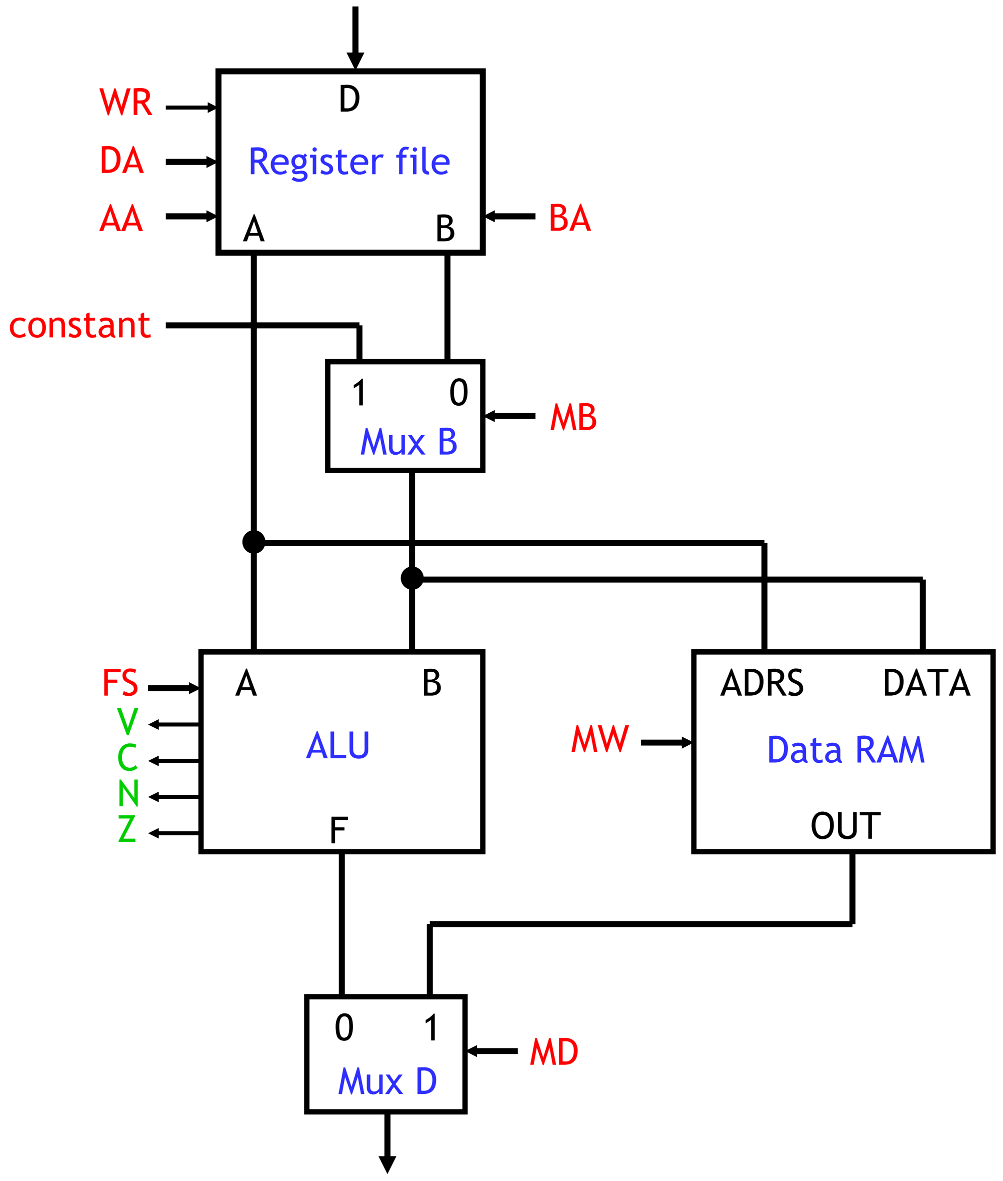
Control units

- Yesterday we showed how to translate assembly instructions into a binary machine language representation.
- Today we fill in the last piece of the processor and build a control unit to convert these binary instructions into the appropriate datapath signals.
- At the end of the day, we'll have a simple but complete processor!

Datapath review

- The datapath contains all of the circuitry and memory to do a variety of computations.
- The actual computations are determined by the various datapath control inputs in red.
  - AA, BA and MB select the sources for operations.
  - FS picks an ALU function.
  - MW = 1 to write to RAM.
  - MD, WR and DA allow data to be written back to the register file.
- The status bits V, C, N and Z provide further information about the ALU output.



Instruction set review

- Data manipulation instructions have one destination register and up to two sources, which can be two registers or a register and a constant.

```
ADD    R1, R2, R3        R1 ← R2 + R3
SUB    R1, R2, #2        R1 ← R2 – 2
```

- Data transfer instructions use register-indirect addressing mode to copy data between registers and memory.

```
LD     R1, (R2)          R1 ← M[R2]
ST     (R3), R1          M[R3] ← R1
```

- Jumps and branches on different conditions can also be executed.

```
JMP    LABEL1            PC ← LABEL1
BZ     R2, LABEL2        if R2 = 0 then PC ← LABEL2
```

Instruction format review

- Yesterday we encoded our assembly language instructions into a binary representation suitable for storing into memory.
- First we defined different binary formats for instructions with different types of operands.

Format	15 9	8 6	5 3	2 0
Register	Opcode	DR	SA	SB
Immediate	Opcode	DR	SA	OP
Jump/branch	Opcode	AD	SA	AD

Opcode:	7-bit Operation Code
DR:	3-bit Destination Register
SA:	3-bit Source Register A
SB:	3-bit Source Register B
OP:	3-bit Constant Operand
AD:	6-bit Address Field

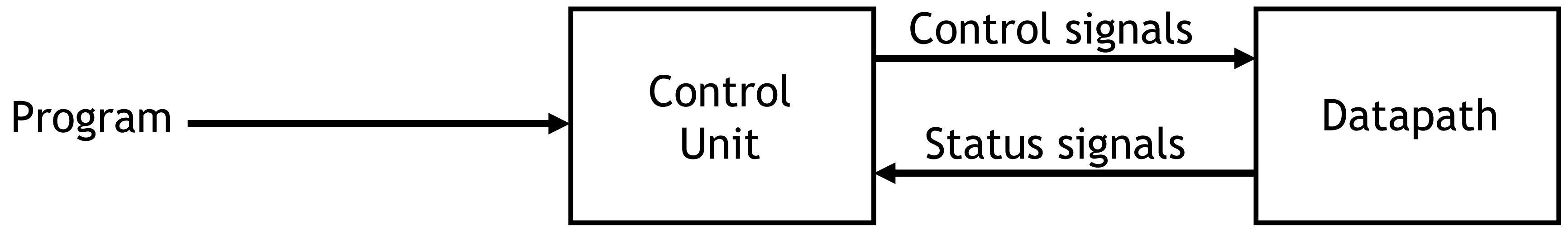
Opcode review

- We also defined binary opcodes to represent all the possible operations.

Category		Opcode bits						
		6	5	4	3	2	1	0
Register ALU instructions		0	0	FS				
Data transfer operations	ST	0	1	0	XXXX			
	LD	0	1	1	XXXX			
Immediate ALU instructions		1	0	FS				
Branches and jumps	Branches	1	1	0	X	BC		
	JMP	1	1	1	XXXX			

- We arranged the operations in a way that will make today's control unit easier to design.
- Many opcodes have unused bits to keep the different instruction formats consistent and to leave room for future expansion of the ISA.

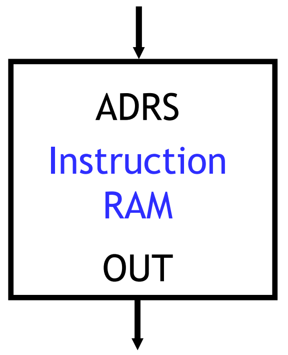
Block diagram of a processor

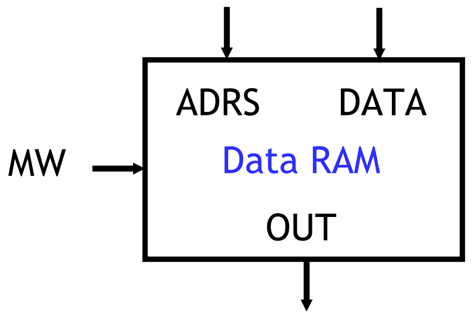


- The control unit is the missing piece which connects assembly language programs and the datapath.
  - It reads and executes program instructions in the correct sequence, accounting for any possible jumps and branches.
  - It converts each machine instruction into a set of control signals for the datapath, including WR, DA, AA, BA, MB, FS, MW and MD.
- We'll start by explaining the main components of the control unit, and then we'll see how to implement those components in more detail.

Where does the program go?

- We'll use a Harvard architecture, which includes two memory units.
  - An instruction memory holds the 16-bit program instructions.
  - A separate data memory is used for computations, as we already saw.
- The advantage is that we can read an instruction *and* load or store data in the same clock cycle.

