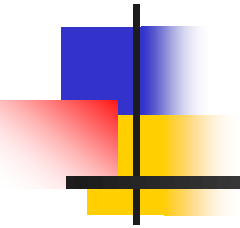
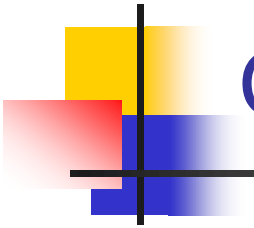


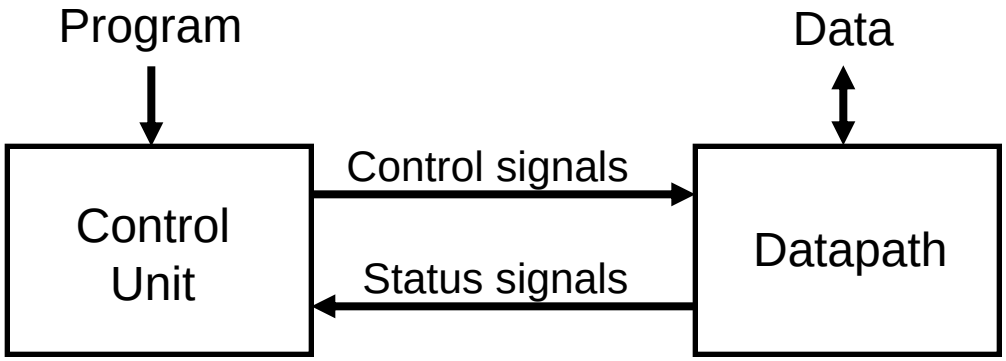
Processor Design Basics: Control Unit



Overview

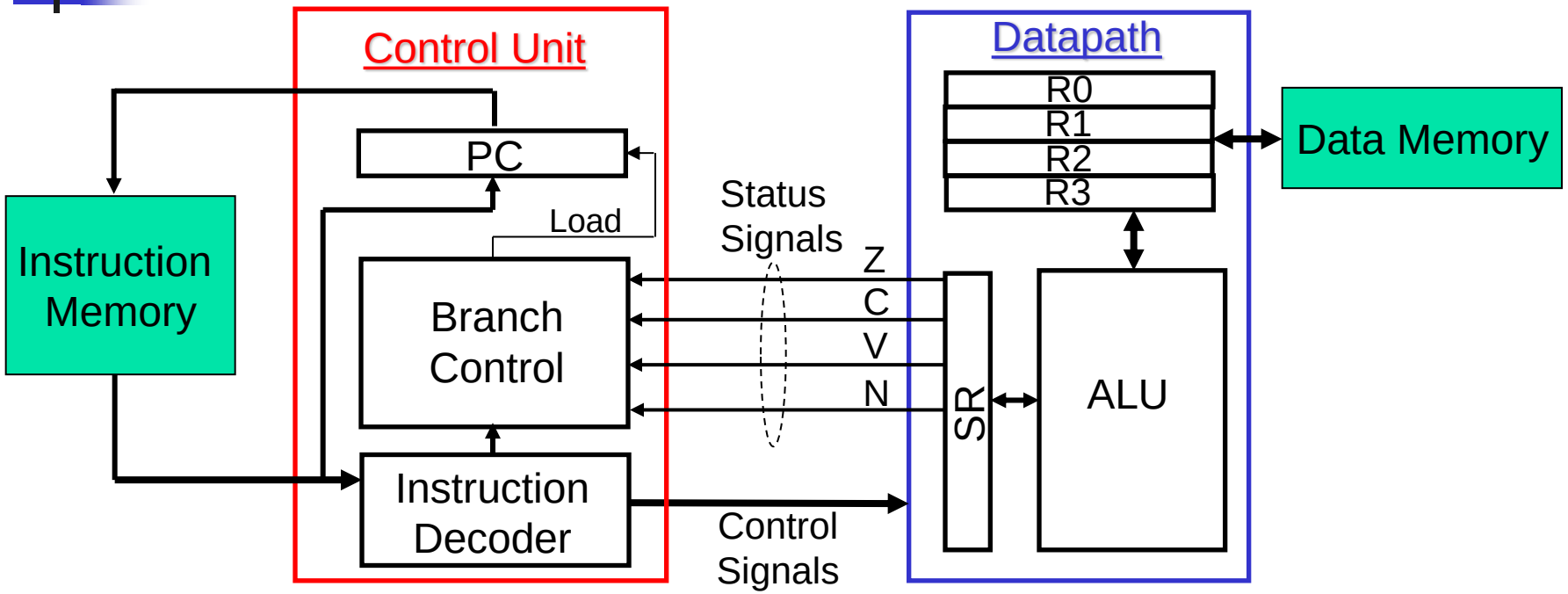
- From Assembly to Machine Language
  - Instruction Formats for our Processor
    - Register Format
    - Immediate Format
    - Branch Format
  - Selecting Instruction Opcodes
  - Complete Encoding of Instructions
- Control Unit Design
  - Program Counter
  - Instruction Decoder
  - Branch Control
- Summary

Block Diagram of a Generic Processor



- We have 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 to the processor.
- What's left to be discussed is the Control Unit that converts assembly language instructions into datapath control signals.
  - How assembly instructions can be represented in a binary format?
  - How to design a control unit for our simple processor?

Example of a Simple Processor



- Here we will look at the control unit which connects programs with the datapath.
  - It converts program instructions into control signals for the datapath
  - It executes program instructions in the correct sequence
- The datapath also sends information back to the control unit.
  - ALU status bits V, C, N, Z can be inspected by branch instructions to alter a program's control flow.

The Instruction Set of our Processor

- The design of the control unit starts by analyzing the instruction set of a processor.

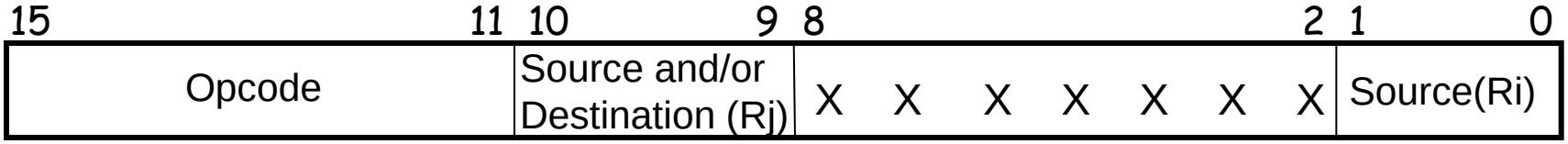
Instruction Type	Operation	Mnemonic	Operation	Status Bits	Description
Data Manipulation Instructions	**Register-format Arithmetic & Logic Operations**	LDR Rj, Ri	Rj ← Ri	Z, N	
		INC Rj, Ri	Rj ← Ri + 1	Z, N	
		DEC Rj, Ri	Rj ← Ri - 1	Z, N	
		ADD Rj, Ri	Rj ← Rj + Ri	C, V, Z, N	
		ADDC Rj, Ri	Rj ← Rj + Ri + C	C, V, Z, N	
		SUB Rj, Ri	Rj ← Rj + Ri' + 1	C, V, Z, N	
		AND Rj, Ri	Rj ← Rj ∧ Ri	Z, N	
		OR Rj, Ri	Rj ← Rj ∨ Ri	Z, N	
		XOR Rj,Ri	Rj ← Rj ⊕ Ri	Z, N	
		NOT Rj, Ri	Rj ← Ri'	Z, N	
	Register-format Shift Operations	SHL Rj, Ri	Rj ← Ri << 1	NO effect	
		SHR Rj, Ri	Rj ← Ri >> 1	NO effect	
Data Movement Instructions	**Memory write** (from registers)	ST (Rj), Ri	Mem[R0|Rj] ← Ri	NO effect	
	Memory read (to registers)	LD Rj, (Ri)	Rj ← Mem[R0|Ri]	NO effect	
	Immediate transfer operations	LDI Rj, #const8	Rj ← const8	NO effect	
		STI (Rj), #const8	Mem[R0|Ri] ← const8	NO effect	
Control Flow Instructions	**Branches**	BZ #offset11	PC ← PC + offset11	NO effect	
		BNZ #offset11	PC ← PC + offset11	NO effect	
		BC #offset11	PC ← PC + offset11	NO effect	
		BNC #offset11	PC ← PC + offset11	NO effect	
		BV #offset11	PC ← PC + offset11	NO effect	
		BNV #offset11	PC ← PC + offset11	NO effect	
		BN #offset11	PC ← PC + offset11	NO effect	
		BNN #offset11	PC ← PC + offset11	NO effect	
	Jump	JMP Rj, Ri	PC ← Rj|Ri	NO effect	

From Assembly to Machine Language

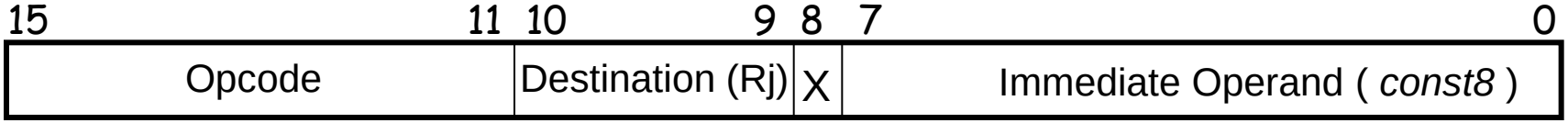
- We must define a machine language
  - binary representation of the assembly instructions for our processor
- The instructions of our processor can be divided into 3 groups, which have different operands and will need different representations/formats.
  - Register format instructions require two registers (**Rj** and **Ri**) from the register file to be specified.
  - Immediate format instructions require one register (**Rj**) from the register file and one 8-bit constant operand (***const***).
  - Branch format instructions require one 11-bit constant address (***offset***) to be specified.
- For the three different instruction formats, it is best to make their binary representations as similar as possible
  - This will make the control unit hardware simpler
- We have briefly discussed the instruction formats when we discussed the Instruction Set Architecture of our processor.

