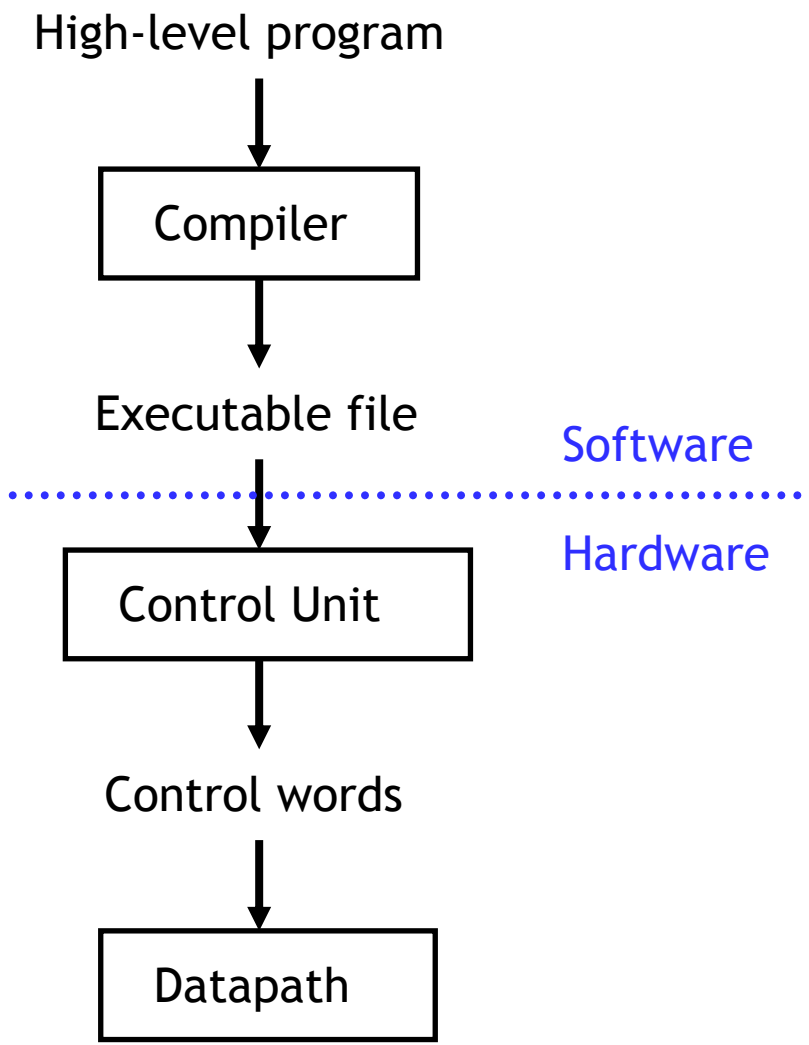
Instruction set architectures

- Yesterday we built a simple, but complete, datapath.
- The datapath is ultimately controlled by a program, so today we'll look at programming in more detail.
  - We'll talk about how high-level programs are executed on processors.
  - Then we will also introduce a sample instruction set architecture for our datapath from yesterday, and present some example programs.
- Next week we'll finish our processor by designing the control unit, which converts program instructions into signals for the datapath.

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 that 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.



Some CPUs

- AMD and Intel make 8086-compatible chips that use the same machine language but are implemented in very different ways.

- The CSIL labs have a lot of Sun workstations, which use Sun UltraSPARC processors.

- IBM and Motorola both manufacture PowerPC processors, which appear in Apple Macintosh computers and Nintendo's GameCube.

- MIPS designs processors that are used in SGI computers, the Sony Playstation 2, and other places...

High-level languages

- High-level languages provide many useful programming constructs.
  - For, while, and do loops
  - If-then-else statements
  - Functions and procedures for code abstraction
  - Variables and arrays for storage
- Many languages provide safety features as well.
  - Static or dynamic typechecking
  - Garbage collection
- High-level languages are also relatively portable.Theoretically, you can write one program and compile it for many different processors.
- It may be hard to understand what's so "high-level" about languages like Java or C++ until you compare them with...

Low-level languages

- Every CPU has a low-level instruction set, or machine language, which closely reflects that processor's design.
- Unfortunately, this means instruction sets are not easy for humans to work with!
  - Control flow is limited to "jump" and "branch" instructions, which you must use to make your own loops and conditionals.
  - Support for functions and procedures may be limited.
  - Memory addresses must be explicitly specified. You can't just declare new variables and use them!
  - Very little error checking is done for machine language code.
  - It's difficult to convert machine language programs written for one processor to another.
- Later we'll look at some rough translations from C to machine language.