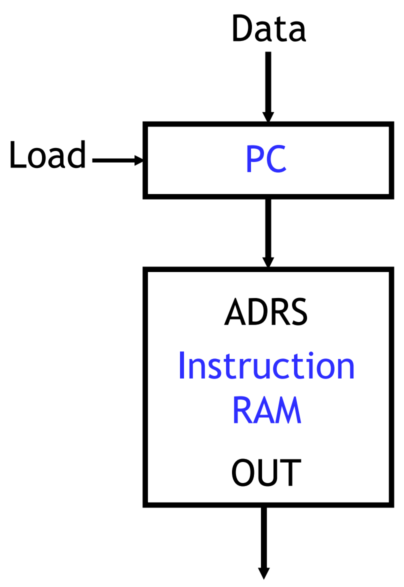




- For simplicity, our diagrams will not show any WR or DATA inputs to the instruction memory—we'll assume the program is already loaded, and it can't be changed while it's running.
- Many modern processors also use a Harvard architecture, with separate instruction and data caches.

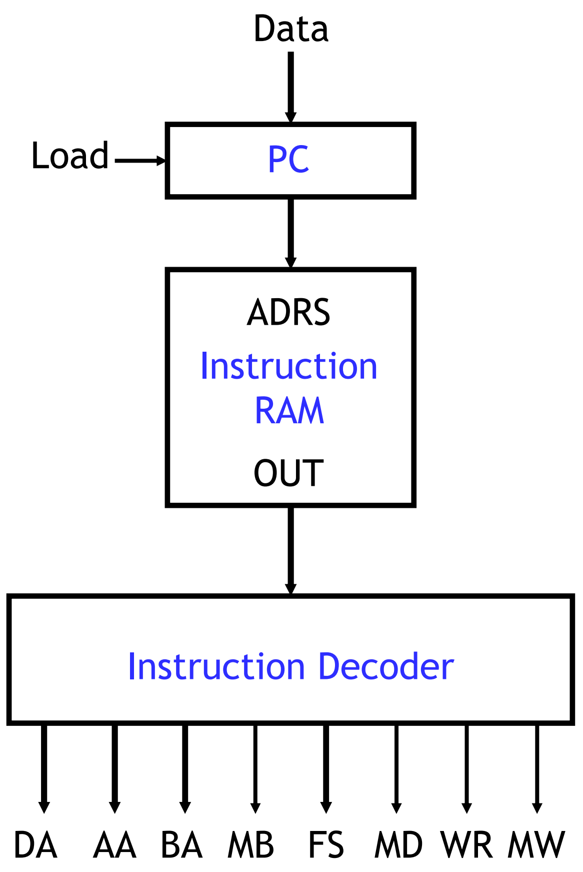
What is the current instruction?

- A program counter or PC addresses the instruction memory, to keep track of the instruction currently being executed.
- On each clock cycle, the counter does one of two things.
  - If Load = 0, the PC increments, so the next instruction in memory will be executed.
  - If Load = 1, then the PC is updated with Data, which represents some address specified in a jump or branch instruction.



How do binary instructions get executed?

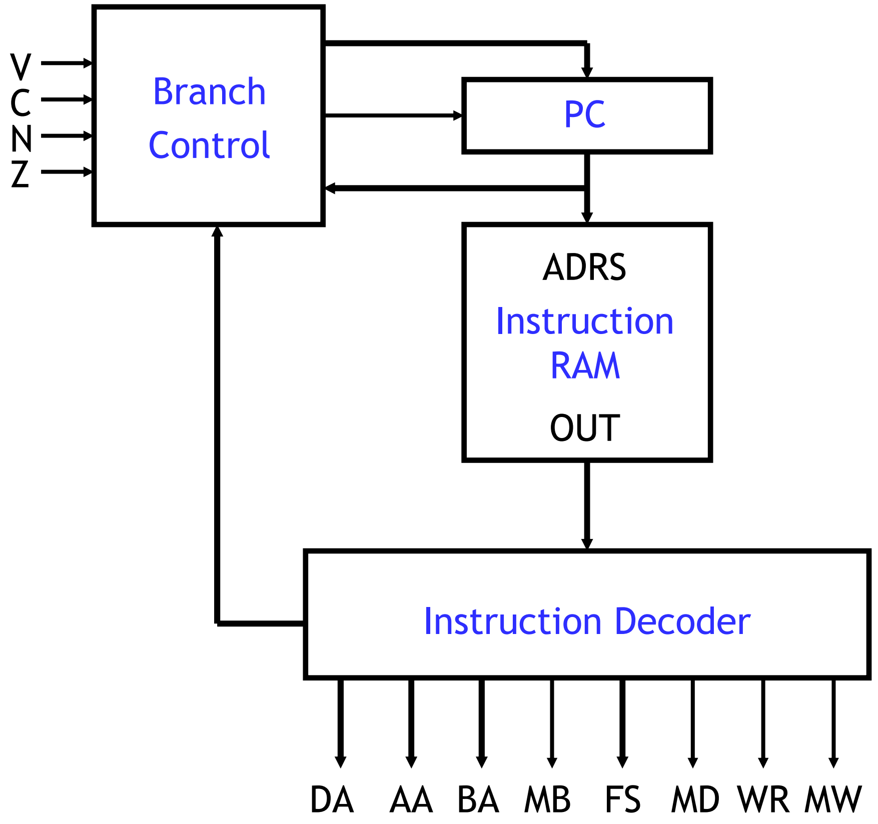
- The instruction decoder is a combinational circuit that takes a single machine language instruction and produces the appropriate control signals for the datapath.
- These signals will tell the datapath which registers or memory addresses to access, and what ALU operations to perform.



(to the datapath)

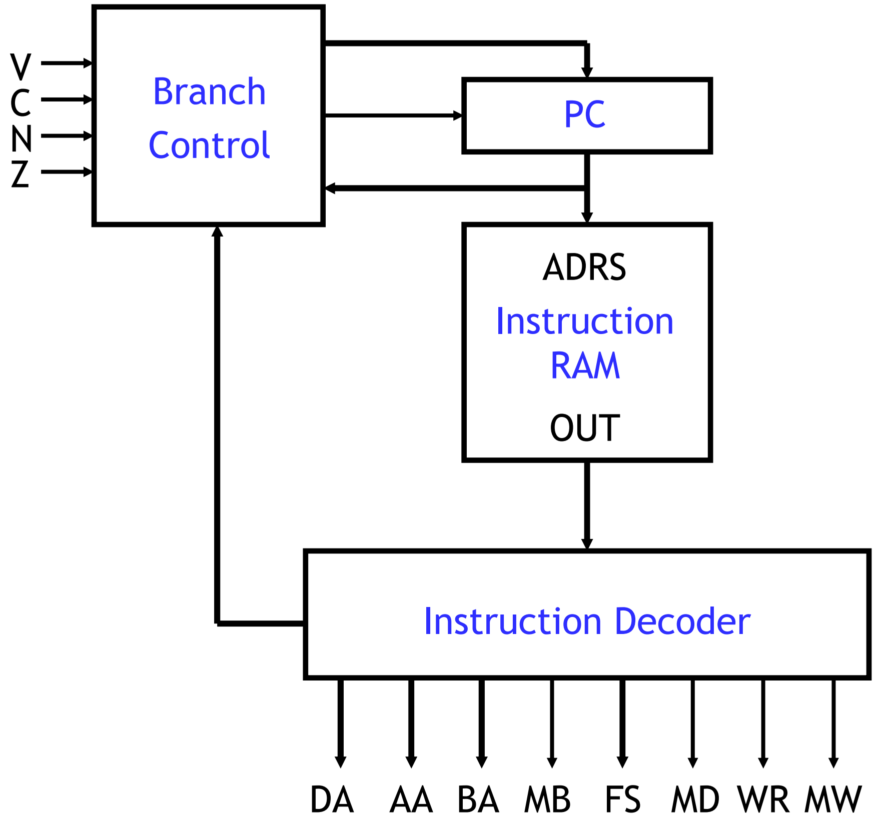
What instruction should be executed next?

- Finally, the branch control unit decides what the PC's next value should be.
  - For normal instructions the PC will just increment, so we can execute the next instruction.
  - For jumps, the PC should be loaded with a new address as specified in the instruction.
  - For branches, the PC will be loaded with a new address only if the matching status bit from the ALU is true.