Instruction Formats for our Processor

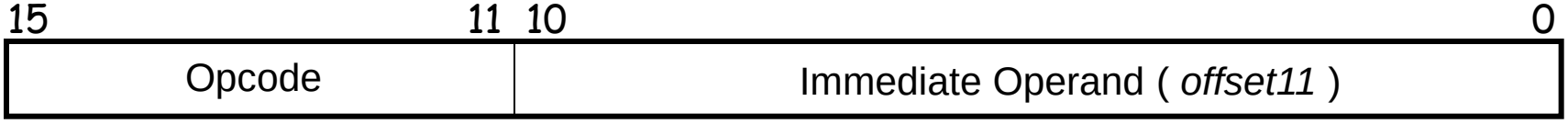
- **Register format (Format1):** for Arithmetic&Logic, Shift, Memory, and Jump instructions:



- **Immediate format (Format2):** for Immediate Transfer instructions:

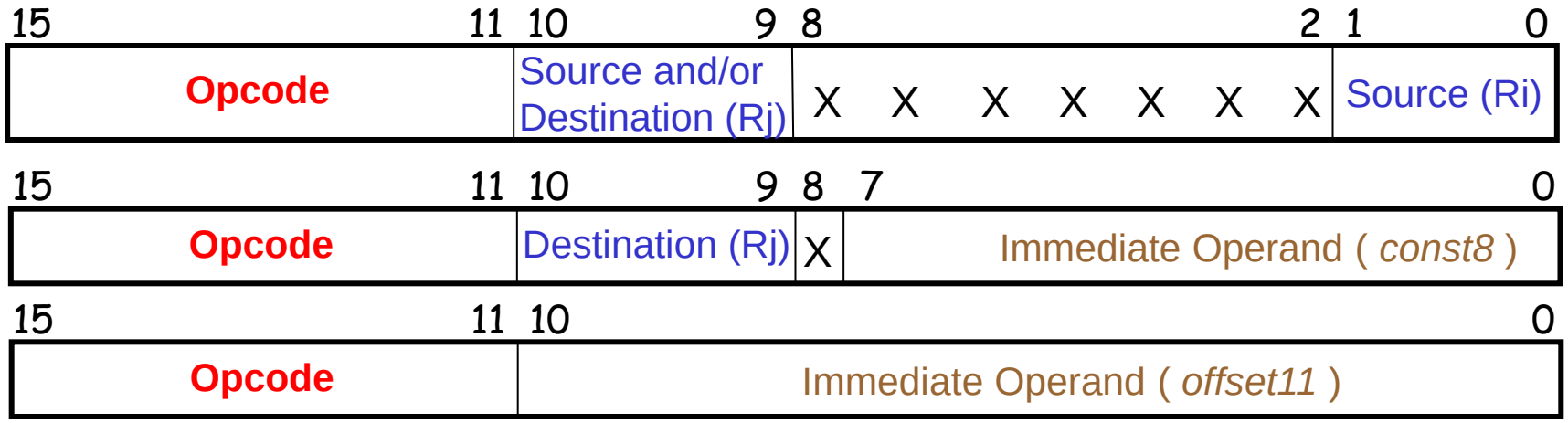


- **Branch format (Format3):** for Branch instructions:

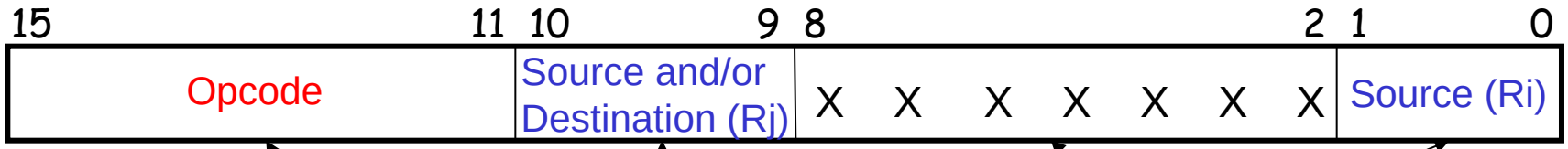


Instruction Formats for our Processor (cont.)

- Each instruction format contains 16 bits because
  - The program memory is 16-bit wide
  - Each program memory cell stores one instruction
- The 5-bit **Opcode** (bits 15 to 11) encodes the operation performed by each instruction:
  - We have 25 instructions, i.e., 25 distinct operations, therefore
  - At least 5-bit Opcode needed to have a unique code for each operation
- The rest of the bits (10 to 0) specify **registers** and/or **immediate operand**.
  - Opcode determines the exact meaning of the rest of the bits in the instruction
  - Thus, the processor first "looks" at the Opcode to identify the instruction format and the operation to be performed.



Register Format



Encoded with 5 bits

Encoded with 2 bits

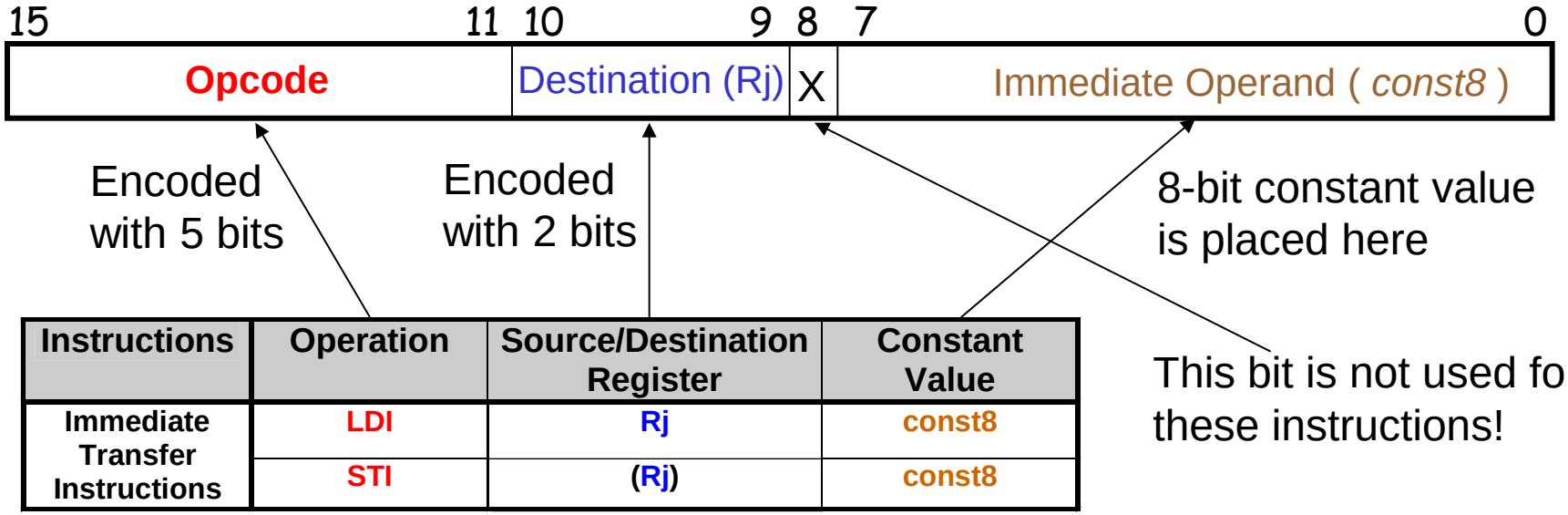
Encoded with 2 bits

These bits are not used for these instructions!

Instructions	Operation	Source/Destination Register	Source Register
Arithmetic & Logic Instructions	LDR	Rj	Ri
	INC	Rj	Ri
	DEC	Rj	Ri
	ADD	Rj	Ri
	ADDC	Rj	Ri
	SUB	Rj	Ri
	AND	Rj	Ri
	OR	Rj	Ri
	XOR	Rj	Ri
	NOT	Rj	Ri
Register-format Shift Operations	SHL	Rj	Ri
	SHR	Rj,	Ri
Memory write (from registers)	ST	(Rj)	Ri
Memory read (to registers)	LD	Rj	(Ri)
Jump	JMP	Rj	Ri

NOTE: We encode the registers with 2 bits because we have only 4 registers (R0 to R3) in our processor's register file.

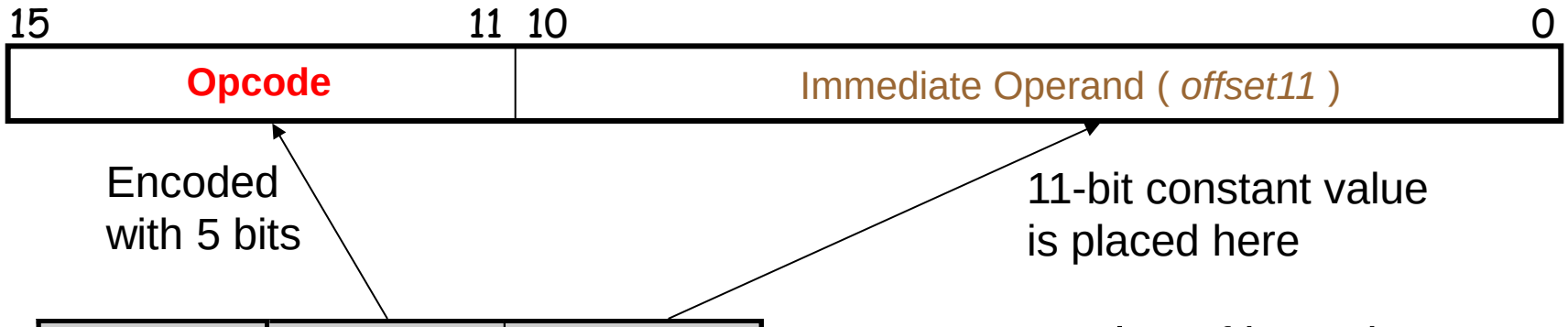
Immediate Format



Instructions	Operation	Source/Destination Register	Constant Value
Immediate Transfer Instructions	LDI	Rj	const8
	STI	(Rj)	const8

NOTE: We have to use only 8-bit constants because the 4 registers (R0 to R3) in our processor's register file and the data memory are 8-bit wide, i.e., they can store only 8-bit values.

Branch Format



Instructions	Operation	Constant Value
Branch Instructions	BZ	offset11
	BNZ	offset11
	BC	offset11
	BNC	offset11
	BV	offset11
	BNV	offset11
	BN	offset11
	BNN	offset11

- Two examples of branch instructions:
  - **BZ # -578 PC ← PC - 578**
  - **BZ # +1022 PC ← PC + 1022**
- Branch format instructions include:
  - A 5-bit instruction opcode.
  - A 11-bit address field, for storing branch offsets.

IMPORTANT: ***offset11*** is treated as an 11-bit signed number, so you can branch up to 1023 addresses forward (2^{10}-1), or up to 1024 addresses backward (-2^{10}) !!!