Compilers

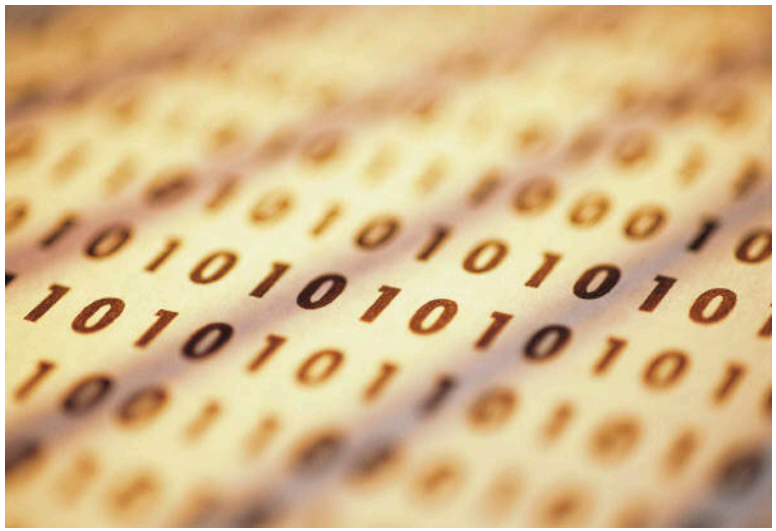
- Compilers are used to translate high-level programs into machine code.
- In the good old days, people often wrote machine language programs by hand to make their programs faster, smaller, or both.
- Now, compilers almost always do a better job than people.
  - Programs are becoming more complex, and it's hard for humans to write and maintain large, efficient machine language code.
  - CPUs are becoming more complex. It's difficult to write code that takes full advantage of a processor's features.

Interpreted languages

- There are also many interpreted languages.
  - Java, Python and Perl are probably the most popular examples.
  - Others include Lisp, Smalltalk, Prolog.
- Interpreted languages take a middle ground.
  - Instead of being compiled directly to machine language, programs are compiled to produce an intermediate code, like Java bytecode.
  - The intermediate code is executed by an interpreter, such as the Java virtual machine.
- This approach tries to balance portability and efficiency.
  - Java bytecode can be executed on any processor and operating system that has a Java virtual machine.
  - The final interpretation step is still faster than a full compilation.

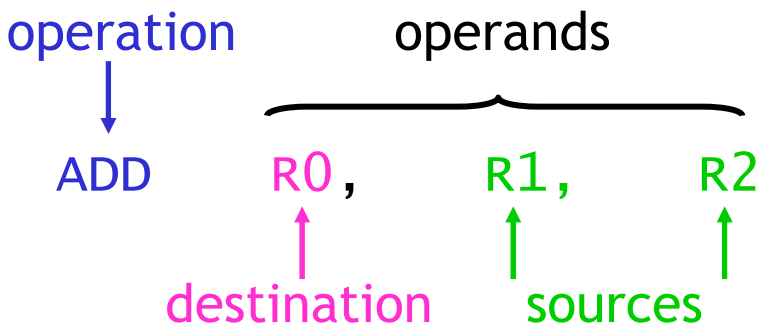
Assembly and machine languages

- 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 nearly a one-to-one correspondence between these names and machine instructions, so converting assembly code to machine language is very easy.
- We will use assembly code today to introduce the basic ideas, and switch to machine language later when we talk about the control unit.



Data manipulation instructions

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



Register transfer equivalent:

```
R0 ← R1 + R2
```

- This is similar to high-level programming statements like the following.

```
R0 = R1 + R2
```

- Here, all of the operands are registers.

More data manipulation instructions

- Here are some other possible data manipulation instructions.