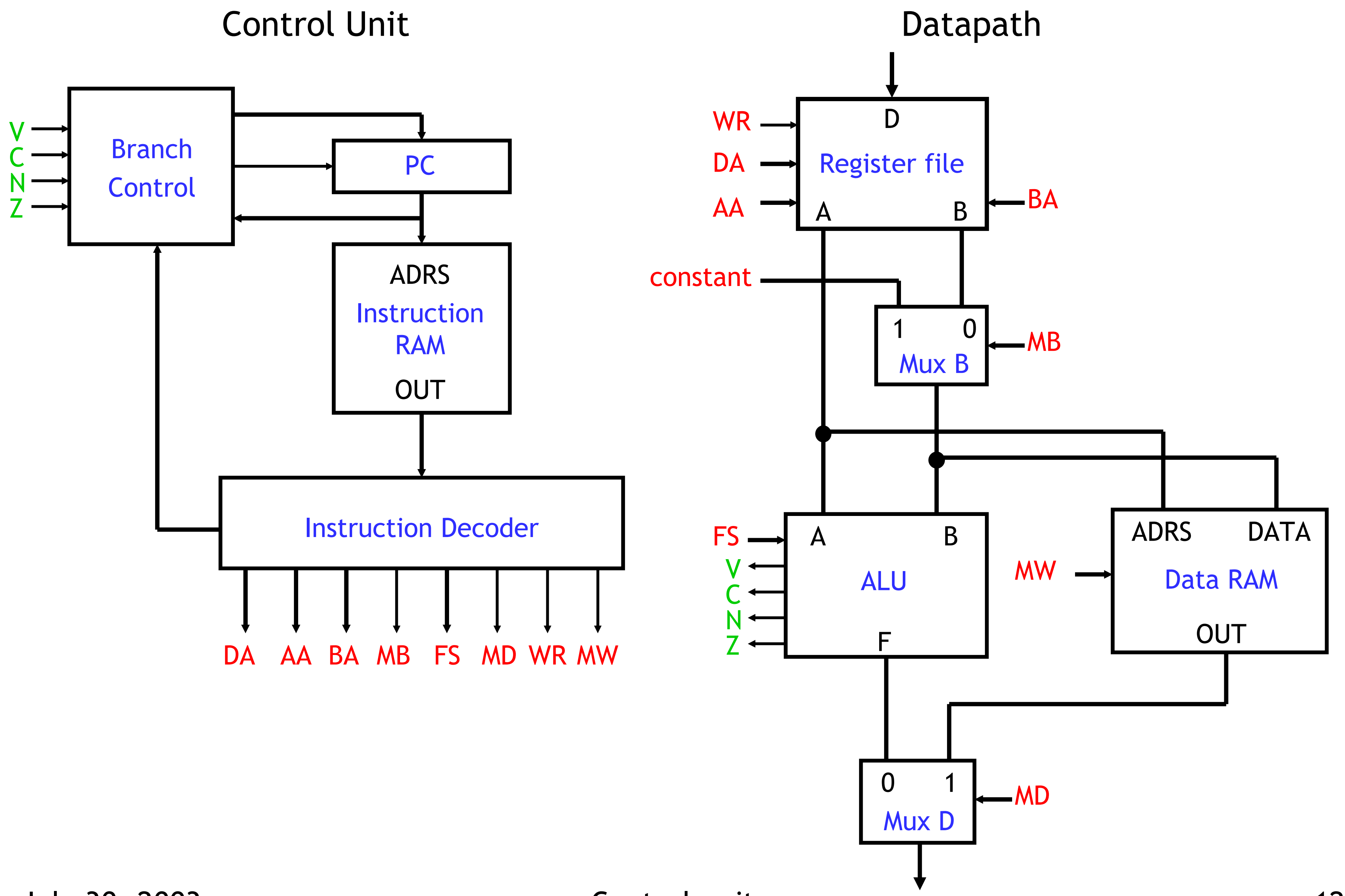


The whole control unit

- This is the basic control unit. On each clock cycle several actions occur.
  1. An instruction is read from the instruction memory.
  2. The instruction decoder generates the correct datapath control signals.
  3. Source registers are read.
  4. The ALU or data memory perform some operation.
  5. ALU or RAM outputs are stored back to the register file.
  6. The PC is incremented, or reloaded for branches and jumps.

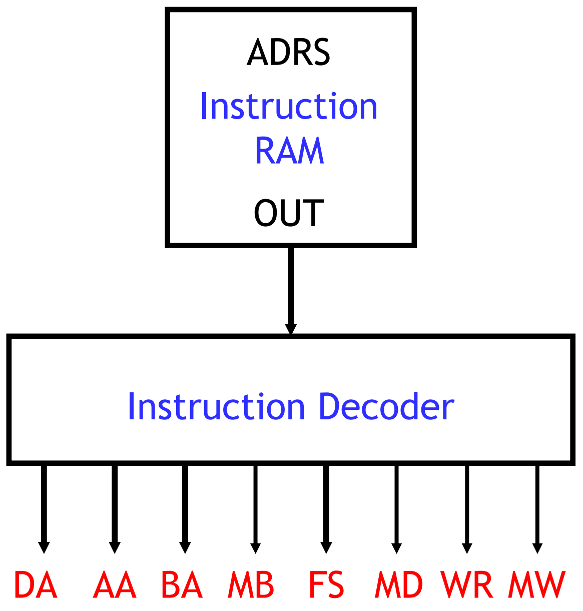


The whole processor

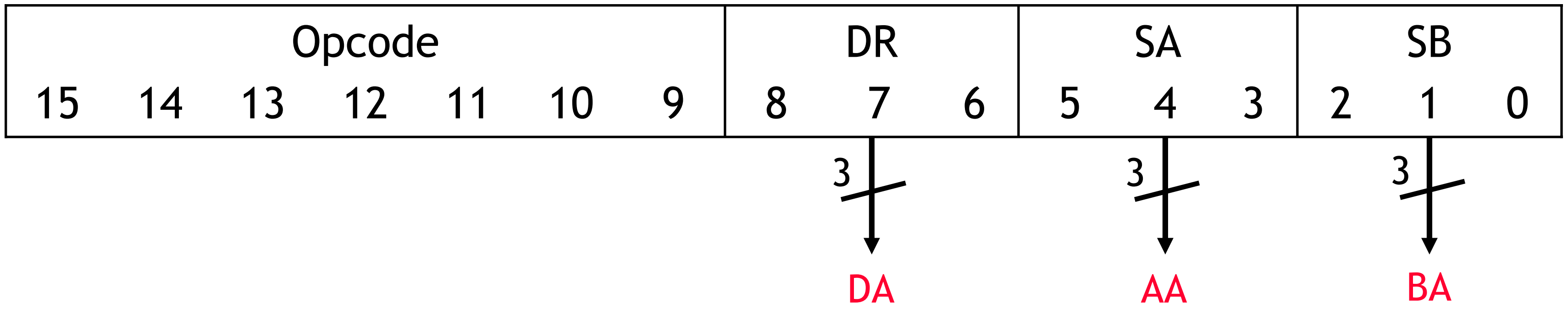


Implementing the instruction decoder

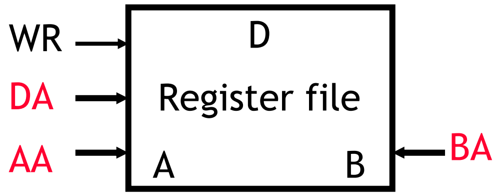
- The first thing we'll look at is how to build the instruction decoder.
- The instruction decoder's input is a 16-bit binary instruction I that comes from the instruction memory.
- The decoder's output is a set of control signals for the datapath.
  - AA, BA and MB select the sources for operations.
  - FS picks an ALU function.
  - MW = 1 to write to RAM.
  - MD, WR and DA allow data to be written back to the register file.
- We'll see how these signals are generated for each of our instructions.



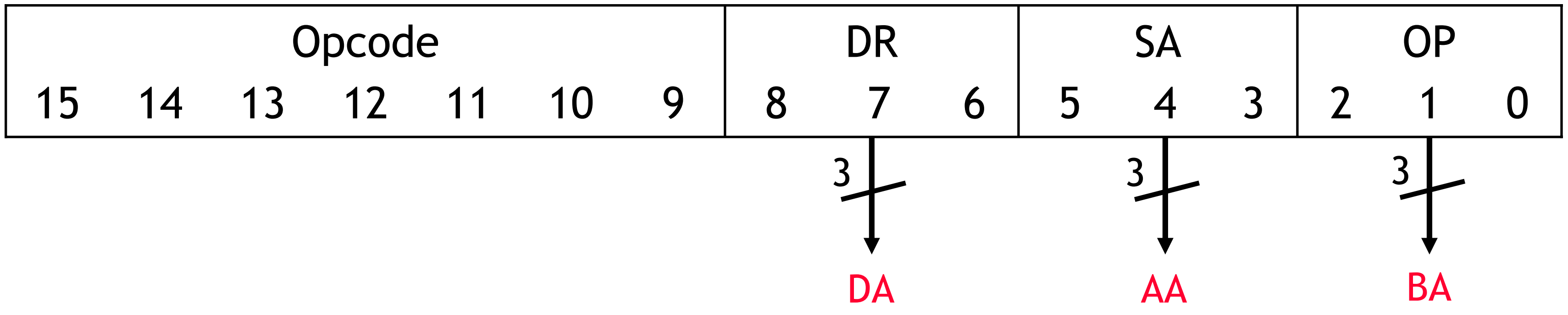
Generating DA, AA, BA



- We designed our encodings so the register file addresses DA, AA and BA can be taken directly out of the 16-bit binary instructions.
  - Instruction bits 8-6 are the datapath destination register input, DA.
  - Bits 5-3 are fed directly to AA, the first register file source.
  - Bits 2-0 are connected directly to BA, the second register source.
- This clearly works for register-format instructions as shown above.



Don’t-care conditions



- But in immediate-format instructions, bits 2-0 hold a constant operand as shown above, and not a second source register!
  – However, immediate instructions only use one source register, so the control signal BA would be a don’t care condition anyway.
  – Similarly, jump and branch instructions require neither a destination register nor a second source register.
- So it’s always okay to extract DA, AA and BA directly from the instruction.

$$DA_2\,DA_1\,DA_0 = I_8\,I_7\,I_6$$
$$AA_2\,AA_1\,AA_0 = I_5\,I_4\,I_3$$
$$BA_2\,BA_1\,BA_0 = I_2\,I_1\,I_0$$

Generating MB, MD, WR and MW

- Last time we divided our instructions into four categories, saying that the instructions in each category would require similar control signals.