This is called Program Counter **Relative** Branch

Example of PC-Relative Branch

- We will use PC-relative addressing for branches, where the operand specifies the number of addresses (offset) to branch from the current instruction.
- We can assume that each instruction occupies one word of memory.

```
    LDI R1, #0x35
    LDI R2, #0x9f
L:  DEC R2, R2
    INC R1, R1
    SUB R2, R1
    BC  L
    ST  (R3), R2
```

```
0x1000:  LDI R1, #0x35
0x1001:  LDI R2, #0x9f
0x1002:  DEC R2, R2
0x1003:  INC R1, R1
0x1004:  SUB R2, R1
0x1005:  BC #0x7FD  // PC ← PC + (-3)
0x1006:  ST (R3), R2
```

- To branch "backwards" the offset operand **L** should be (-3) represented is an 11-bit *signed 2's complement* number → (-3) = **0x7FD**
  - It is possible to branch either "forwards" or "backwards."
  - Branches are often used to implement loops; see some of the examples from previous lectures.

Selecting Instruction Opcodes

- How can we select binary opcodes for each possible instruction?
  - In general, “similar” instructions should have similar opcodes.
    - Again, this will lead to simpler control unit hardware.
  - We can divide our instructions into 7 different categories
    - Each category requires similar datapath control signals.
- We will assign opcodes so that all instructions in the same category will have the same first three opcode bits (bits 15-13 of the instruction).

Instruction Category	Opcode bits 15	14	13
Register-format ALU arithmetic operation	0	0	0
Register-format ALU propagate/shift operation	0	0	1
Register-format ALU logic operation	0	1	0
Data Movement operation	0	1	1
Conditional branch on flag = 0	1	0	0
Conditional branch on flag = 1	1	0	1
Jump	1	1	1

- **What about the rest of the Opcode bits?**

Register Format ALU Instructions

- For ALU instructions we select the Opcode to be the same as the ALU's function selection code (FS) discussed in previous lecture.
- The complete opcode is give in the table on the right.
- For example, a register-based SUB instruction has the opcode 00011.
  - The first three bits 000 indicate a register-based ALU arithmetic instruction.
  - 11 denotes the ALU arithmetic SUB function.
- A shift right instruction SHR has the opcode 00110.
  - 001 indicates a register-based ALU shift instruction.
  - 10 denotes a shift *right*.

INSTR	Opcode	Operation
INC	00000	F = B + 1
ADD	00001	F = A + B
ADDC	00010	F = A + B + Carry-in
SUB	00011	F = A + B' + 1
DEC	00100	F = B - 1
LDR	00101	F = B
SHR	00110	F = sr B (shift right)
SHL	00111	F = sl B (shift left)
AND	01000	F = A $\wedge$ B (AND)
OR	01001	F = A $\vee$ B (OR)
XOR	01010	F = A $\oplus$ B
NOT	01011	F = B'

Data Movement Instructions

- The complete opcode is give in the table on the right.
- For example, a data movement LD instruction has the opcode 01101.
  - The first three bits 011 indicate a data movement instruction.
  - The fourth bit 0 denotes *register/memory* data movement.
  - The fifth bit 1 denotes data is moved/*loaded to register from memory.*

INSTR	Opcode	Operation
ST	01100	Mem[R0|Rj] ← Ri
LD	01101	Rj ← Mem[R0|Rj]
STI	01110	Mem[R0|Rj] ← const8
LDI	01111	Rj ← const8

- A STI instruction has the opcode 01110.
  - 011 indicates data movement.
  - 1 denotes *immediate* data movement, i.e., *constant is moved to memory or register*.
  - 0 denotes *constant is moved/stored to memory.*

Branch and Jump Instructions

- The complete opcode is given in the tables on the right.
  - The first 3 bits determine the branch/jump category
  - The last 2 bits determine the branch condition
- The opcode of instruction JMP has unused bits.
  - There is only one kind of jump
- These unused bits allow for future expansion of the instruction set.
  - For instance, we might add other jump instructions with other addressing modes.

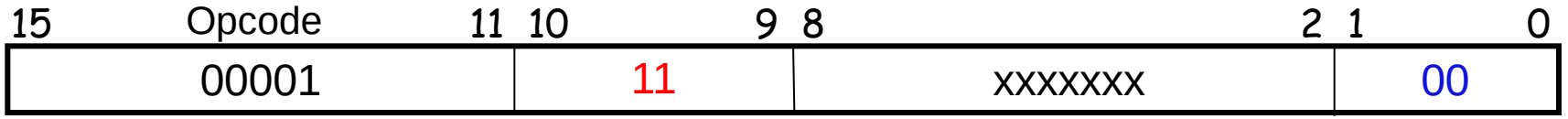
INSTR	Opcode	Operation
BNZ	10000	Branch if non-zero
BNC	10001	Branch if carry clear
BNV	10010	Branch if no overflow
BNN	10011	Branch if positive

INSTR	Opcode	Operation
BZ	10100	Branch if zero
BC	10101	Branch if carry set
BV	10110	Branch if overflow
BN	10111	Branch if negative

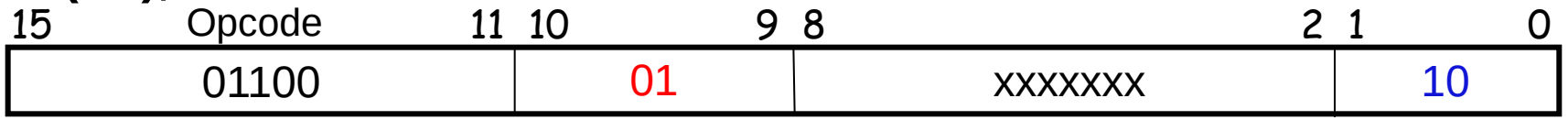
INSTR	Opcode	Operation
JMP	111xx	Jump

Complete Encoding of Instructions

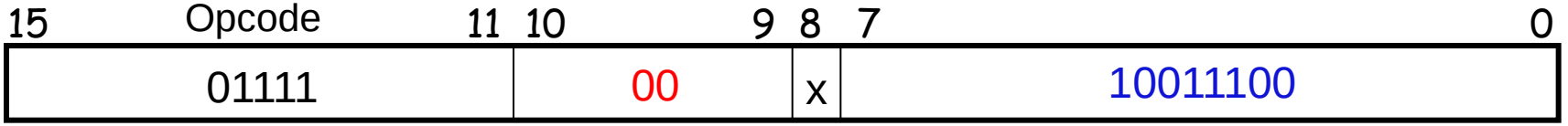
- Below we give the complete encoding of some of our processor instructions.
- **ADD R3, R0**



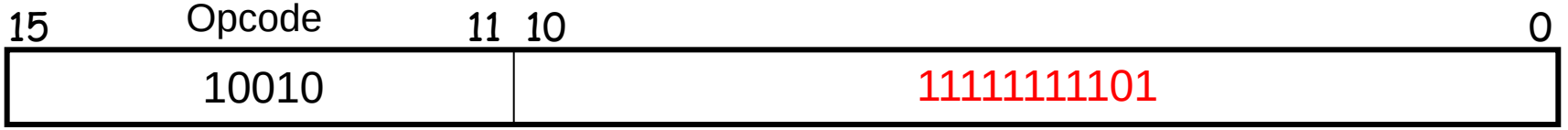
- **ST (R1), R2**



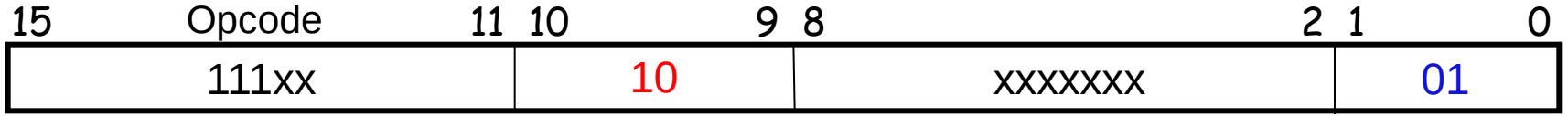
- **LDI R0, #0x9c**



- **BNV # -3**



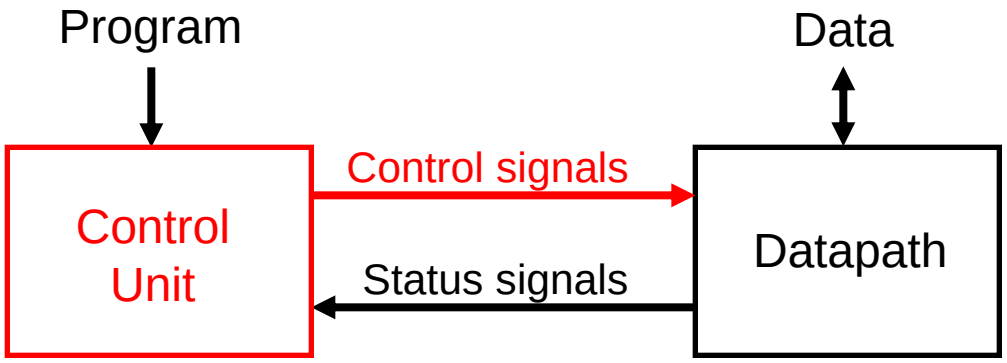
- **JMP R2, R1**



Instruction Encoding Summary

- So far, we have defined a binary machine language for the instruction set of our simple processor.
  - Different instructions have different operands and formats, but keeping the formats uniform will help simplify our hardware.
  - We also try to assign similar opcodes to “similar” instructions.
  - The instruction encodings and datapath are closely related.
    - Opcodes include ALU selection codes
    - The number of available registers determines the size of the code for register operands in instructions.
- This is just *one* example of how to define a machine language.
- Next slides will show you how to build a control unit corresponding to our datapath and instruction set. This will complete our processor!