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

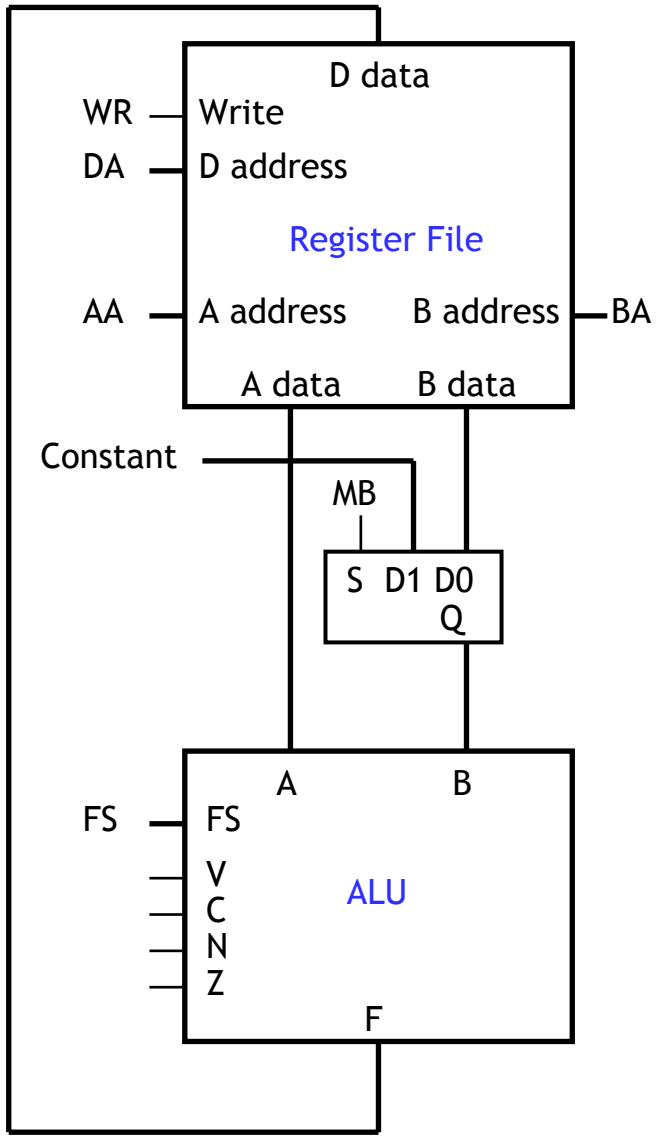
- Some instructions, like the NOT, have only one operand.
- In addition to register operands, constant operands like 1 and 5 are also possible. Constants are sometimes denoted with a hash mark in front.

Relation to the datapath

- These instructions reflect the design of our datapath from yesterday.
- There are at most two source operands in each instruction, since our ALU has just two inputs. The sources can both be registers, or they can be one register and one constant.
- Instructions have just one destination operand, which must be a register.
- More complex operations like