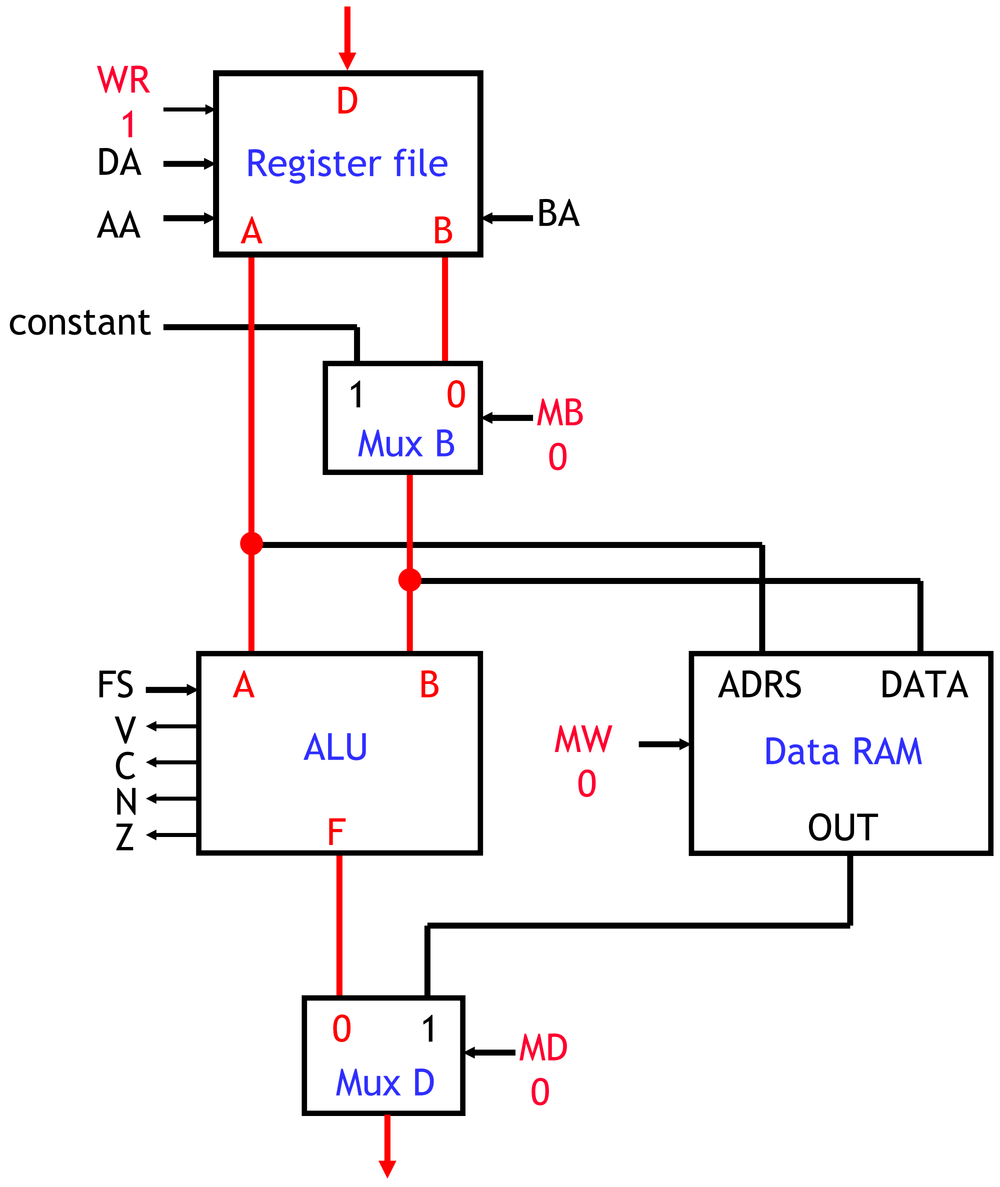
Instruction category	Opcode bits 6-5
Register-format ALU operations	00
Data transfer operations	01
Immediate ALU operations	10
Branches and jumps	11

- Let’s take a look at the datapath control signals needed for the different categories here.

Register-format ALU operations

ADD R1, R2, R3

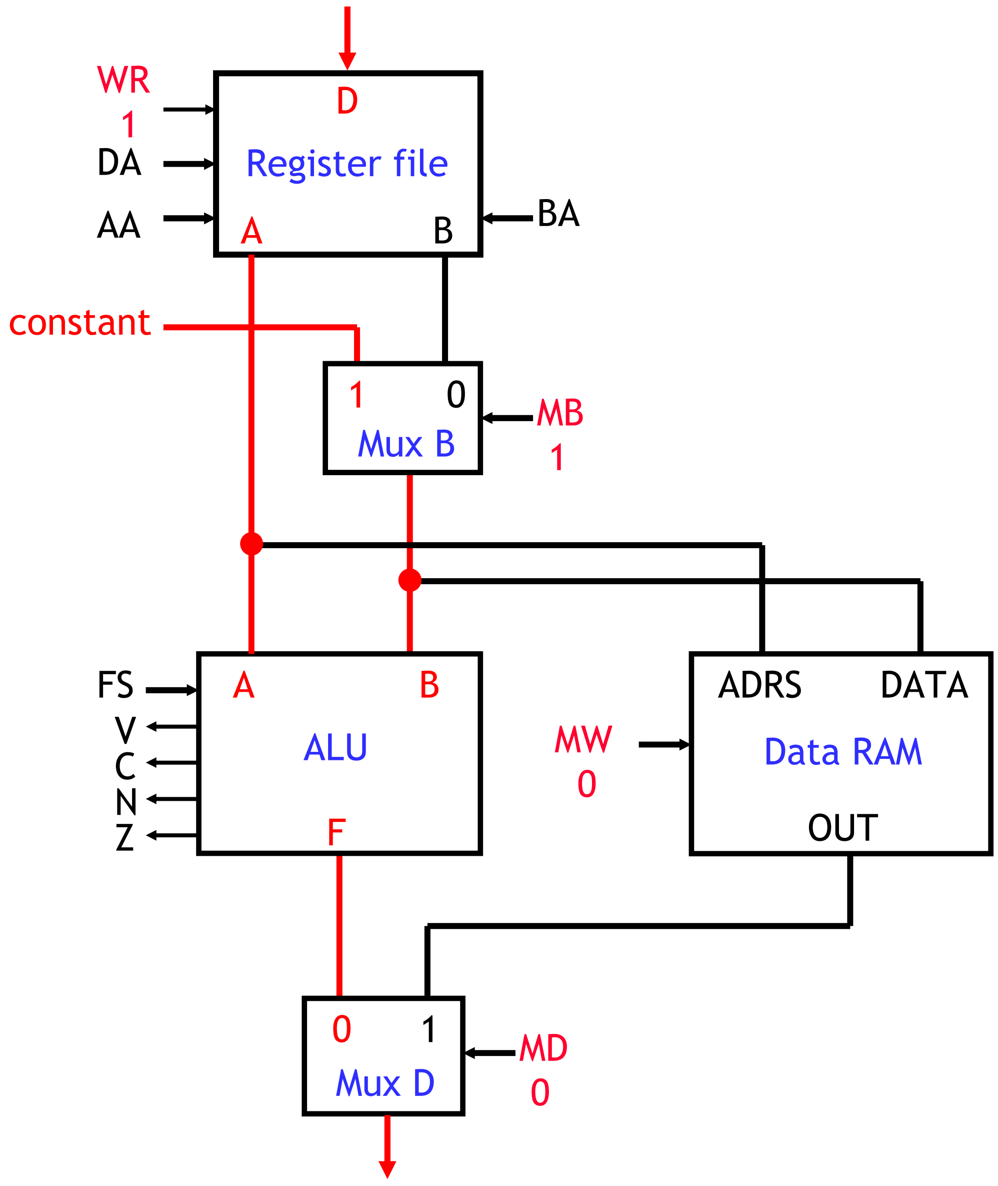
- All register ALU operations need the same values for the following control signals.
  - MB = 0 since the operands all come from the register file.
  - MD = 0 and WR = 1 to save the ALU result back into a register.
  - MW = 0 ensures that RAM is not modified.