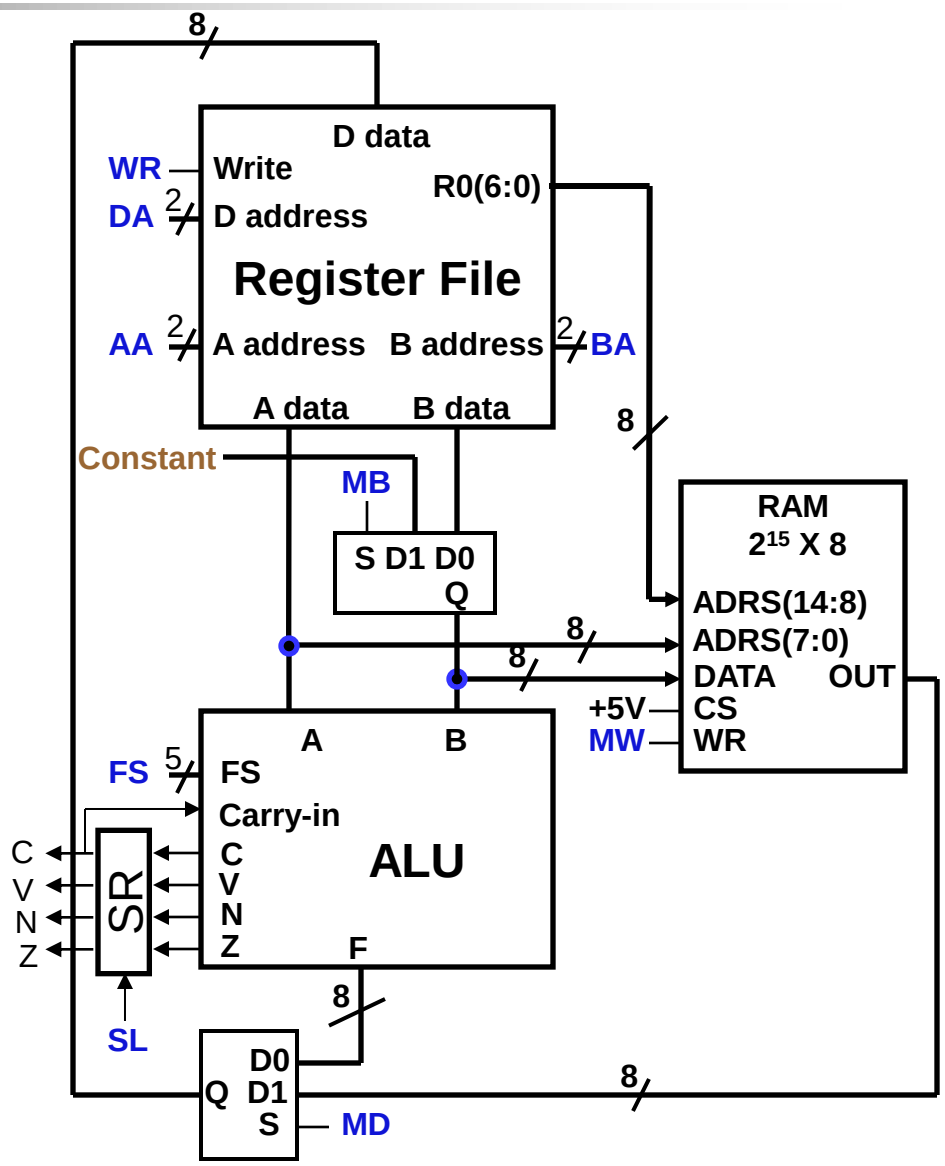
Again This General Picture …



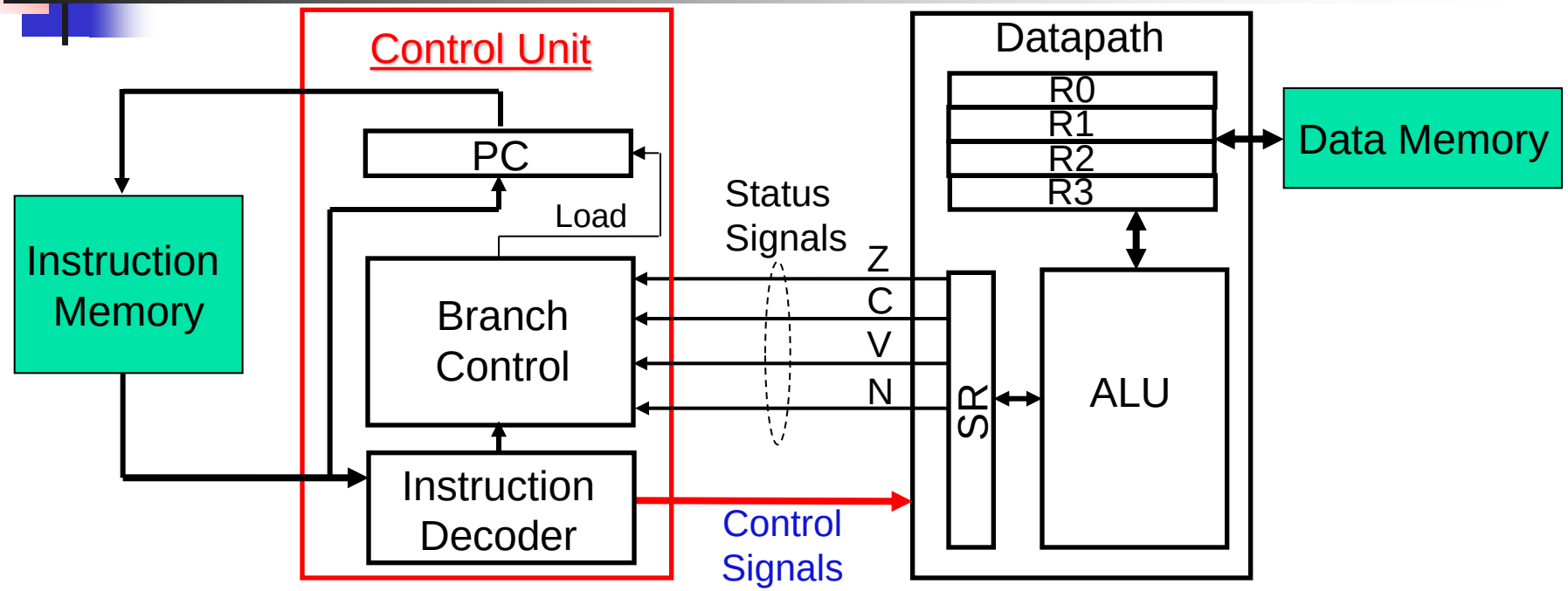
- The Control Unit converts binary instructions coming from a program into Datapath control signals.
- Before, we start designing the control unit, let us see once again the structure of our Datapath and recall the Datapath control signals.

Datapath Review

- **Structure** and **Control Signals** of our Datapath.
- Set **WR = 1** to write one of the four registers in the register file.
- **DA** selects the register to write to.
- **AA** and **BA** select the source registers for the ALU.
- **MB** chooses a register or a constant operand.
- **FS** selects an ALU operation.
- **MW = 1** to write to memory.
- **MD** selects between the ALU result and the RAM output.
- V, C, N and Z are status bits.
- **SL = 1** loads the status bits in the Status Register.



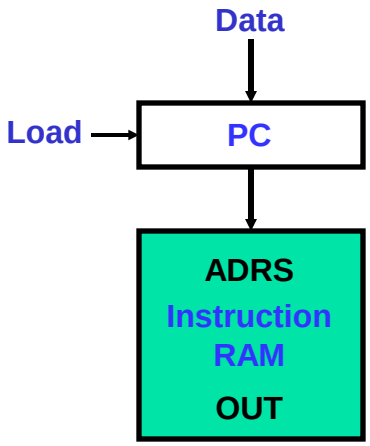
The Control Unit of Our Processor



- 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, SL**.
  - It generates the **"constant"** input for the datapath (not shown in the picture).
  - It executes program instructions in the correct sequence.
- The datapath also sends information back to the control unit.
  - Status V, C, N, Z can be inspected by branch instructions to alter a program's flow
  - Registers' content can be used to load Program Counter (not shown in the picture)

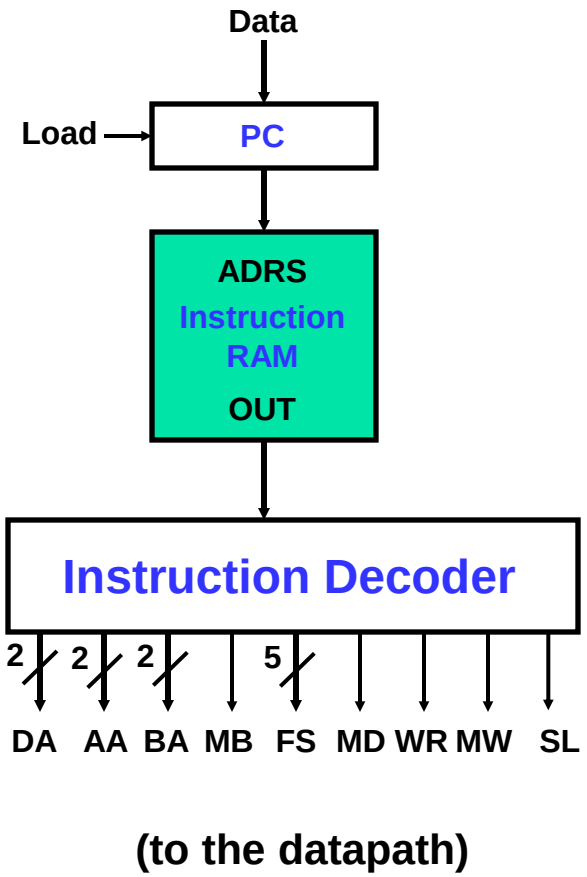
Program Counter

- 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, the PC is updated with Data, which represents some address specified in a jump or branch instruction.



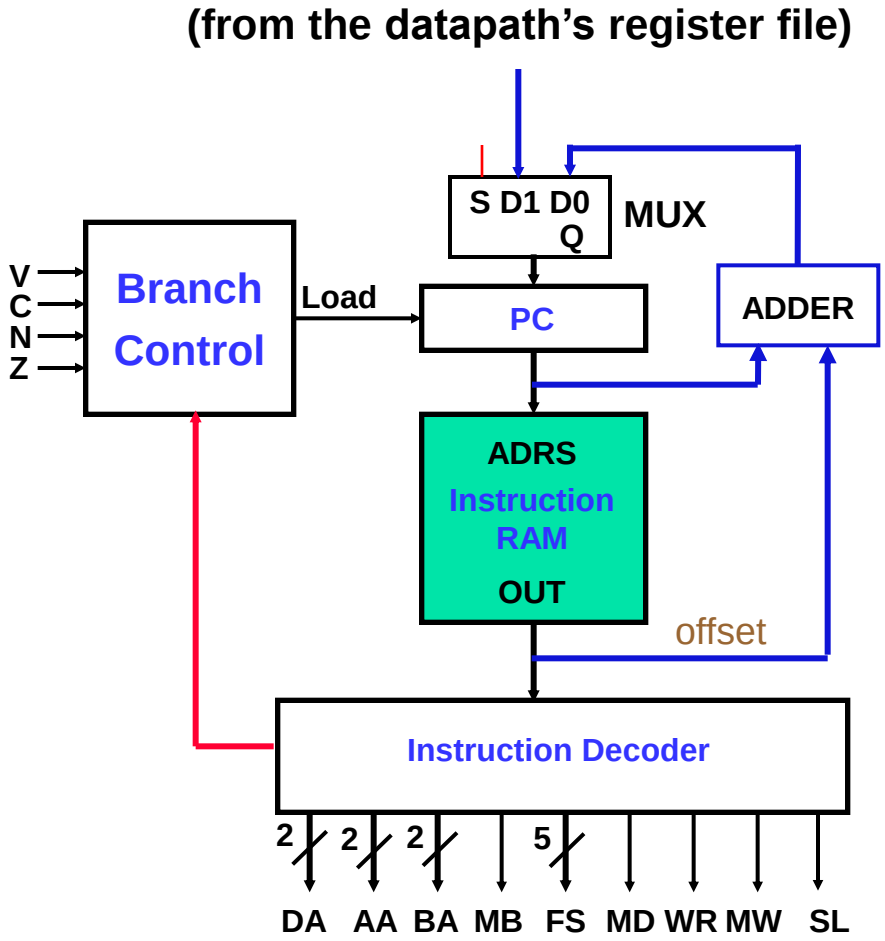
Instruction Decoder

- The instruction decoder is a combinational logics circuit
- It takes a machine language instruction
- It produces the matching control signals for the datapath
- These signals tell the datapath
  - which registers or memory locations to access
  - what ALU operations to perform