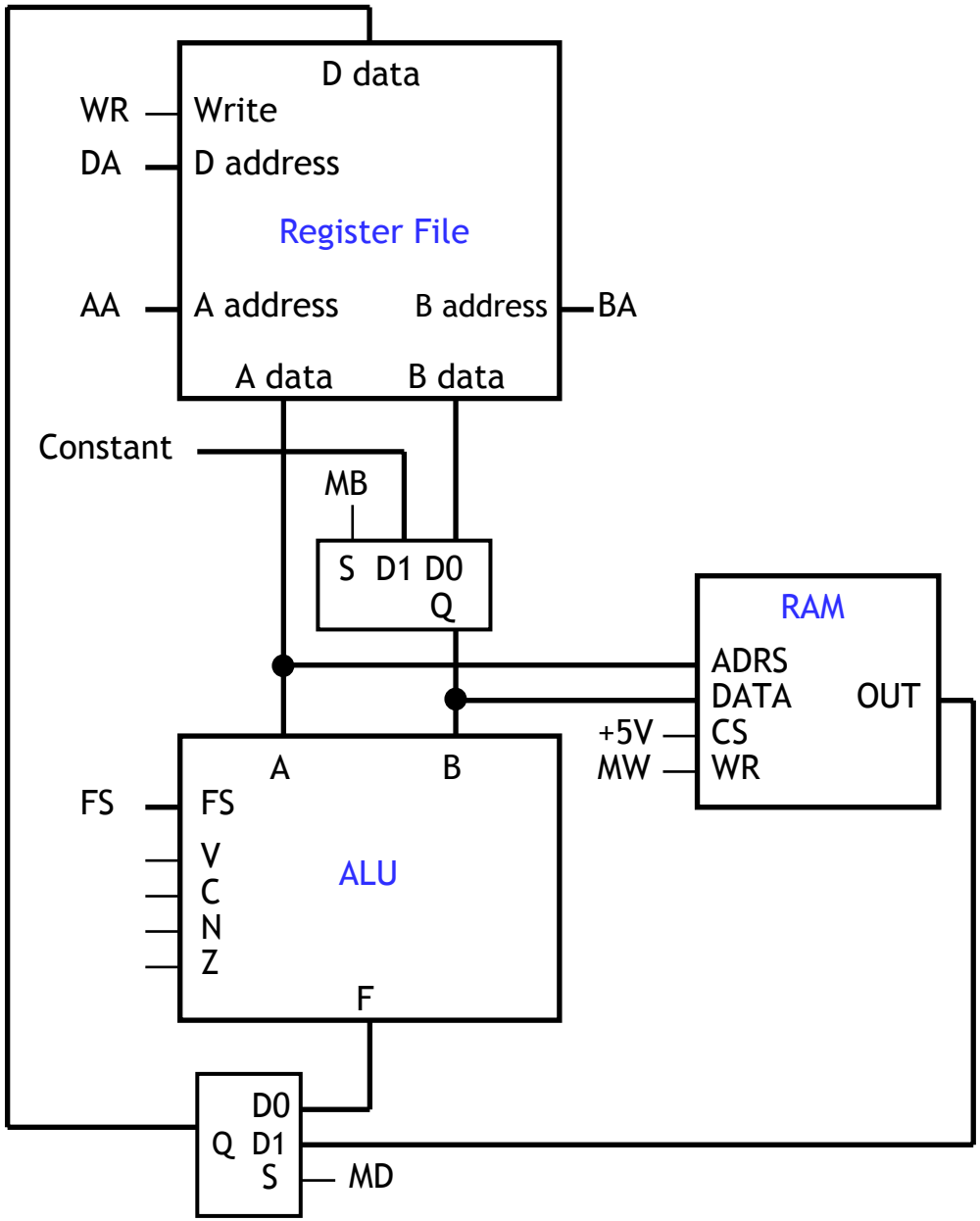
  `R0 ← R1 + R2 - 3`

  must be broken down into several lower-level instructions.



What about RAM?

- Recall that our ALU has direct access to the registers only.
- 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 are stored into memory.
- We rely on data movement instructions to transfer data between RAM and the register file.