Immediate-format ALU operations

SUB R1, R2, #2

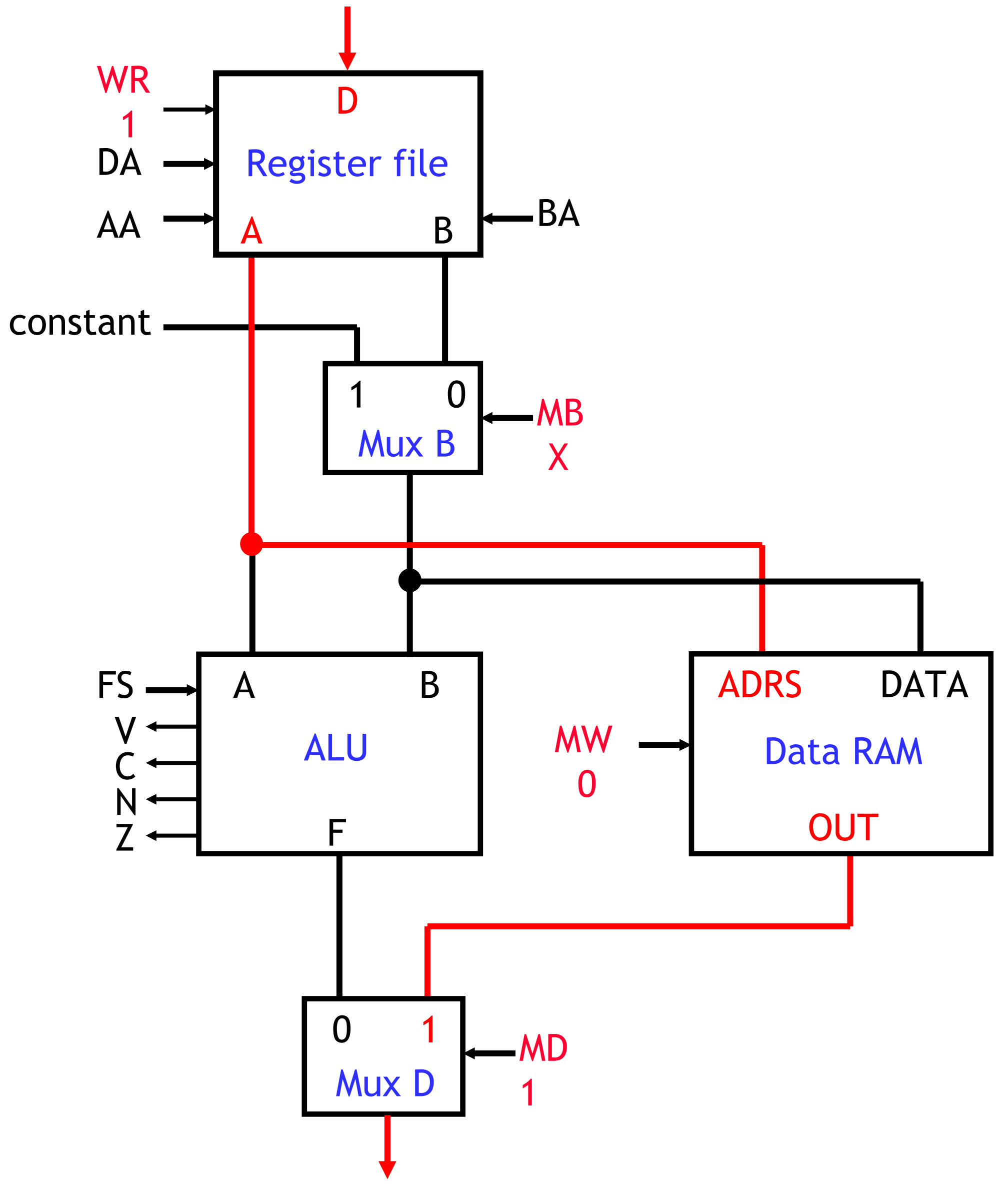
- Immediate ALU operations are similar, except for the MB signal.
  - MB = 1 since the second operand is a constant.
  - MD = 0 and WR = 1 to save the ALU result back into a register.
  - MW = 0 ensures that RAM is not modified.



Memory read

LD (R1), R2

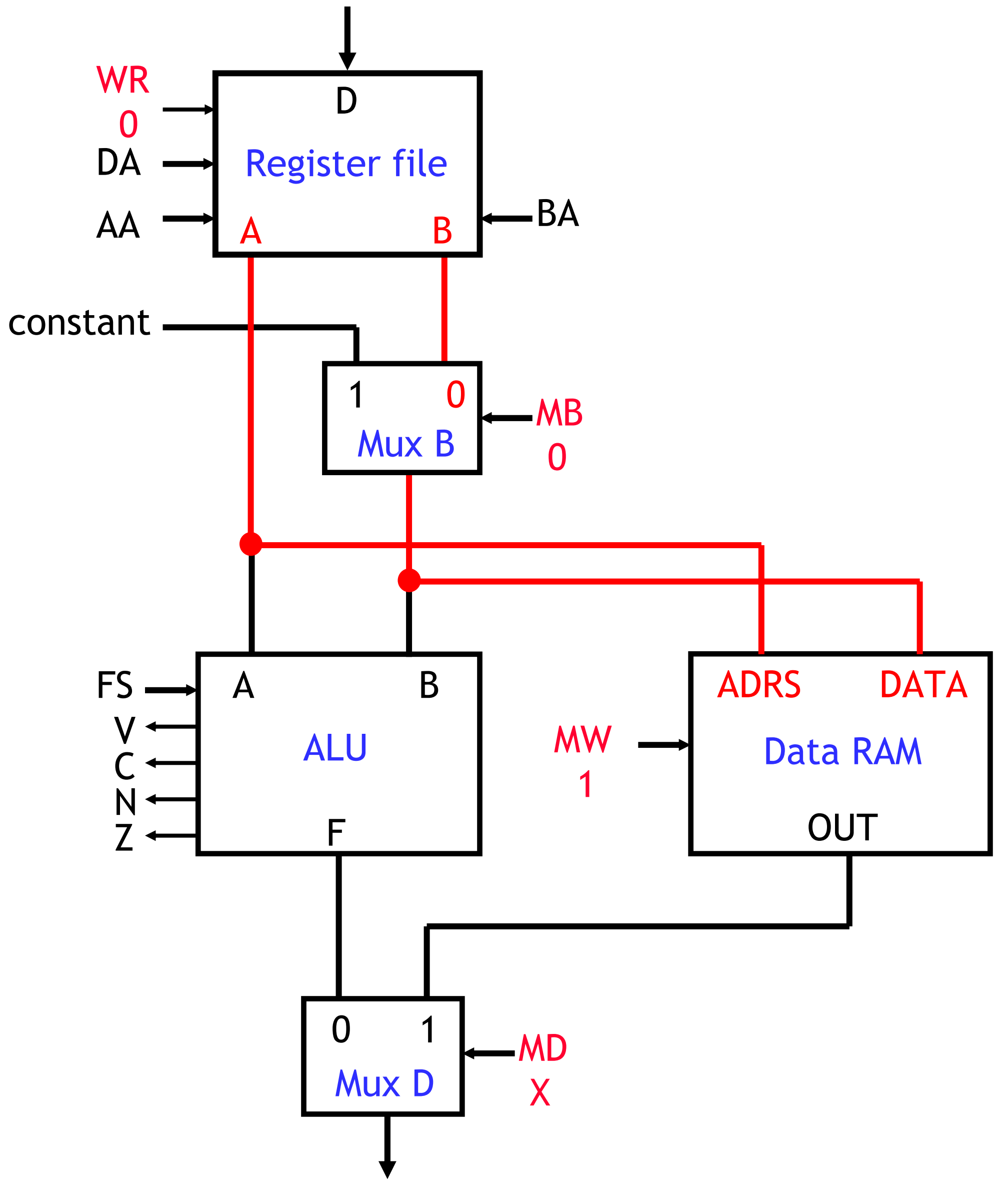
- A memory read requires the following datapath signals.
  - MB = X since there is no second operand.
  - MD = 1 and WR = 1 so the RAM output can be written to the register file.
  - MW = 0 because we are only reading from RAM.



Memory write

ST (R1), R2

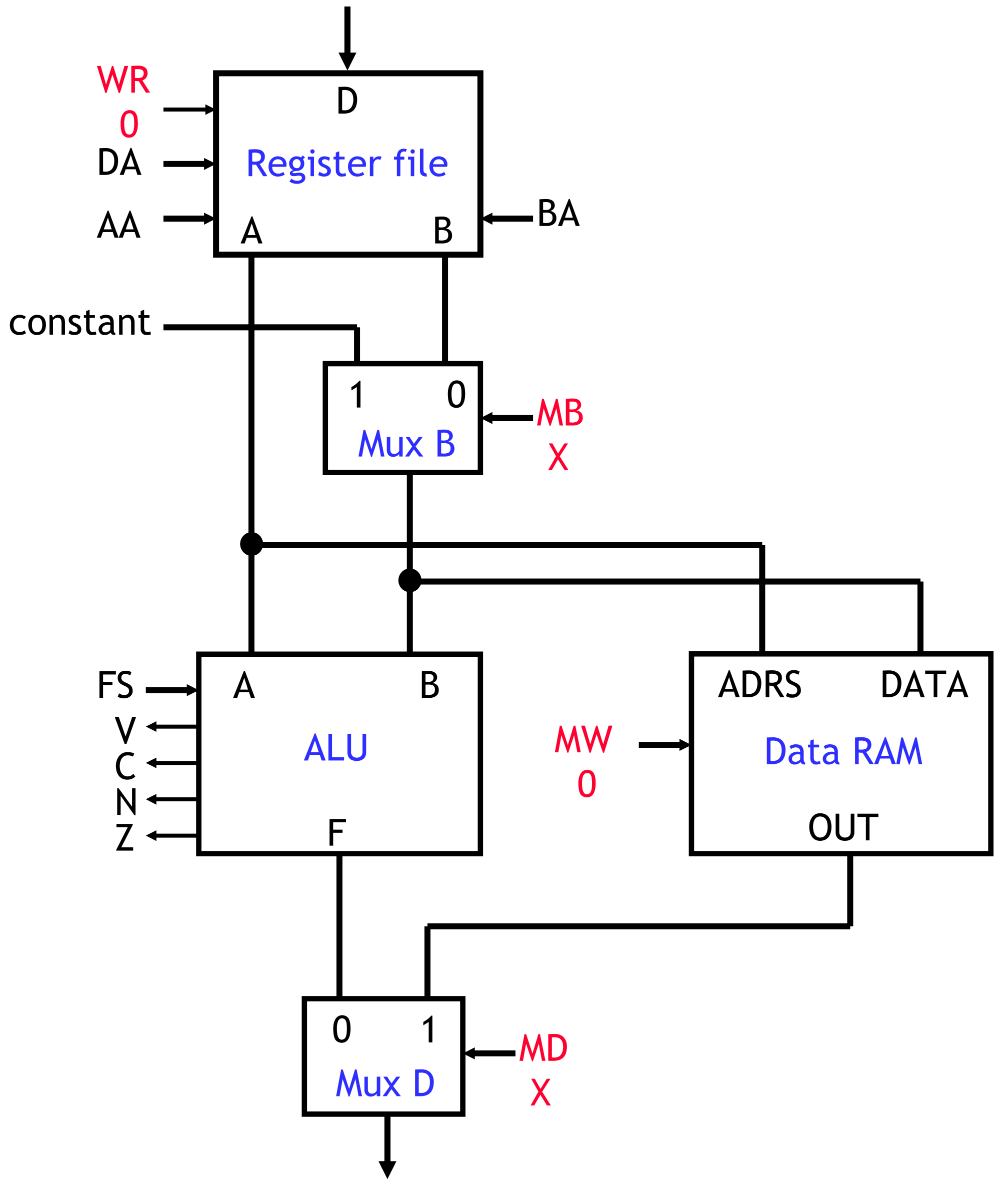
- A memory write needs the following datapath signals.
  - MB = 0 because the data to store comes from the register file.
  - MD = X and WR = 0, since none of the registers are changed.
  - MW = 1 to update RAM.