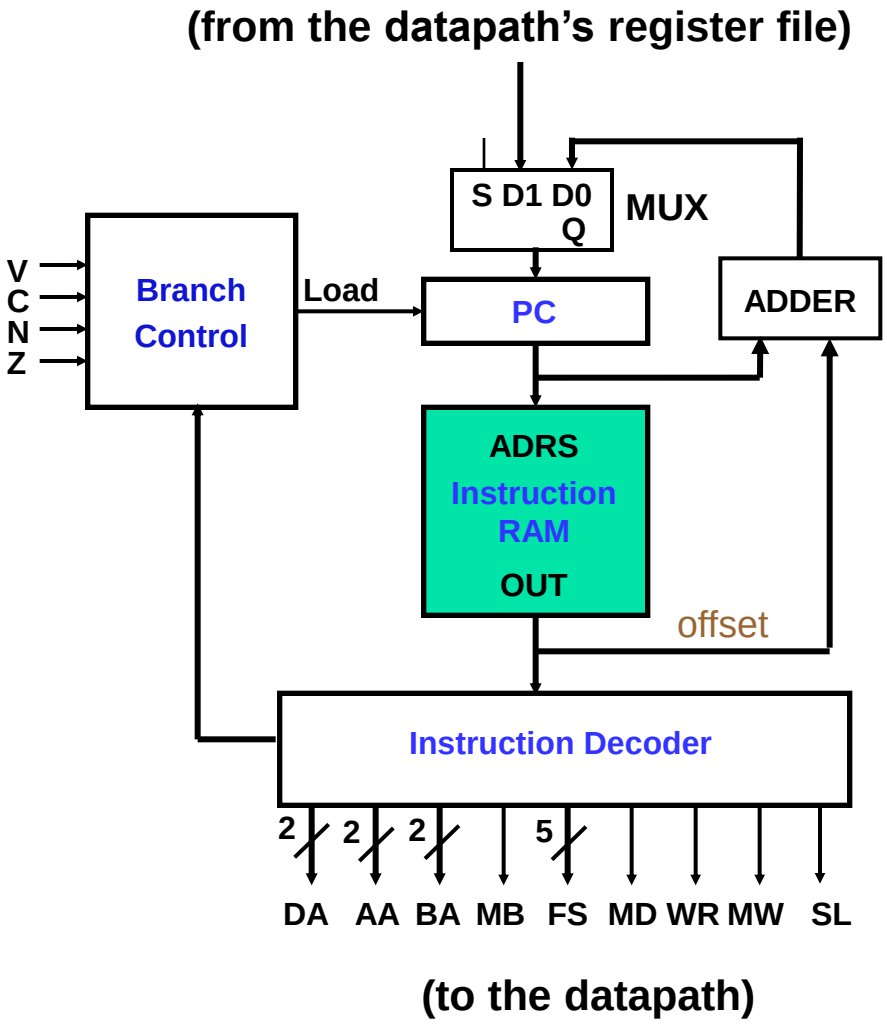
Branch Control Unit

- Finally, the branch control unit decides what the PC's next value should be.
  - For jump instruction
    - PC loaded with the target address specified in two registers of the register file
  - For branch instructions
    - PC loaded with the target address taken from the instruction only if the corresponding status bit is true
  - For all other instructions
    - PC is incremented, i.e., PC = PC + 1

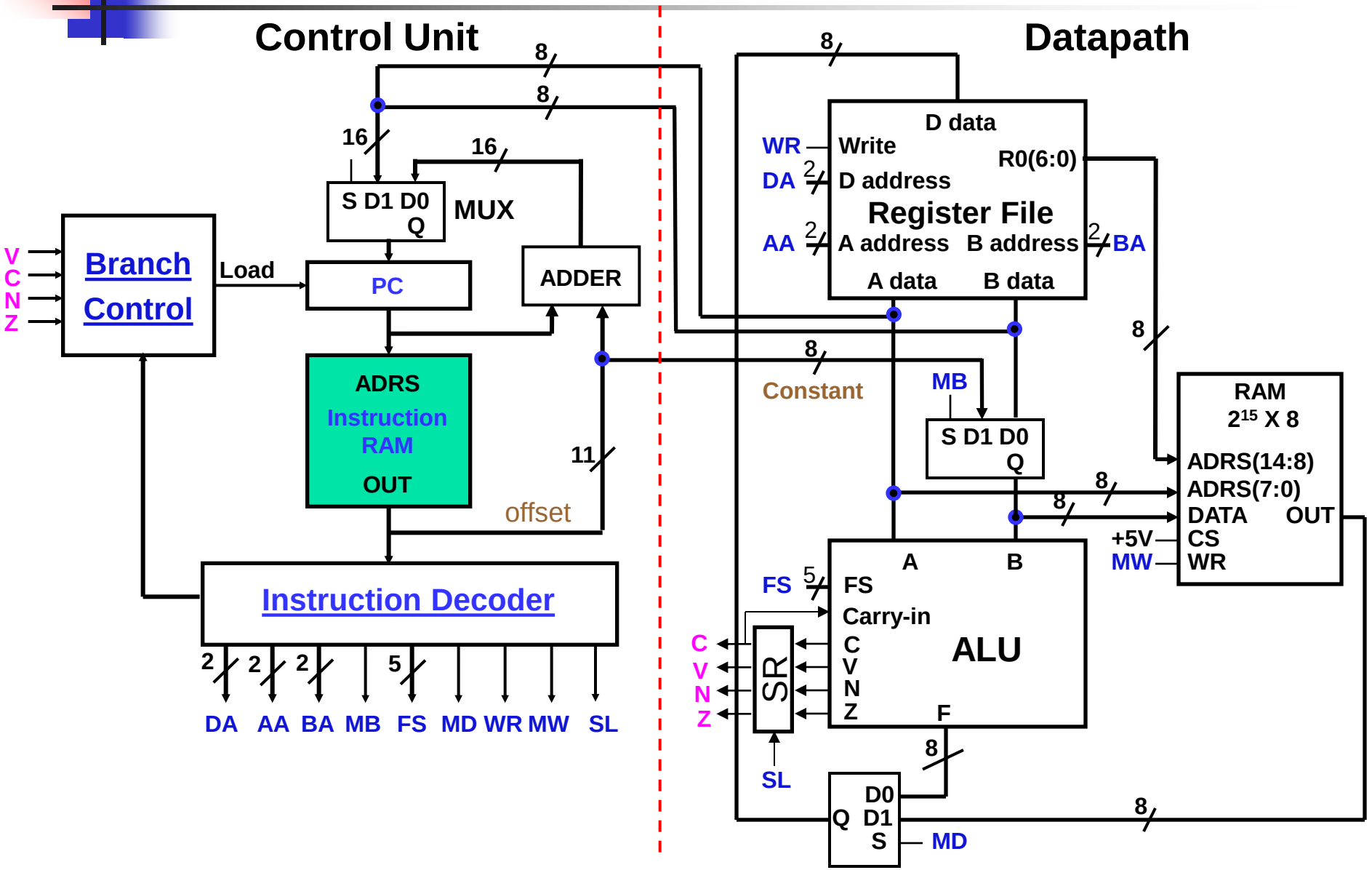


That's it!

- This is the basic control unit. On each clock cycle:
  1. An instruction is read from the instruction memory.
  2. The instruction decoder generates the matching datapath control word.
  3. Datapath registers are read and sent to the ALU or the data memory.
  4. ALU or RAM outputs are written back to the register file.
  5. The PC is incremented, or reloaded for branches and jumps.

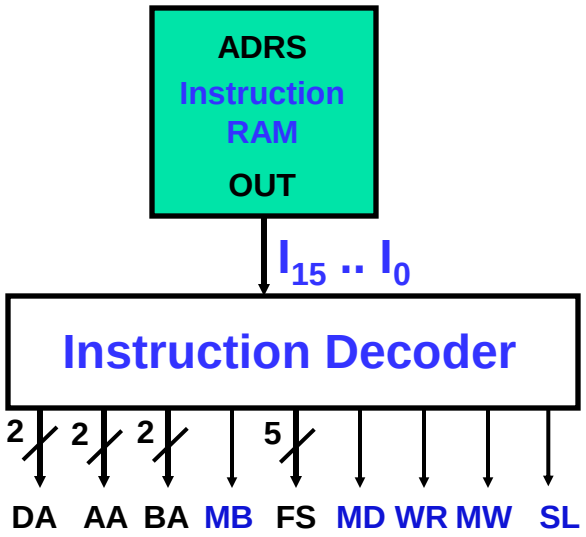


The Whole Processor



Implementing the Instruction Decoder

- It is a combinational circuit
- Its input is a 16-bit binary instruction (I)
  - comes from the instruction memory.
- Its output is a control word for the datapath. This includes:
  - WR, DA, AA, BA, and MD signals to control the register file.
  - FS for the ALU operation.
  - MW for the data memory write enable.
  - MB for selecting the second operand.
  - SL for loading the status register.
- We will see how these signals are generated for each of the three instruction formats.
- Let us start with the following signals:
  - MB, MD, WR, MW, and SL