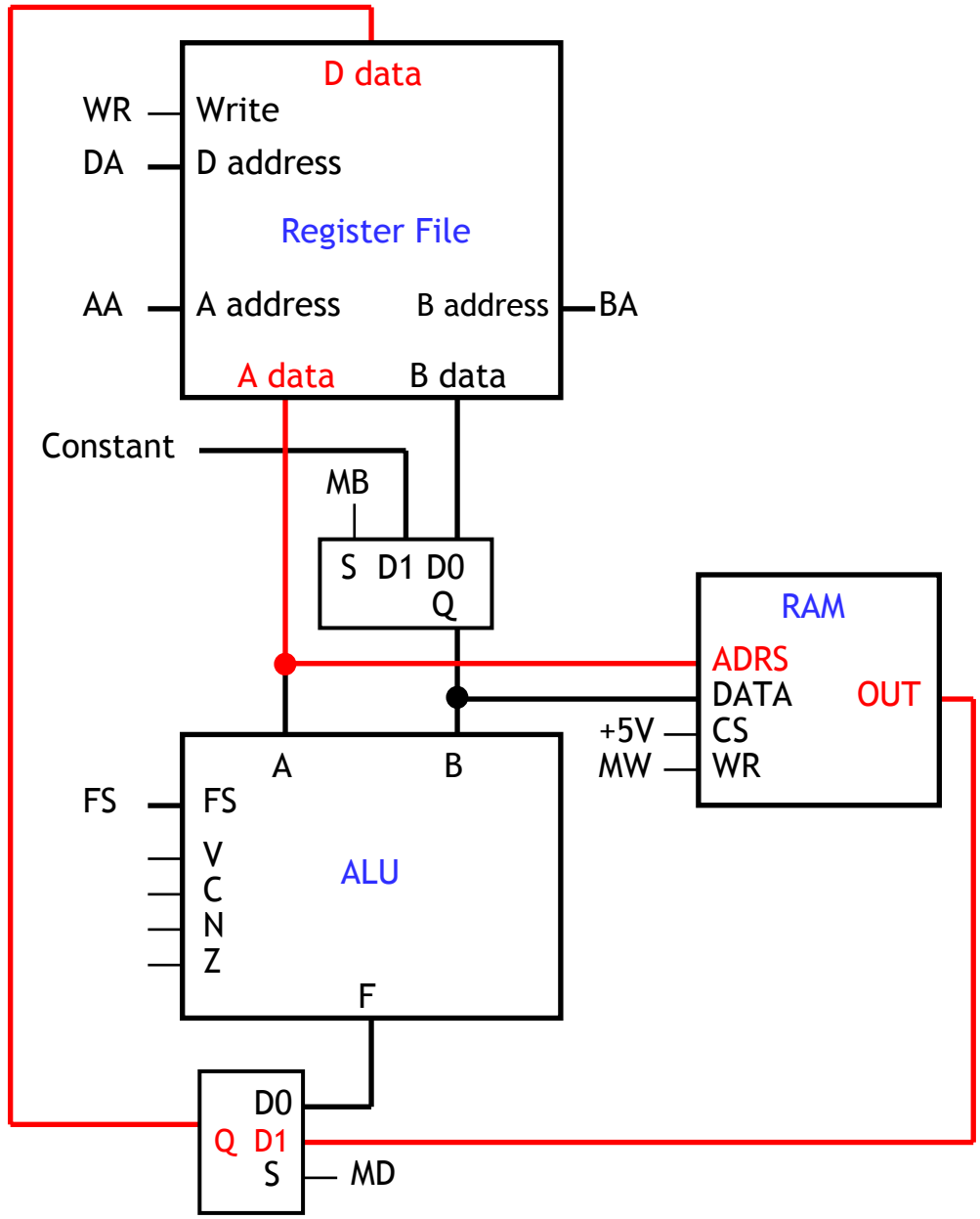


Loading a register from RAM

- 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.