Jumps and branches

```
BZ   R2, +19
JMP  -5
```

- Jumps and branches shouldn't alter the registers or memory, so we must be careful to set WR and MW explicitly.
  - MB and MD are both don't cares.
  - WR = 0 ensures none of the registers are changed.
  - MW = 0 ensures nothing in memory is changed either.



MB, MD, WR and MW

- The following table summarizes the correct values of MB, MD, WR and MW for each of the different instruction categories we defined.

Category		Control signals			
		MB	MD	WR	MW
Register ALU instructions		0	0	1	0
Data transfer operations	ST	0	X	0	1
	LD	X	1	1	0
Immediate ALU instructions		1	0	1	0
Branches and jumps		X	X	0	0

- This is the sense in which these categories contain “similar” instructions.

Generating MB, MD, WR, and MW

Category		Instruction bits			Control signals			
		15	14	13	MB	MD	WR	MW
Register ALU instructions		0	0	X	0	0	1	0
Data transfer operations	ST	0	1	0	0	X	0	1
	LD	0	1	1	X	1	1	0
Immediate ALU instructions		1	0	X	1	0	1	0
Branches and jumps		1	1	X	X	X	0	0

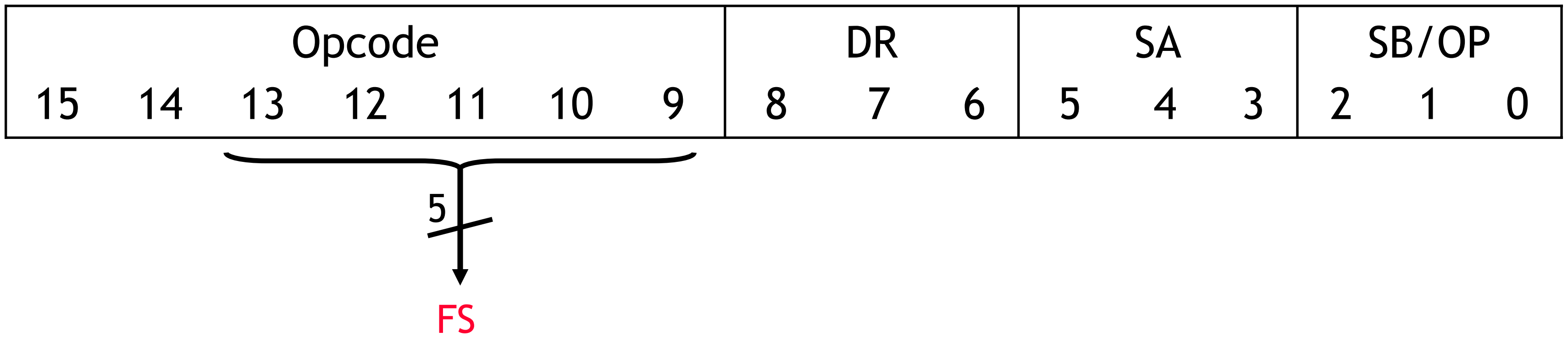
- Remember that instructions in the same category all begin with the same two or three opcode bits.
- Thus, the datapath control signals MB, MD, WR and MW can be expressed as functions of the first three opcode bits, or instruction bits I_{15} to I_{13}.

$MB = I_{15}$ $MD = I_{14}$

$WR = I_{14}' + I_{15}' I_{13}$ $MW = I_{15}' I_{14} I_{13}'$

Generating FS

- Yesterday we used the ALU function selection code as the last five bits in the opcodes for register- and immediate-format ALU instructions.
- For example, a register-based XOR has the opcode 0001100.
  - The first two bits 00 indicate a register-based ALU instruction.
  - 01100 is the ALU function selection code for the XOR operation.
- So we can send the function selection code from a register or immediate format instruction directly to the datapath's FS control input.



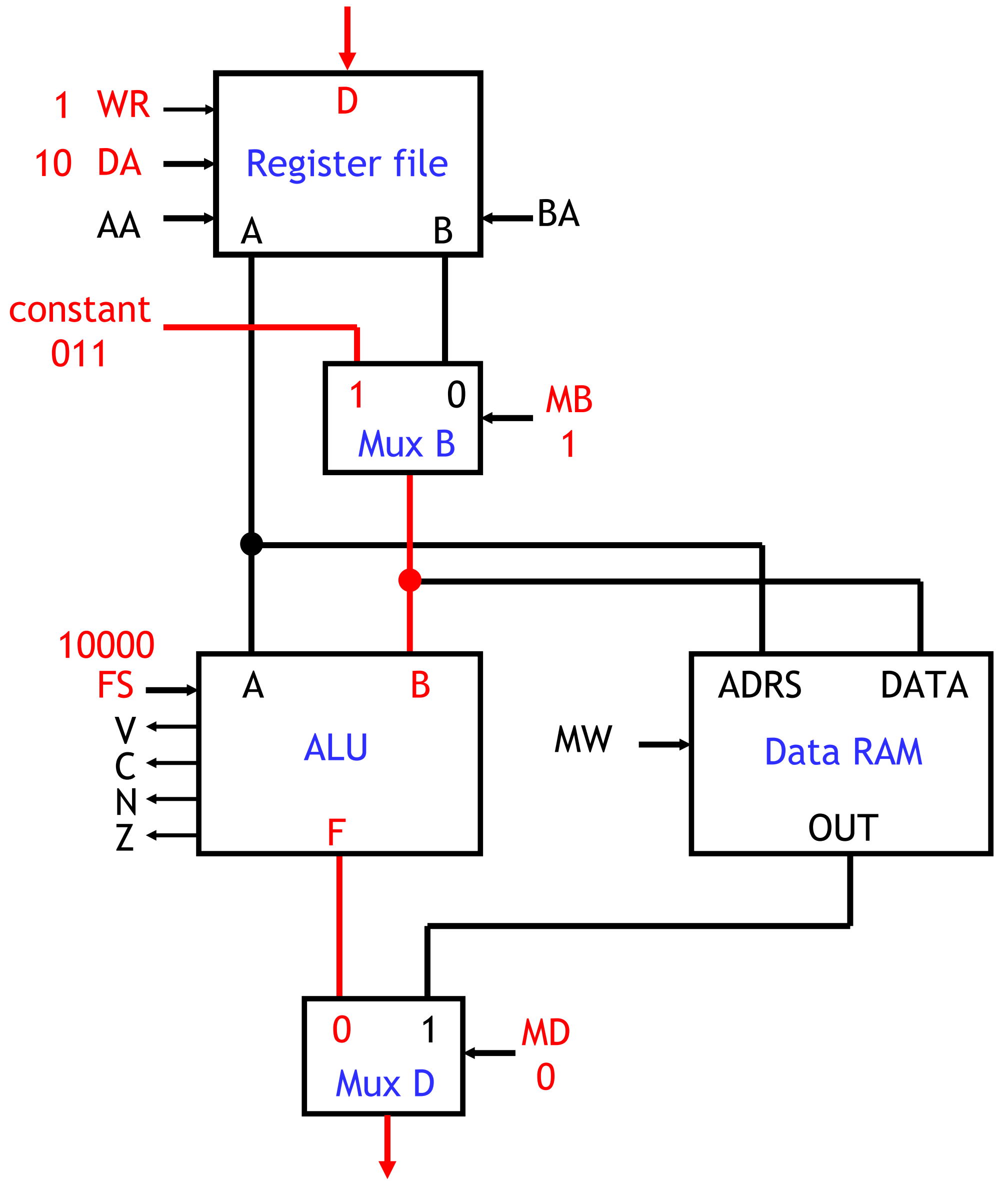
$$FS_4\, FS_3\, FS_2\, FS_1\, FS_0 = I_{13}\, I_{12}\, I_{11}\, I_{10}\, I_9$$

Immediate loads

- FS should be a don't-care for most of the other instructions, which do not involve the ALU.
- However the ALU *is* needed for immediate load operations.

