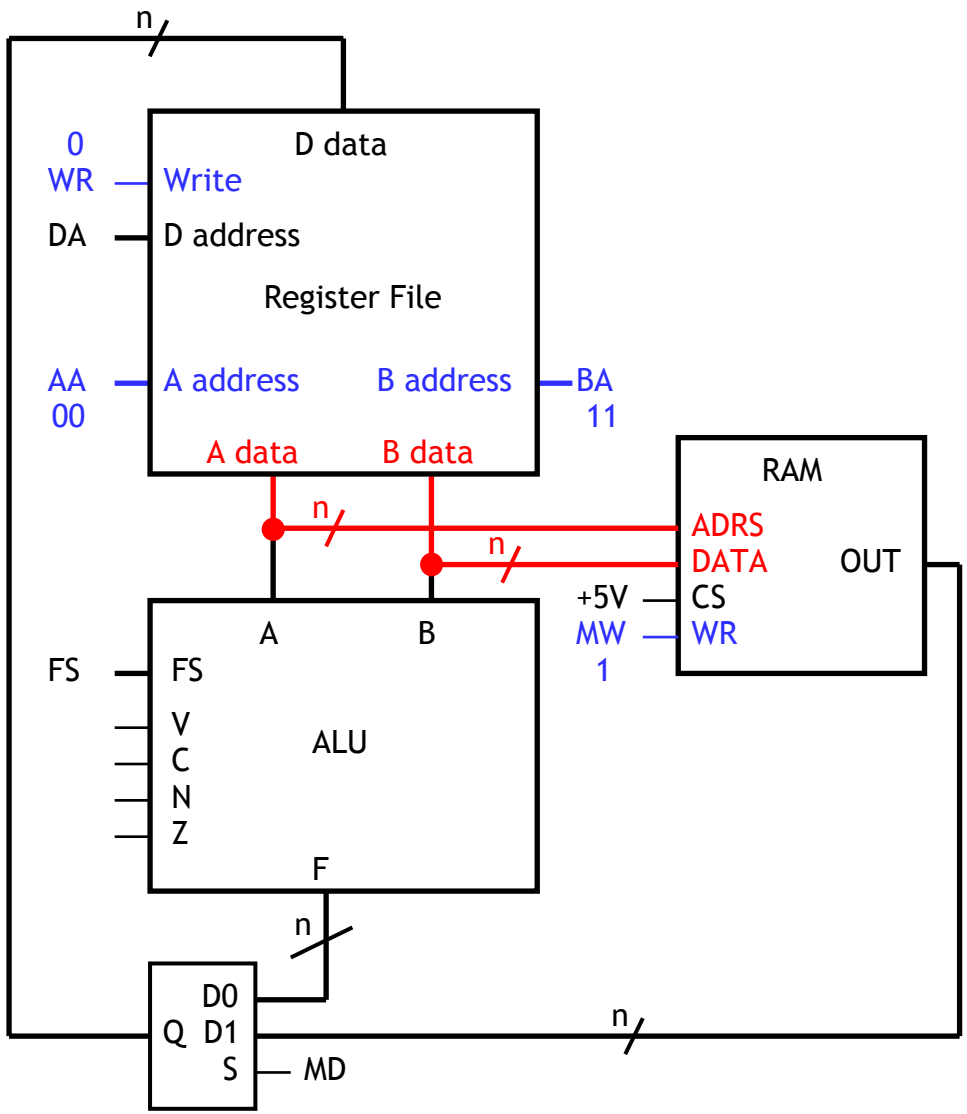
R3 ← R3 + 1

- AA = 11, so R3 is read from the register file and directed to the ALU's A input.
- FS needs to be 00001 for the operation A + 1. Then, R3 + 1 appears as the ALU output F.
- If MD is set to 0, this output will go back to the register file.
- To write to R3, we need to set DA = 11 and WR = 1.
- Again, MW should be 0 so that RAM contents aren't changed.
- We didn't use BA.