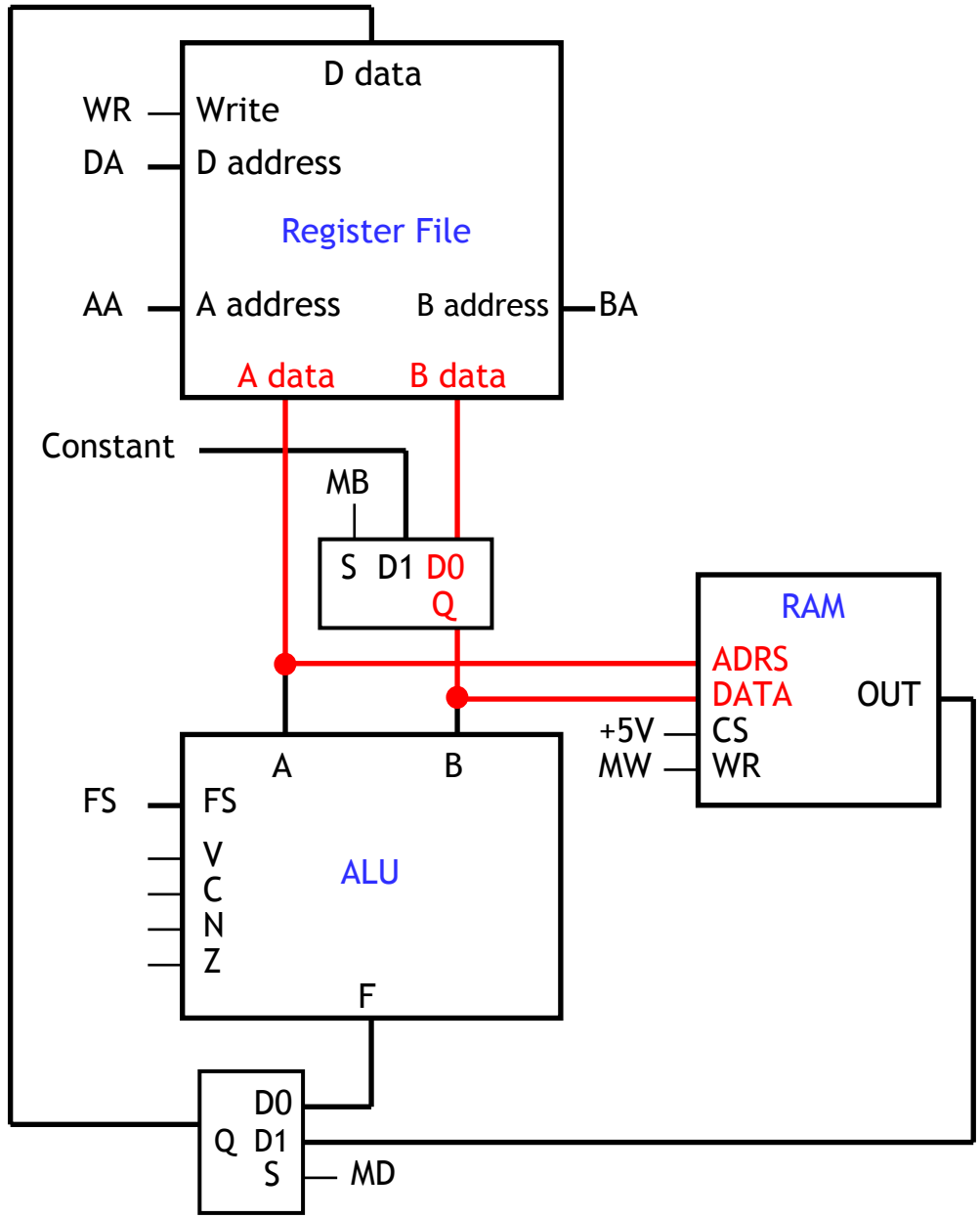


Storing a register to RAM

- 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, like R1 above.

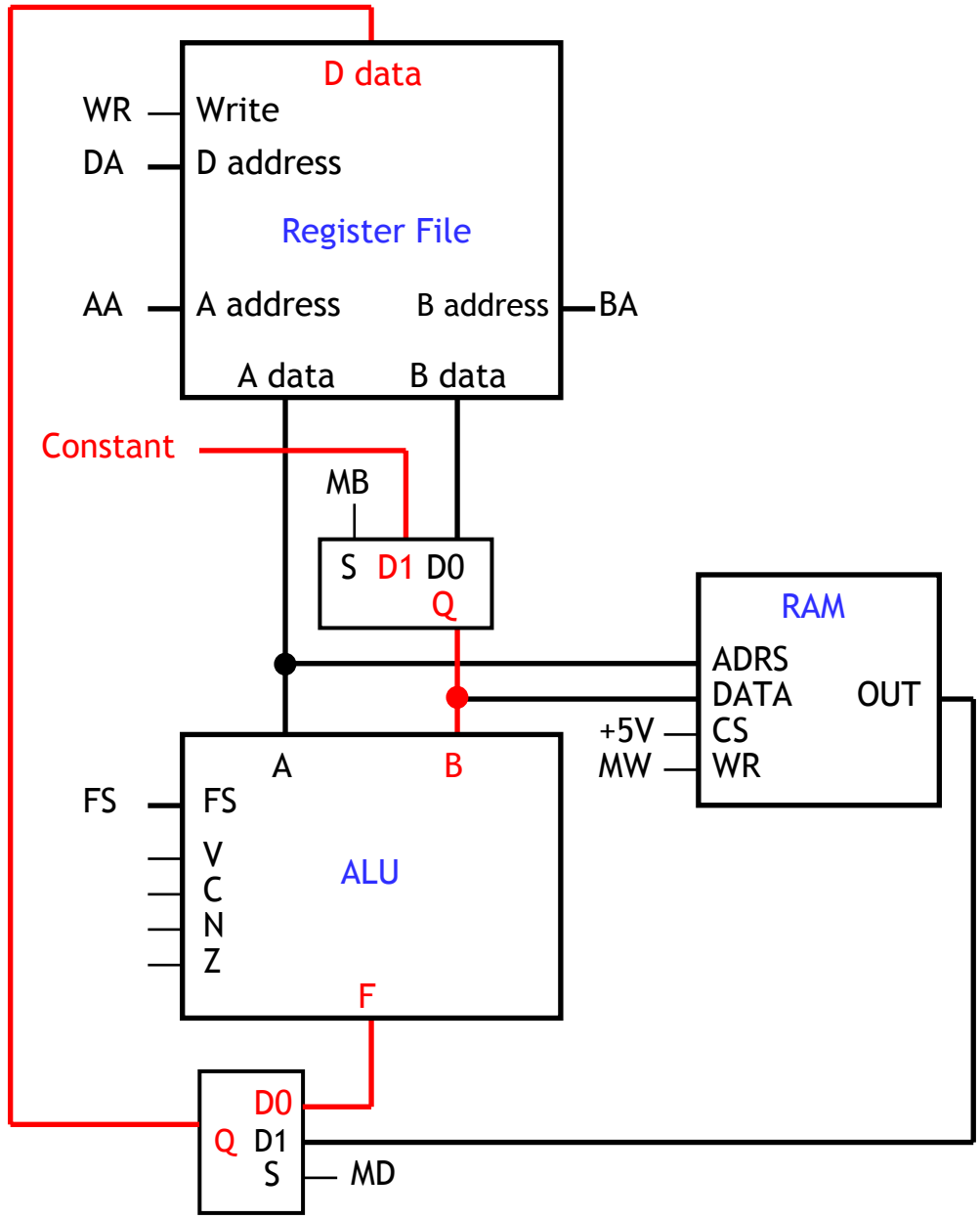


Loading a register with a constant

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

  `LD R1, #0` `R1 ← 0`

- This 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.