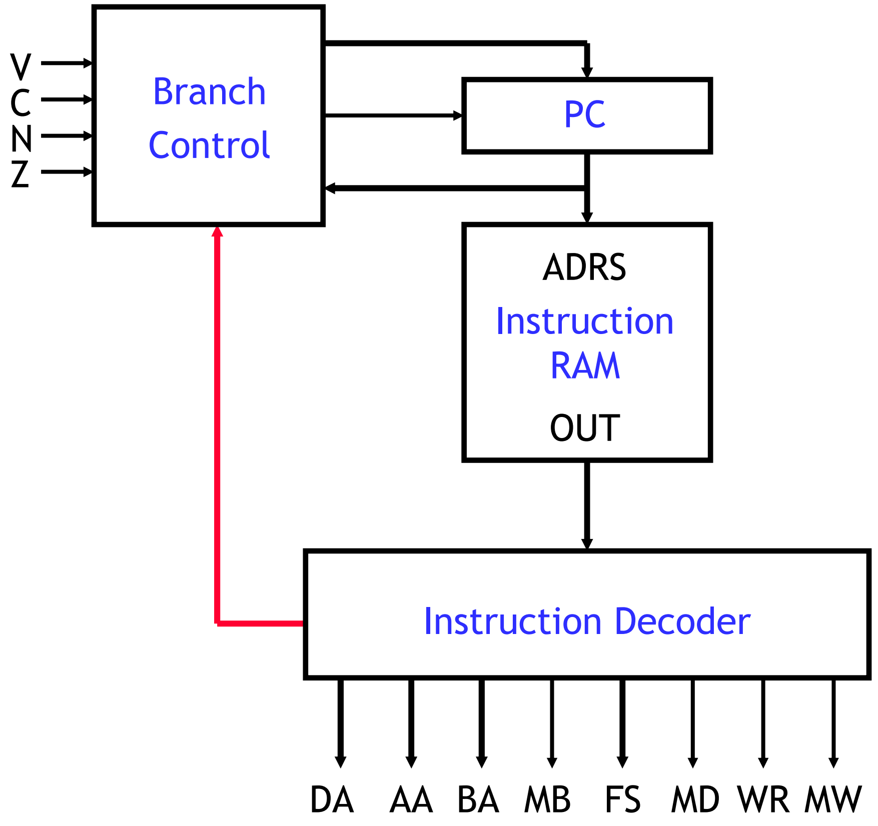
LD R2, #3

- The constant value must pass through Mux B, the ALU, and Mux D before it gets back to the register file.
- The easiest way to handle this is to treat immediate loads as immediate ALU operations with the opcode 1010000, using the ALU transfer operation "G = B."



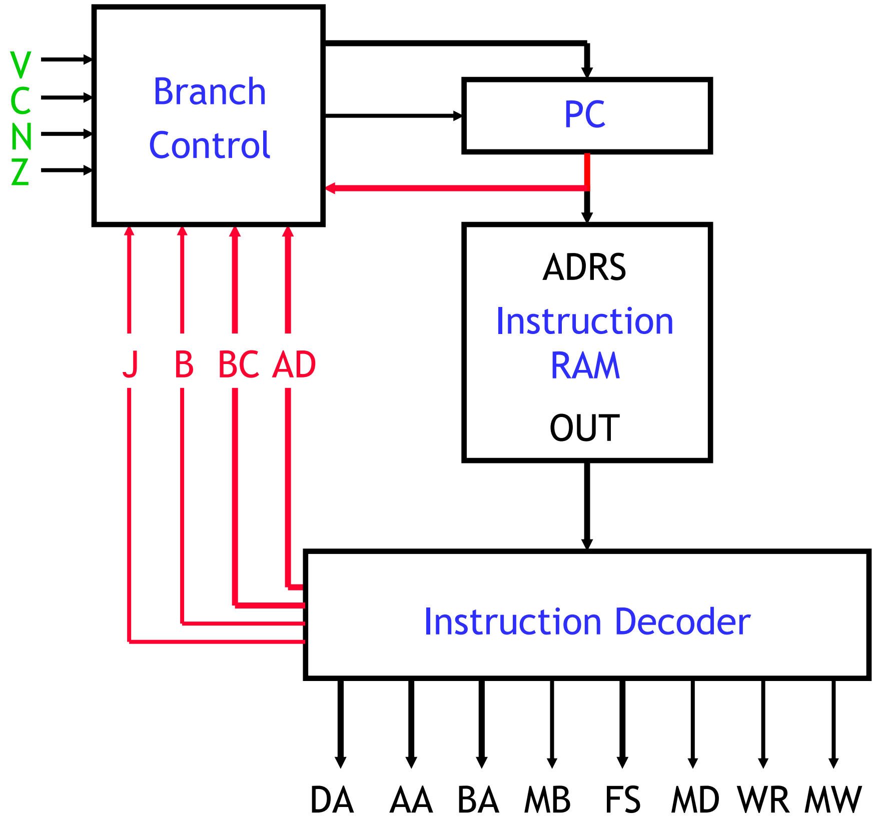
The branch control unit

- Next, let's see how to manage the control flow of a program.
- The branch control unit needs a lot of information about the current instruction.
  - Whether it's a jump, a branch, or some other instruction.
  - For branches and jumps, the target address.
  - For branches, the exact branch condition.
- All of this can be generated by the instruction decoder, which already is processing the instruction words.



Branch control unit inputs

- Inputs J and B will be true if the current instruction is a jump or branch, respectively.
- BC will indicate the exact branch condition, for branch instructions.
- AD will be the address offset, used for both branches and jumps. The current PC is also needed, since we use PC-relative addressing.
- Status bits V, C, N and Z come from the ALU in the datapath, and will determine whether or not a branch is taken.



Branch	Code
BC	000
BN	001
BV	010
BZ	011

Branch	Code
BNC	100
BNN	101
BNV	110
BNZ	111

Generating J and B

- The instruction decoder generates J and B from instruction opcodes.

Instruction bits

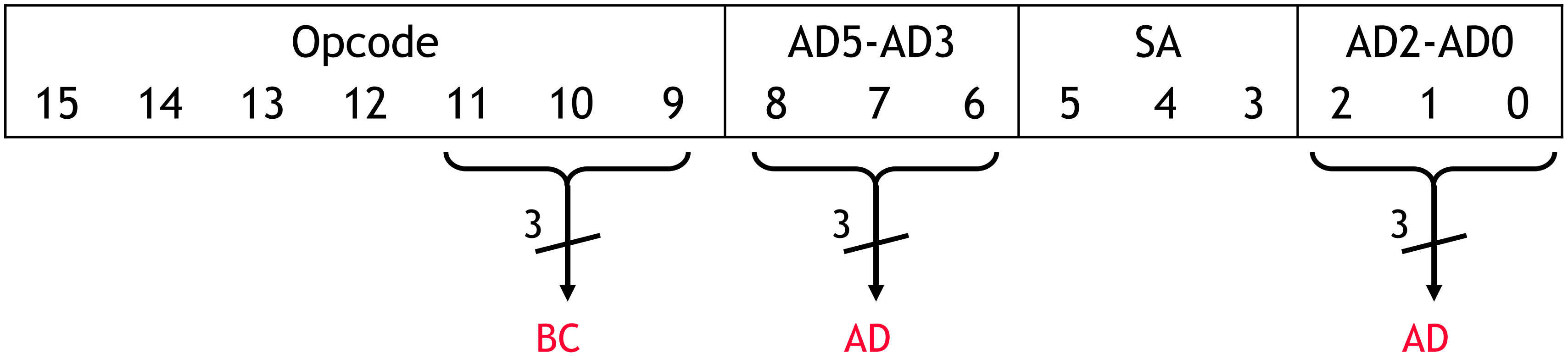
	15	14	13	12	11	10	9
Branches	1	1	0	X	BC		
JMP	1	1	1	XXXX			

- Remember that branch opcodes all begin with 110, and the opcode for JMP begins with bits 111.
- So it's easy to see the equations for J and B.

$J = I_{15}\, I_{14}\, I_{13}$ $B = I_{15}\, I_{14}\, I_{13}'$

Generating BC and AD

- We defined the branch opcodes so they already contain the branch type, so BC can come straight from the instruction opcode.
- AD can also be taken directly out of the instruction.