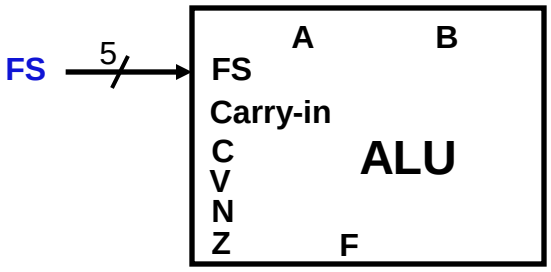
Generating MB, MD, WR, MW, and SL

Instr	Opcode bits					Control Signals				
	I_{15}	I_{14}	I_{13}	I_{12}	I_{11}	MB	MD	WR	MW	SL
INC	0	0	0	0	0	0	0	1	0	1
ADD	0	0	0	0	1	0	0	1	0	1
ADDC	0	0	0	1	0	0	0	1	0	1
SUB	0	0	0	1	1	0	0	1	0	1
DEC	0	0	1	0	0	0	0	1	0	1
LDR	0	0	1	0	1	0	0	1	0	1
SHR	0	0	1	1	0	0	0	1	0	1
SHL	0	0	1	1	1	0	0	1	0	1
AND	0	1	0	0	0	0	0	1	0	1
OR	0	1	0	0	1	0	0	1	0	1
XOR	0	1	0	1	0	0	0	1	0	1
NOT	0	1	0	1	1	0	0	1	0	1
ST	0	1	1	0	0	0	x	0	1	0
LD	0	1	1	0	1	x	1	1	0	0
STI	0	1	1	1	0	1	x	0	1	0
LDI	0	1	1	1	1	1	0	1	0	0
BNZ	1	0	0	0	0	x	x	0	0	0
BNC	1	0	0	0	1	x	x	0	0	0
BNV	1	0	0	1	0	x	x	0	0	0
BNN	1	0	0	1	1	x	x	0	0	0
BZ	1	0	1	0	0	x	x	0	0	0
BC	1	0	1	0	1	x	x	0	0	0
BV	1	0	1	1	0	x	x	0	0	0
BN	1	0	1	1	1	x	x	0	0	0
	x	x	x	x	x	x	x	x	x	x
JMP	1	1	1	1	1	x	x	0	0	0

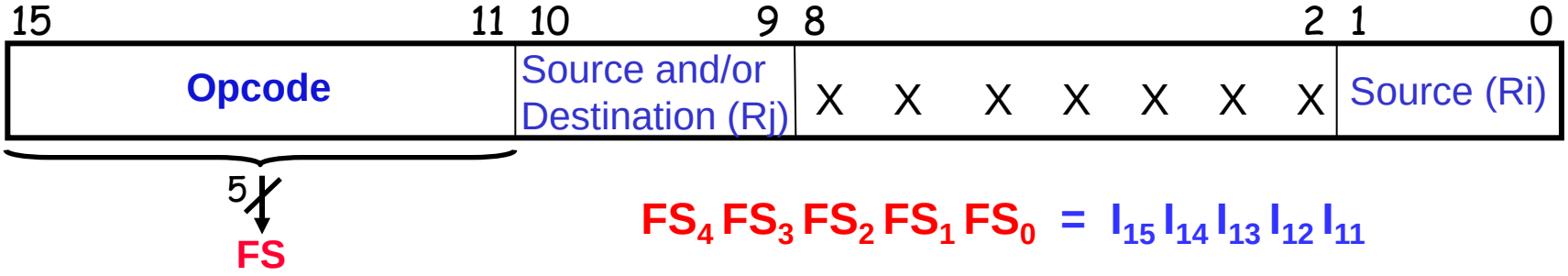
- The table shows the correct control signals MB, MD, WR, MW, and SL for each instruction.
- There are several patterns visible in this table.
  - MW = 1 only for memory write operations.
  - MB = 1 only for immediate instructions, which require a constant.
  - MD is unused when WR = 0.
  - Jump and branches modify neither registers nor main memory.
- From the table we can derive the Boolean equations for the signals:
  - $MB = I_{15}' I_{14} I_{13} I_{12}$
  - $MD = I_{15}' I_{14} I_{13} I_{12}'$
  - $WR = I_{15}' (I_{14}' + I_{13}' + I_{11})$
  - $MW = I_{15}' I_{14} I_{13} I_{11}'$
  - $SL = I_{15}' (I_{14}' + I_{13}')$

Generating FS

- The ALU function selection code (FS) is the same as the operation code (Opcode) for arithmetic, logic, and shift instructions (see previous slides).
- Thus, the control unit can "generate" the ALU's FS control signal just by taking it directly out of the instruction Opcode.
- For register-format ALU instructions:

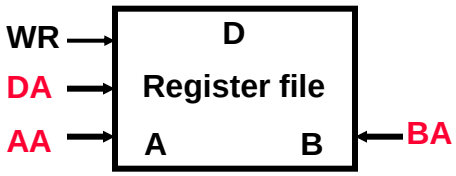


INSTR	Opcode	FS
INC	00000	00000
ADD	00001	00001
ADDC	00010	00010
SUB	00011	00011
DEC	00100	00100
LDR	00101	00101
SHR	00110	00110
SHL	00111	00111
AND	01000	01000
OR	01001	01001
XOR	01010	01010
NOT	01011	01011

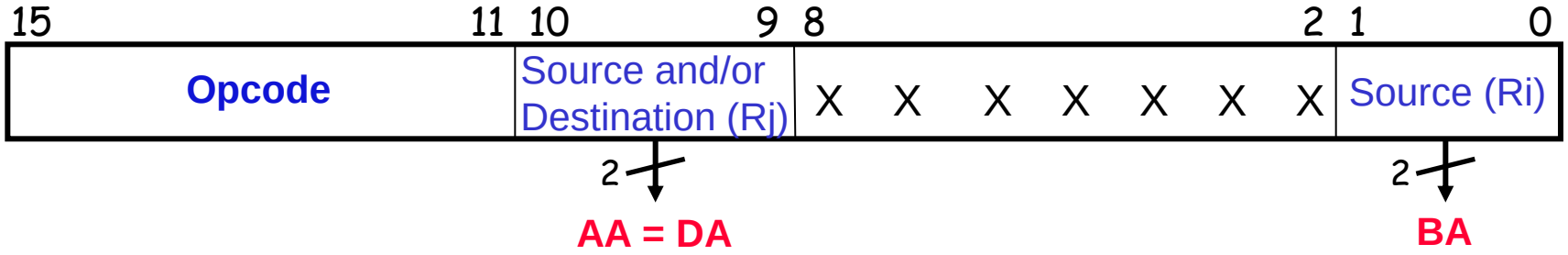


$FS_4\ FS_3\ FS_2\ FS_1\ FS_0 = I_{15}\ I_{14}\ I_{13}\ I_{12}\ I_{11}$

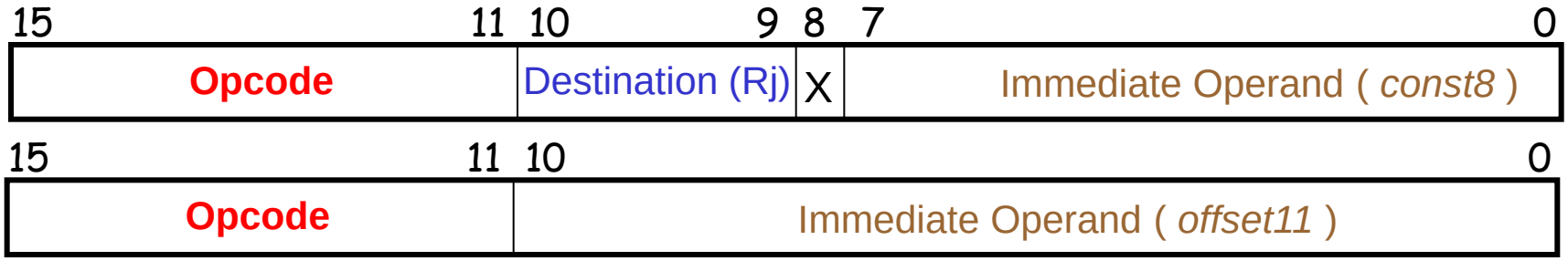
Generating DA, AA, BA



- The register file addresses DA, AA and BA can be taken directly out of the 16-bit binary instructions.
  - Instruction bits 10-9 are the destination register, DA.
  - Bits 10-9 are fed directly to AA, the first register file source.
  - Bits 1-0 are connected directly to BA, the second source.
- This clearly works for a register-format instruction where bits 10-9 and 1-0 were defined to hold the destination and source registers.
- Notice, that the source register A is also the destination register, thus AA = DA.



Don’t-care Conditions

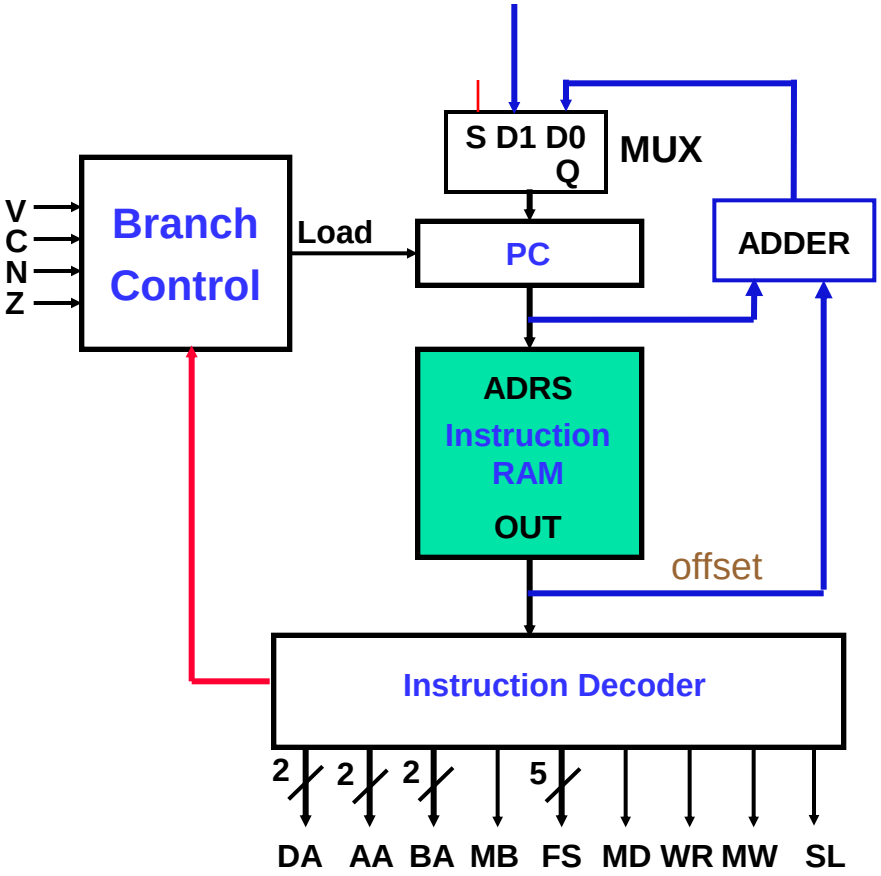


- In immediate-format instructions, bits 7-0 store a constant operand, not a second source register!
  - However, immediate instructions only use one register, so the control signal BA would be a don’t care condition anyway.
- Similarly, branch instructions require neither a destination register nor a second source register.
- So, we can always take DA, AA and BA directly from the instruction.