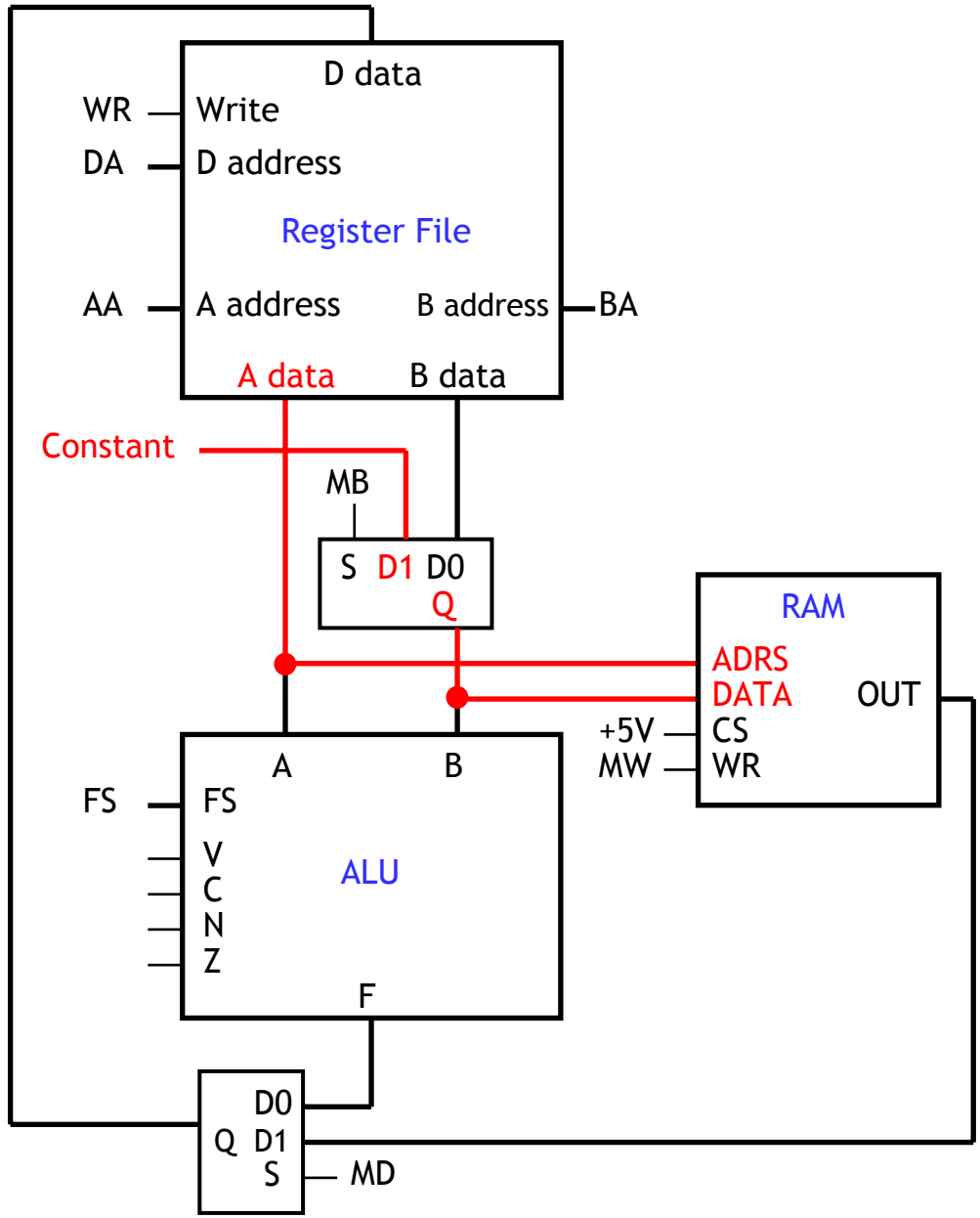


Storing a constant to RAM

- 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.