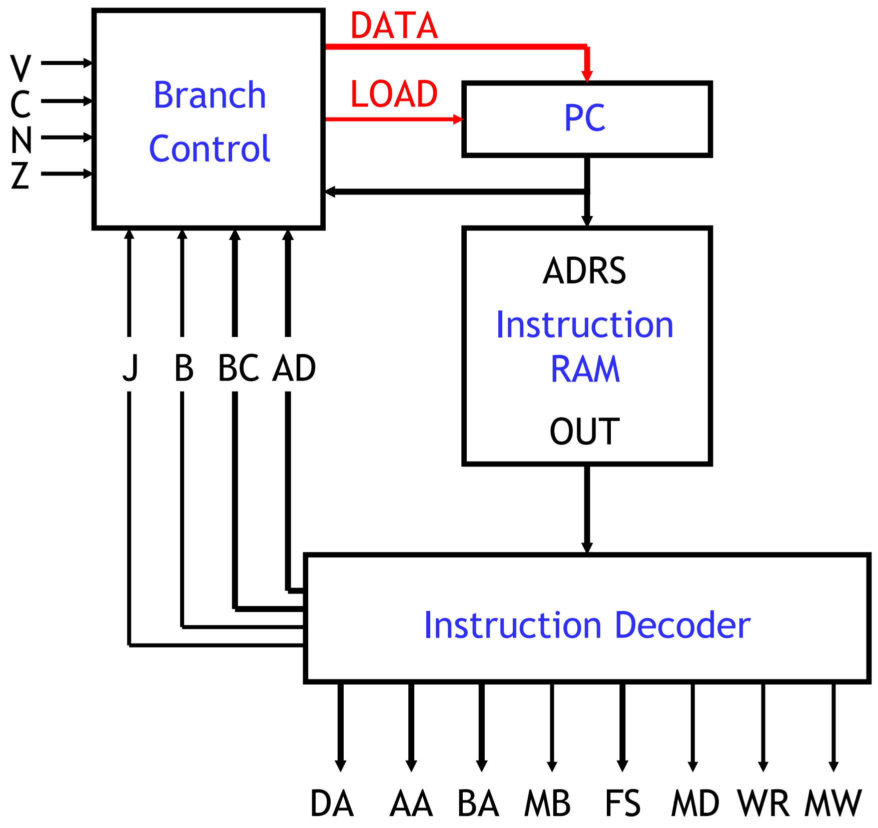


$$BC_2\,BC_1\,BC_0 = I_{11}\,I_{10}\,I_9$$

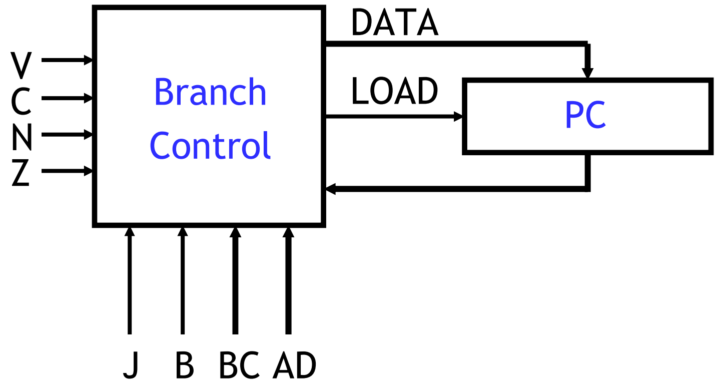
$$AD_5\,AD_4\,AD_3\,AD_2\,AD_1\,AD_0 = I_8\,I_7\,I_6\,I_2\,I_1\,I_0$$

Branch control unit outputs

- The branch control unit uses all of its inputs to generate just two outputs for controlling the PC.
- A LOAD signal will be 0 when the PC should just increment, or 1 when the PC has to load a target address for a branch or jump.
- A DATA value contains the target address for the PC, for branches that are taken and jumps.

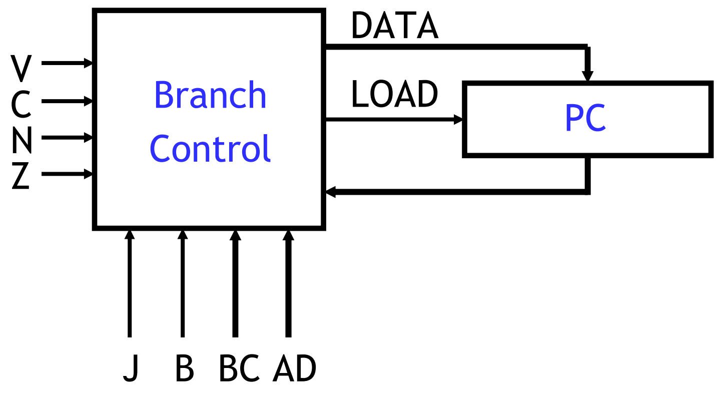


Generating LOAD and DATA



- There are three possible next states of the program counter, depending on the values of J and B.
- If J = 0 and B = 0, the current instruction is *not* a jump or branch, so the PC should just increment and go on to the next instruction.
- If J = 1, the current instruction is JMP.
  - We use PC-relative addressing, so the branch control unit should add the address offset AD to the current value of PC, and store that back into the program counter.
  - Remember that AD is signed, so we can jump forwards or backwards.

Generating LOAD and DATA for branches



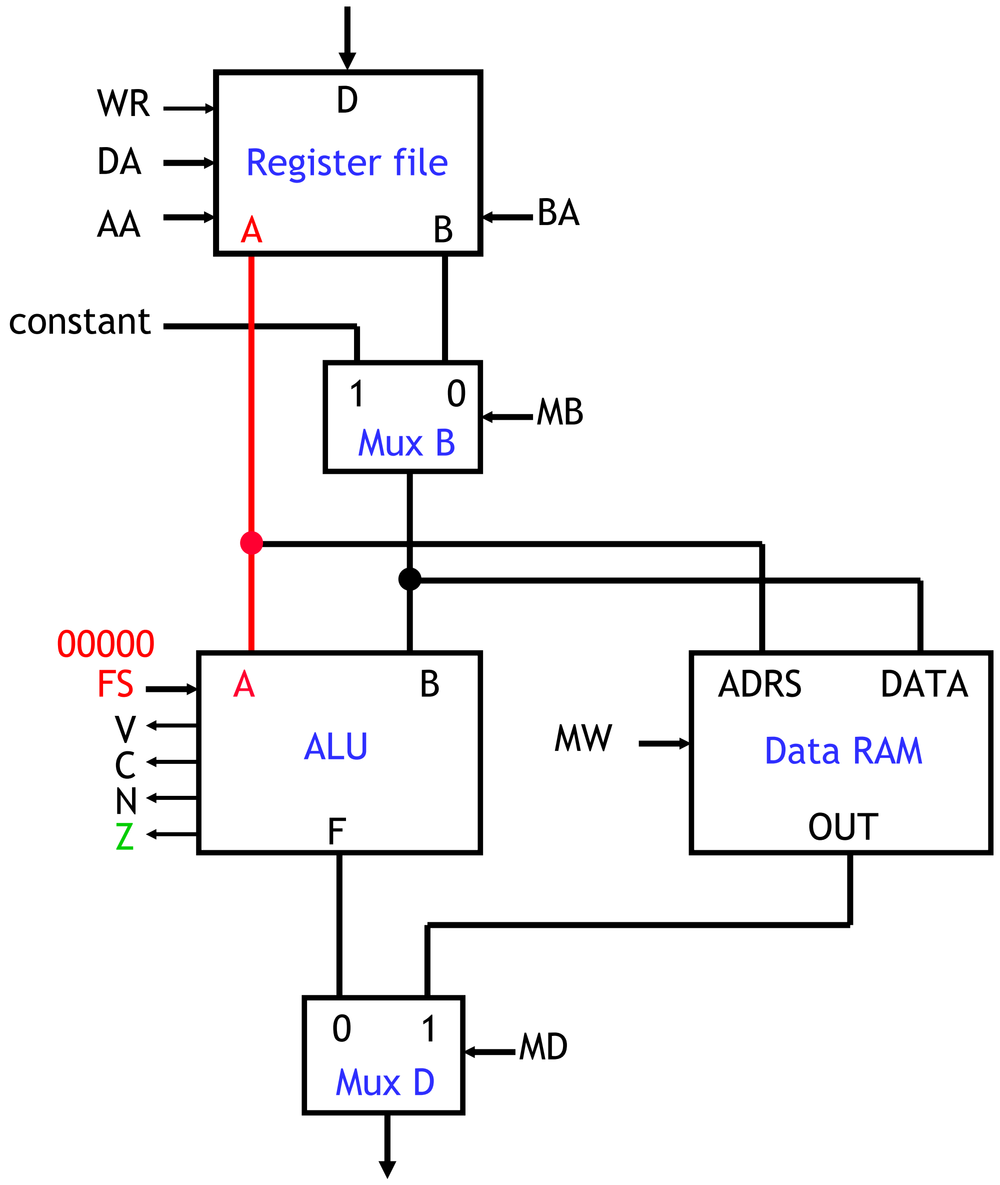
- If B = 1, the current instruction is a conditional branch.
- The branch control unit first determines if the branch should be taken.
  - It checks the type of branch (BC) and the status bits (VCNZ).
  - For example, if BC = 011 (branch if zero) and Z = 1, then the branch condition is true and the branch should be taken.
- Then the branch control unit sets the PC appropriately.
  - If the branch is taken, AD is added to the PC, just as for jumps.
  - Otherwise the PC is simply incremented, as for normal instructions.

FS for branch instructions

- One last detail is that branches depend on the ALU to produce the status bits appropriately.
- For example, in the instruction

  BZ R2, +19

  the contents of R2 have to go through the ALU just so that Z will be set appropriately.
- For our branches, we just need the ALU function "G = A" with FS = 00000 or 00111.



Summary

- Today we saw an outline of a processor's control unit hardware.
  - The program counter addresses an instruction memory, which holds a machine language program.
  - An instruction decoder takes instructions and generates the matching control signal inputs for the datapath and a branching unit.
  - The branch control unit handles instruction sequencing.
- Control unit implementations depend heavily on both the instruction set architecture and the datapath.
  - Careful selection of opcodes and instruction formats can make the control unit hardware simpler.
  - In MP4 you'll design the control unit for a slightly different processor.
- We now have a whole processor! This is the culmination of everything we did this semester, starting from those tiny little primitive gates.