$$DA_1\, DA_0 = I_{10}\, I_9$$
$$AA_1\, AA_0 = I_{10} I_9$$
$$BA_1\, BA_0 = I_1\, I_0$$

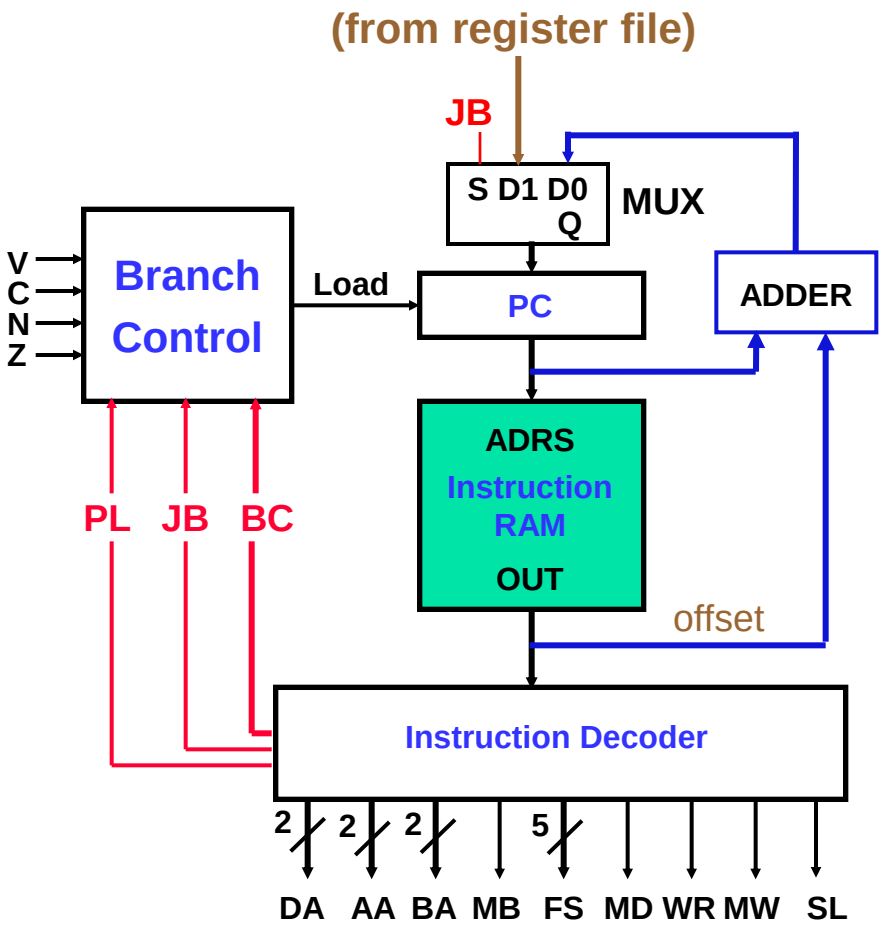
More About the Branch Control Unit

- The branch control unit needs a lot of information about the current instruction.
  - Whether it is a jump, a branch, or some other instruction.
  - For branches, the specific branch condition.
- All of this can be generated by the instruction decoder, which has to process the instruction words anyway.



Branch Control Unit Inputs and Outputs

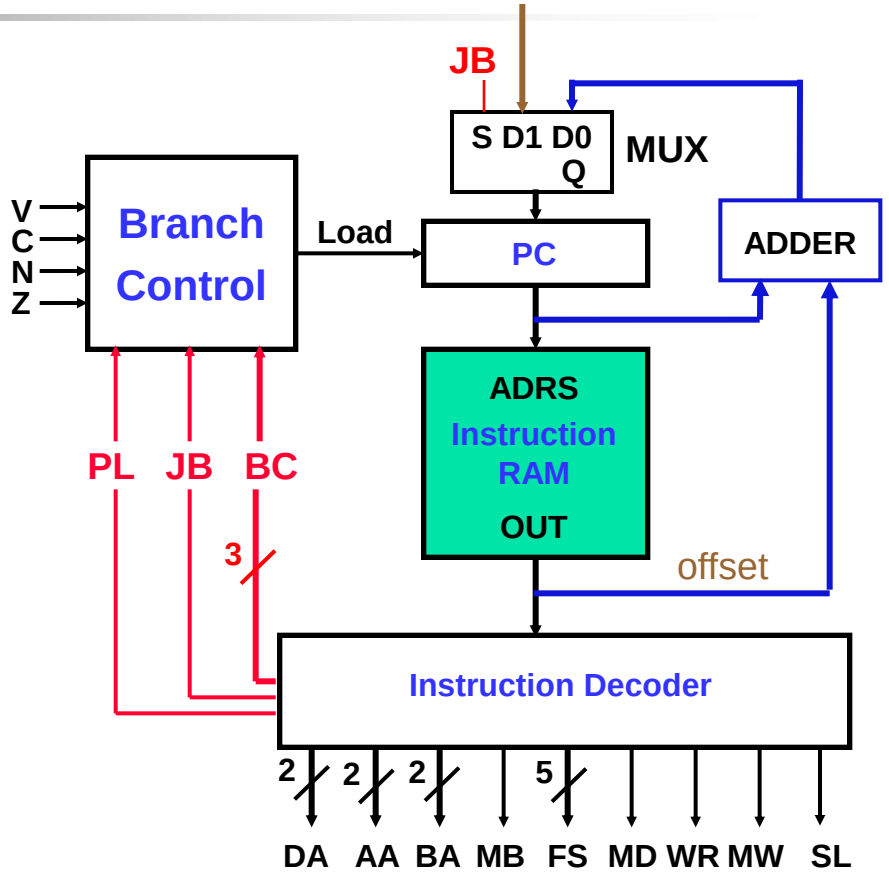
- Branch control inputs:
  - PL = 1 – **P**C **L**oad may be needed (jump or branch)
  - JB = 1 – **J**ump instruction
  - BC – **B**rach **C**ondition
  - Status bits V, C, N and Z come from the Datapath.
- Branch control outputs:
  - A Load signal for the PC.
  - When Load = 1,
    - If JB = 0 then the target address to branch is loaded from the instruction memory.
    - If JB = 1 then the target address to jump is loaded from the register file.



Branch Control Unit Inputs

- The decoder sends the following data to the branch control unit:
  - PL and JB indicate the type of instruction.
  - BC encodes the kind of branch.

BC	Condition
000	Branch if non-zero
001	Branch if carry clear
010	Branch if no overflow
011	Branch if positive
100	Branch if zero
101	Branch if carry set
110	Branch if overflow
111	Branch if negative



PL	JB	Instruction
0	x	Other
1	0	Branch
1	1	Jump

Generating PL and JB

Instr	Opcode bits					Signals	
	I_{15}	I_{14}	I_{13}	I_{12}	I_{11}	PL	JB
INC	0	0	0	0	0	0	x
ADD	0	0	0	0	1	0	x
ADDC	0	0	0	1	0	0	x
SUB	0	0	0	1	1	0	x
DEC	0	0	1	0	0	0	x
LDR	0	0	1	0	1	0	x
SHR	0	0	1	1	0	0	x
SHL	0	0	1	1	1	0	x
AND	0	1	0	0	0	0	x
OR	0	1	0	0	1	0	x
XOR	0	1	0	1	0	0	x
NOT	0	1	0	1	1	0	x
ST	0	1	1	0	0	0	x
LD	0	1	1	0	1	0	x
STI	0	1	1	1	0	0	x
LDI	0	1	1	1	1	0	x
BNZ	1	0	0	0	0	1	0
BNC	1	0	0	0	1	1	0
BNV	1	0	0	1	0	1	0
BNN	1	0	0	1	1	1	0
BZ	1	0	1	0	0	1	0
BC	1	0	1	0	1	1	0
BV	1	0	1	1	0	1	0
BN	1	0	1	1	1	1	0
	x	x	x	x	x	x	x
JMP	1	1	1	1	1	1	1

- The instruction decoder generates PL and JB from instruction Opcodes.
  - Note that if PL = 0, then the value of JB does not matter.
  - As expected, PL and JB only matter for jumps and branches.
- From the table on the left we can derive the Boolean functions for the signals:
  - $PL = I_{15}$
  - $JB = I_{14}$

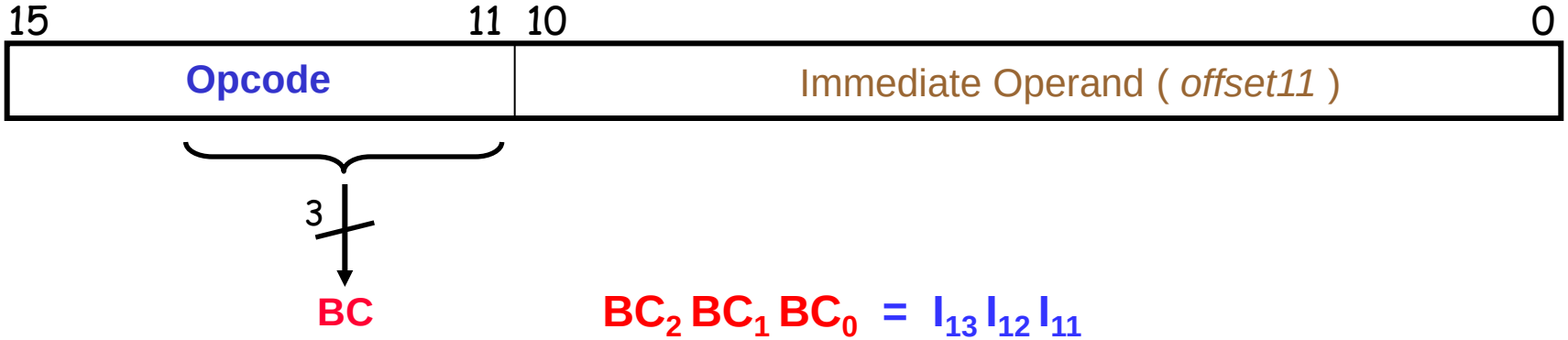
PL	JB	Instruction
0	x	Other
1	0	Branch
1	1	Jump

Generating BC

- We defined the branch opcodes so that they already contain the branch type, so BC can come straight from the instruction Opcode.