A small example

- Here's an example register-transfer operation.

$$M[1000] \leftarrow M[1000] + 1$$

- Below 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.
  - In CS232 you'll see more about 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 computer memory sequentially as well.
  - 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.

Jumps

- A jump instruction *always* changes the value of the PC.
  - The operand specifies exactly how to change the PC.
  - For simplicity, we often use labels to denote actual addresses.
- For example, a program can skip certain instructions.

```
        LD   R1, #10
        LD   R2, #3
        JMP L
    K   LD   R1, #20      // These two instructions
        LD   R2, #4       //   would be skipped
    L   ADD R3, R3, R2
        ST   (R1), R3
```

- You can also use jumps to repeat instructions.

```
        LD   R1, #0
    F   ADD R1, R1, #1
        JMP F             // An infinite loop!
```

Branches

- A branch instruction *may* change the PC, depending on whether a given condition is true.

```
      LD   R1, #10
      LD   R2, #3
      BZ   R4, L          // Jump to L if R4 == 0
K     LD   R1, #20        // These instructions may be
      LD   R2, #4         // skipped, depending on R4
L     ADD  R3, R3, R2
      ST   (R1), R3
```

Types of branches

- Branch conditions are often based on the ALU result.
- This is what the ALU status bits V, C, N and Z are used for. With them we can implement various branch instructions like the ones below.

Condition	Mnemonic	ALU status bit
Branch on overflow	BV	V = 1
Branch on no overflow	BNV	V = 0
Branch if carry set	BC	C = 1
Branch if carry clear	BNC	C = 0
Branch if negative	BN	N = 1
Branch if non-negative	BNN	N = 0
Branch if zero	BZ	Z = 1
Branch if not zero	BNZ	Z = 0

- Other branch conditions (e.g., branch if greater, equal or less) can be derived from these, along with the right ALU operation.

High-level control flow

- These jumps and branches are much simpler than the control flow constructs provided by high-level languages.
- Conditional statements execute only if some Boolean value is true.

```
// Find the absolute value of *X
R1 = *X;
if (R1 < 0)
    R1 = -R1;              // This might not be executed
R3 = R1 + R1;
```

- Loops cause some statements to be executed many times

```
// Sum the integers from 1 to 5
R1 = 0;
for (R2 = 1; R2 <= 5; R2++)
    R1 = R1 + R2;          // This executes five times
R3 = R1 + R1;
```

Translating the C if-then statement

- We can use branch instructions to translate high-level conditional statements into assembly code.

```
R1 = *X;
if (R1 < 0)
    R1 = -R1;
R3 = R1 + R1;
```

```
     LD   R1, (X)          // R1 = *X
     BNN  R1, L            // Skip MUL if R1 is not negative
     MUL  R1, R1, #-1      // R1 = -R1
L    ADD  R3, R1, R1       // R3 = R1 + R1
```

- Sometimes it's easier to *invert* the original condition. Here, we effectively changed the `R1 < 0` test into `R1 >= 0`.

Translating the C for loop

- Here is a translation of the for loop, using a hypothetical BGT branch.

```
R1 = 0;
for (R2 = 1; R2 <= 5; R2++)
    R1 = R1 + R2;
R3 = R1 + R1;
```

```
      LD   R1, #0          // R1 = 0
      LD   R2, #1          // R2 = 1
FOR   BGT  R2, #5, L       // Stop when R2 > 5
      ADD  R1, R1, R2      // R1 = R1 + R2
      ADD  R2, R2, #1      // R2++
      JMP  FOR             // Go back to the loop test
L     ADD  R3, R1, R1      // R3 = R1 + R1
```

Functions

- Function calls and returns also affect a program's control flow.
  - The CPU must remember which instruction is being executed when the function call is made, so it can return properly.
  - There also needs to be some convention for passing arguments to a function, and accepting return values from a function.
- This is complicated by functions possibly calling *other* functions.
  - Many return addresses need to be remembered.
  - Processors have a limited number of registers, which must somehow be shared by all functions.
- For more information, you'll have to take more classes! CS232 will talk about functions in depth, for example.

Summary

- Machine language is the interface between software and processors.
- High-level programs must be translated into machine language before they can be executed.
- There are three main categories of instructions.
  - Data manipulation operations include the arithmetic instructions.
  - Data transfer operations to data between registers and RAM
  - Control flow instructions change the instruction execution order.
- Instruction set architectures depend highly on the host CPU's design.
  - Today we saw instructions that are appropriate for our datapath.
  - Next Monday we'll look at some other possible architectures.