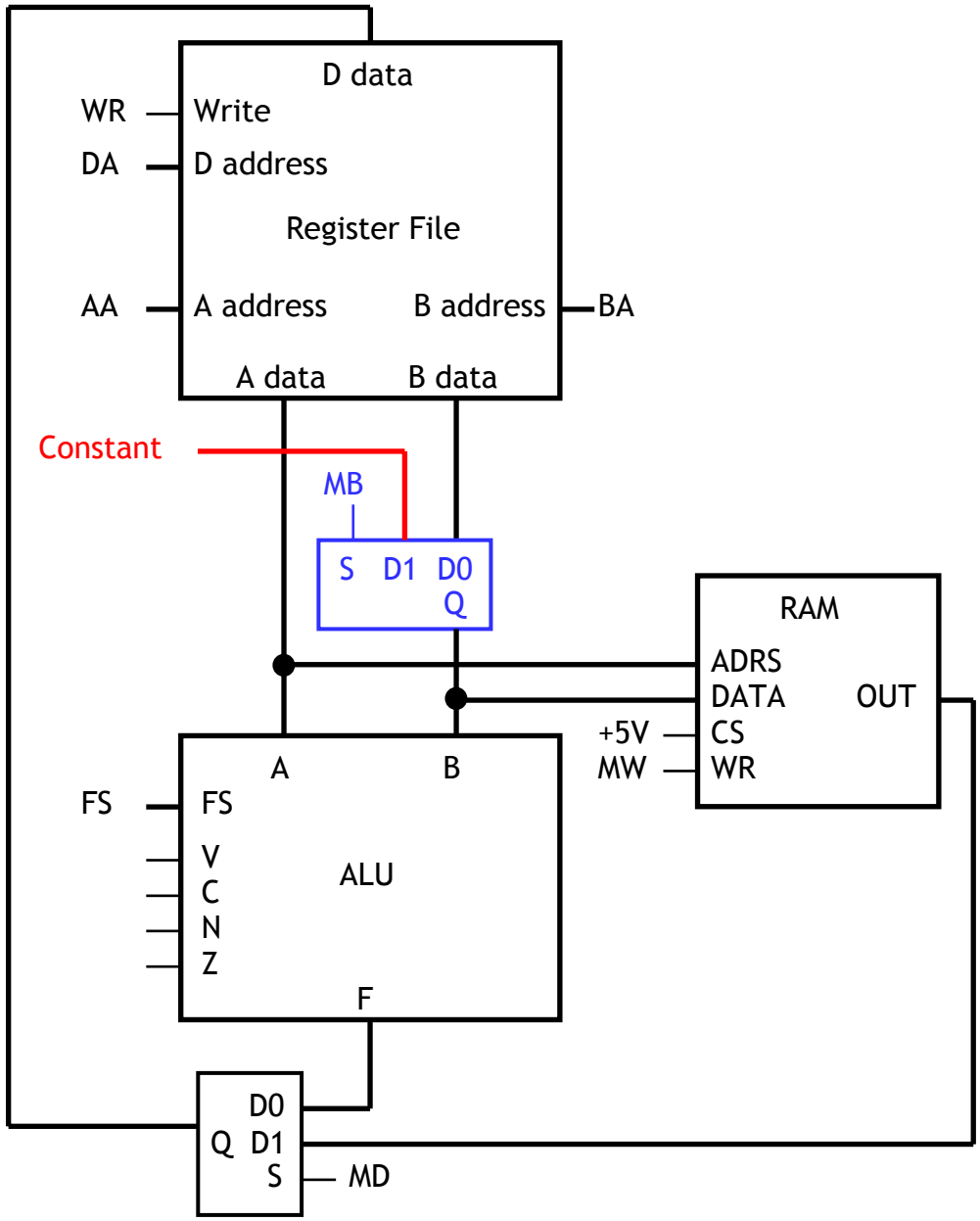
M[R0] ← R3

- Finally, we must store the value of R3 into RAM address R0.
- The memory address comes from "A data" and the contents come from "B data." We have to set AA = 00 and BA = 11, to send R0 to ADRS and R3 to DATA.
- MW must be 1 to write to RAM.
- No register updates are needed, so WR should be 0, which means MD and DA are unused.
- We also didn't need the ALU, so FS can be ignored.



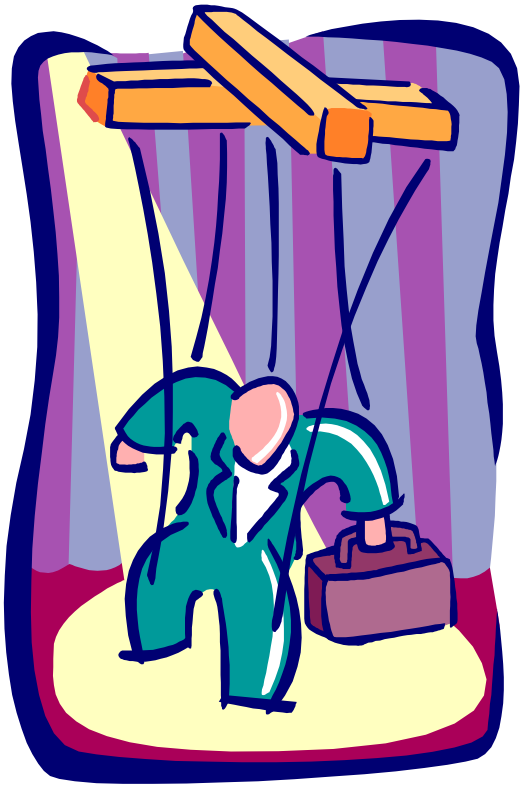
Constant in

- One last addition to our basic datapath is a Constant input.
- The revised datapath is shown here, with a new multiplexer and control signal MB.
- We'll see how this is used later. Intuitively, it provides an easy way to initialize a register or memory location with an arbitrary number.



Control units

- From these examples, you can see that different actions can be performed by providing different inputs for the datapath control signals.
- The two multiplexers MB and MD help us select from one of several sources, just like the muxes in the register transfer lecture last time.
- The second question we had was "Who exactly decides which registers are read and written, and which ALU function is executed?"
  - In real computers, 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 next week, and then control units the week after that.



Summary

- A processor's basic job is to move data between registers, possibly with some ALU computations.
- We presented a register transfer language for describing computation and movement of data between registers.
- The datapath is the part of a processor where this is all done.
  - The basic components are an ALU, a register file and some RAM.
  - The ALU does all of the computations, while the register file and RAM provide storage for the ALU's operands and results.
- Various control signals in the datapath determine its behavior.
- Next we'll see how programmers give commands to the processor, and how those commands are translated in control signals for the datapath.