INSTR	Opcode	Operation
BNZ	10000	Branch if non-zero
BNC	10001	Branch if carry clear
BNV	10010	Branch if no overflow
BNN	10011	Branch if positive

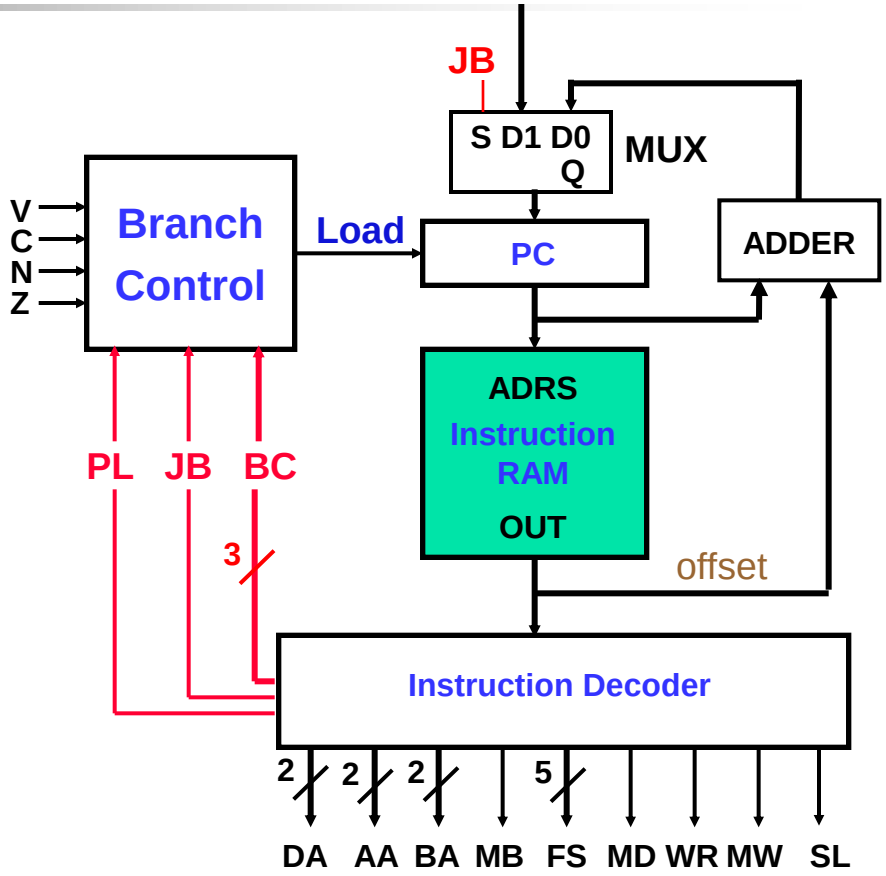
INSTR	Opcode	Operation
BZ	10100	Branch if zero
BC	10101	Branch if carry set
BV	10110	Branch if overflow
BN	10111	Branch if negative



$BC_2\ BC_1\ BC_0 = I_{13}\ I_{12}\ I_{11}$

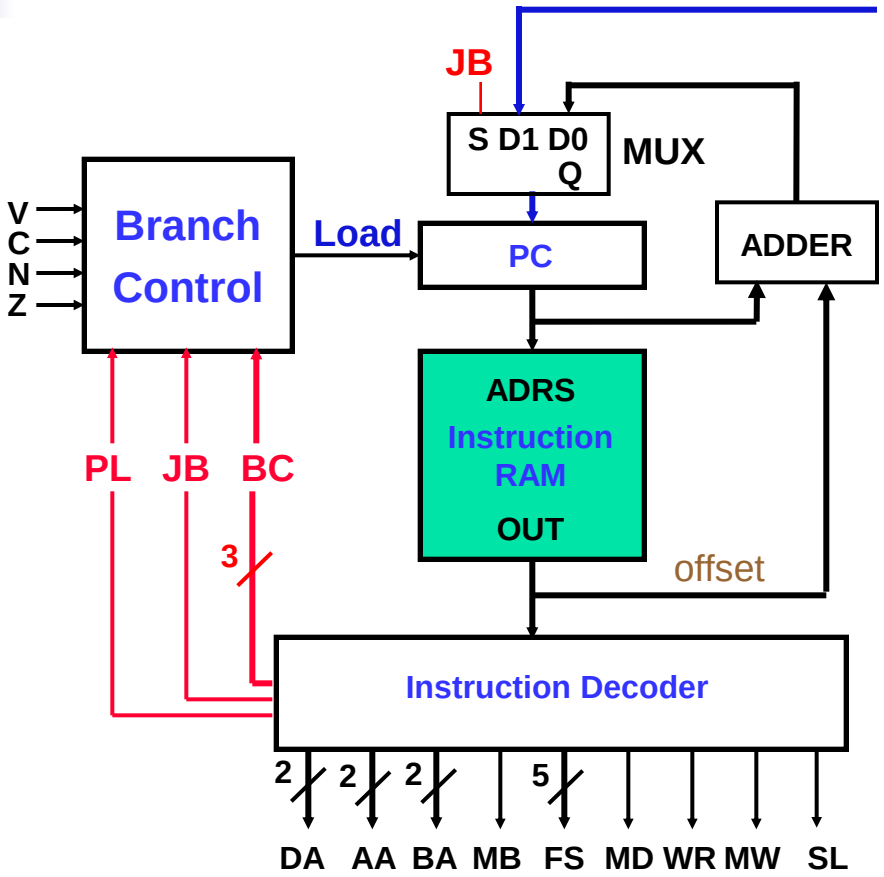
Branch Control Unit and PC

- We have seen how the instruction decoder generates PL, JB, and BC. How does the branch unit use these to control the PC?
- There are three cases, depending on the values of PL and JB.
- If PL = 0, the current instruction is ***not*** a jump or branch
- So the branch control just needs to make the program counter increment, and execute the next instruction, i.e., Load = 0.



PL	JB	Instruction
0	**x**	**Other**
1	0	Branch
1	1	Jump

Jumps

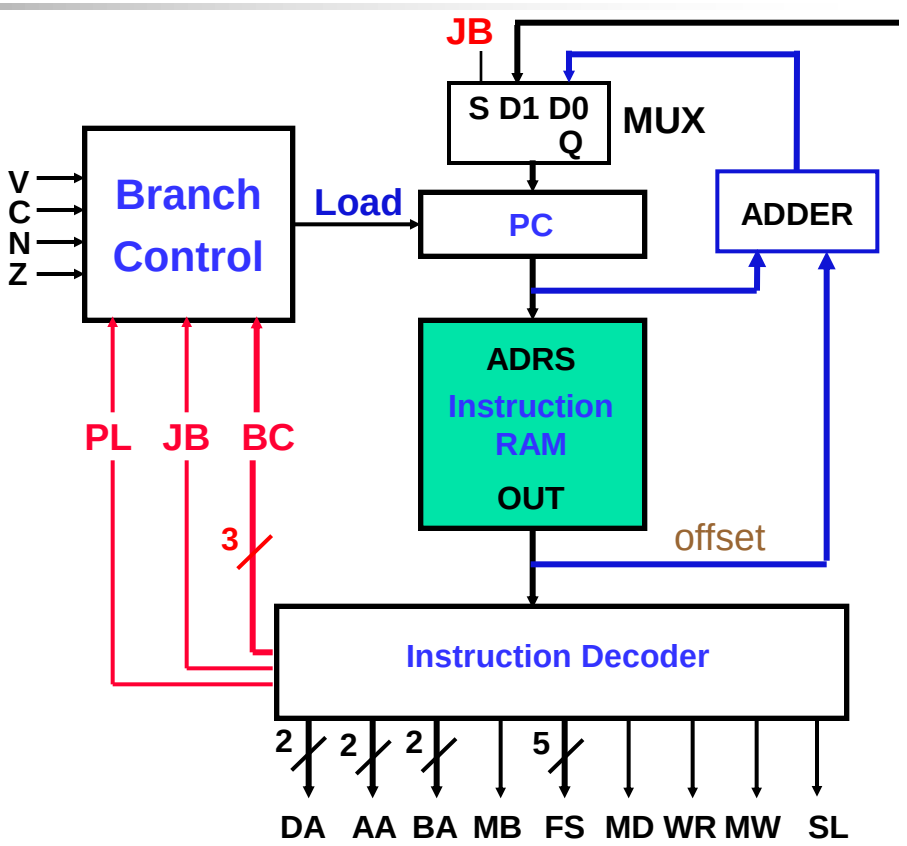


PL	JB	Instruction
0	x	Other
1	0	Branch
1	**1**	**Jump**

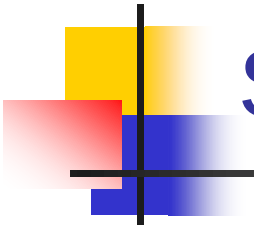
- If PL = 1 and JB = 1, the current instruction must be a jump.
- In our processor the jump address is taken from the register file.
  - JB = 1 will switch the multiplexer **MUX** to connect the PC with the register file.
  - The Branch Control unit sets Load = 1 to allow the loading of the PC.

Branches

- If PL = 1 and JB = 0, the current instruction is a conditional branch.
- The output of ADDER is connected to the PC by MUX.
- 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 = 100 (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 should be taken, then Load = 1 making **PC = PC + offset**
  - Otherwise, Load = 0 and the PC is incremented, just as for normal instructions.
  - Recall that **offset** is a signed number, so we can branch forwards or backwards.



PL	JB	Instruction
0	x	Other
1	**0**	**Branch**
1	1	Jump



Summary

- Today we have designed the control unit hardware.
  - The **program counter** points into a special instruction memory, which contains a machine language program.
  - An **instruction decoder** looks at each instruction and generates the correct control signals for the datapath and the branching unit.
  - The **branch control unit** handles instruction sequencing.
- The control unit implementation depends on both the Instruction Set Architecture and the Datapath.
  - Careful selection of opcodes and instruction formats can make the control unit simpler.
- We now have the whole processor! This is the culmination of everything we did in the FDSD course, starting from those tiny little primitive gates.