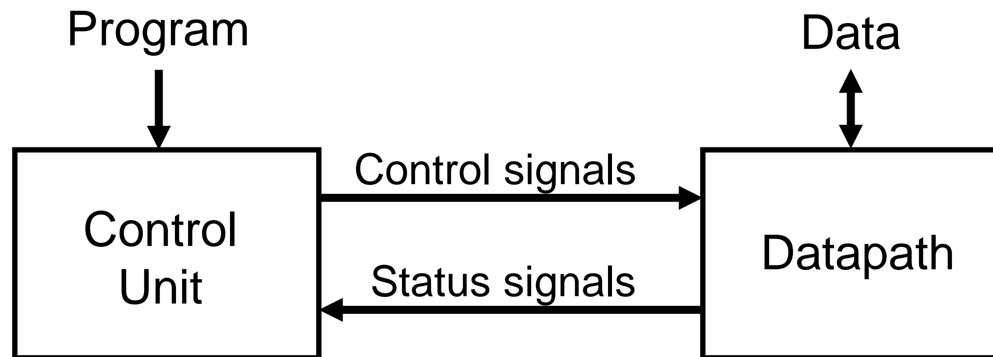
Processor Design Basics: Instruction Set Architecture

Overview

- Block Diagram of a Generic Processor
  - Von Neumann vs. Harvard Architecture
  - Example of a Simple 8-bit Processor
- Introduction to Instruction Set Architecture
  - Programming Model
  - Instruction Specifications
    - Data Manipulation Instructions
    - Data Movement Instructions
    - Control Flow Instructions
  - Instruction Formats
- How to Program Processors
  - High-level Languages
  - Low-level Languages (Assembly and Machine Languages)
  - Compiling
- Summary

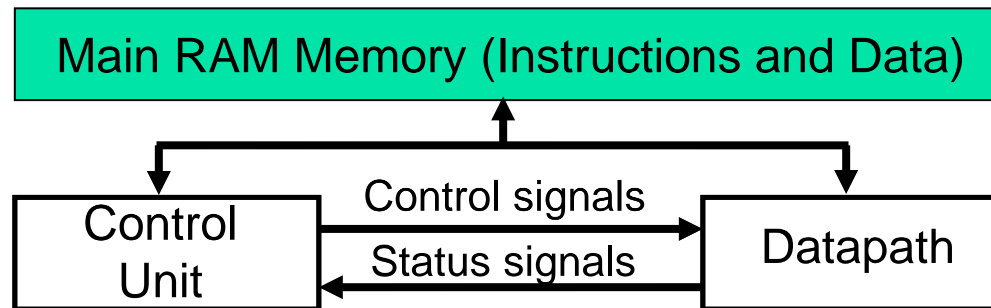
Block Diagram of a Generic Processor



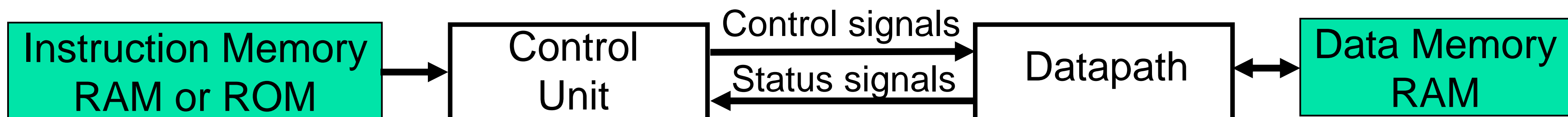
- We can divide the design of a processor into three parts:
  - An Instruction Set is the programmer's interface to the processor.
  - The Datapath does all of the actual data processing.
  - A Control unit uses the programmer's instructions to tell the datapath what to do.
- But first, Where Do The Program and Data Go?

Von Neumann vs. ***Harvard*** Architecture

- Von Neumann Architecture
  - A ***single main memory*** that holds both program instructions and data.

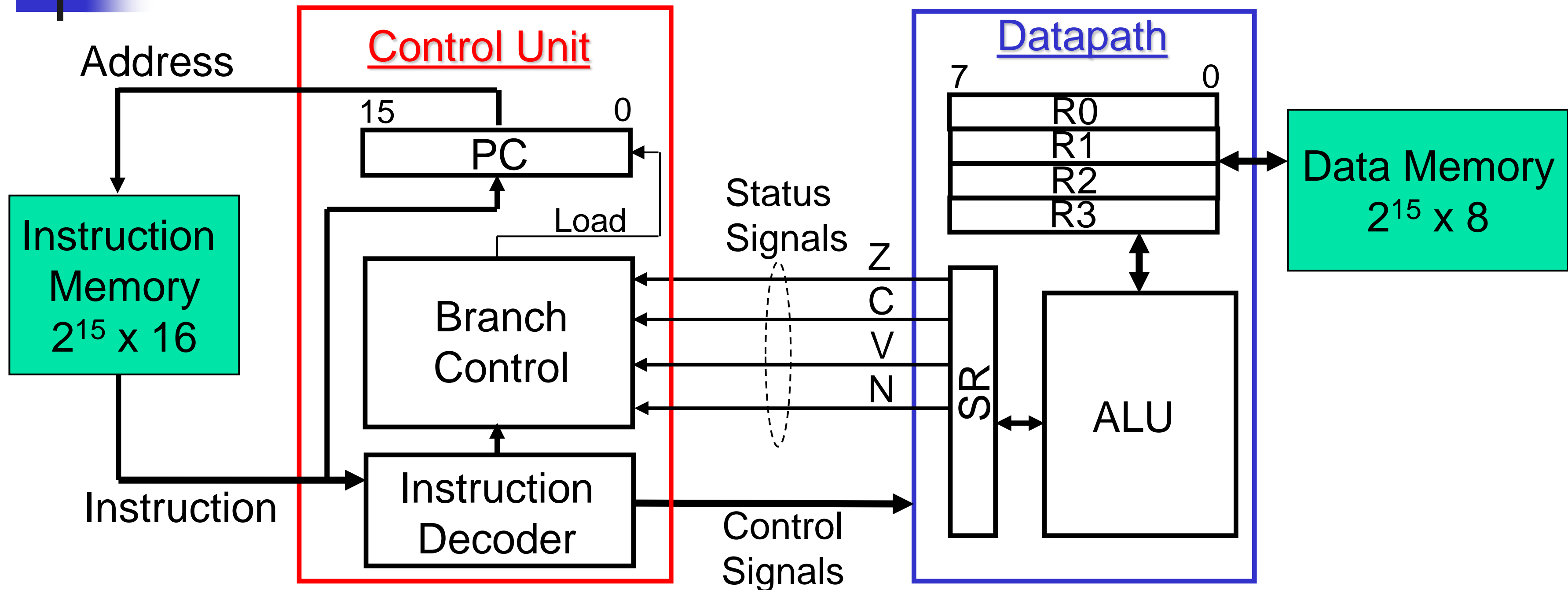


- Harvard Architecture
  - It includes two memory units: instruction and data memory.
    - The instruction memory holds the program instructions.
    - The separate data memory is used for computations.
  - The advantage is that we can read an instruction (from the instruction memory) *and* load or store data (from/to the data memory) in the same clock cycle.



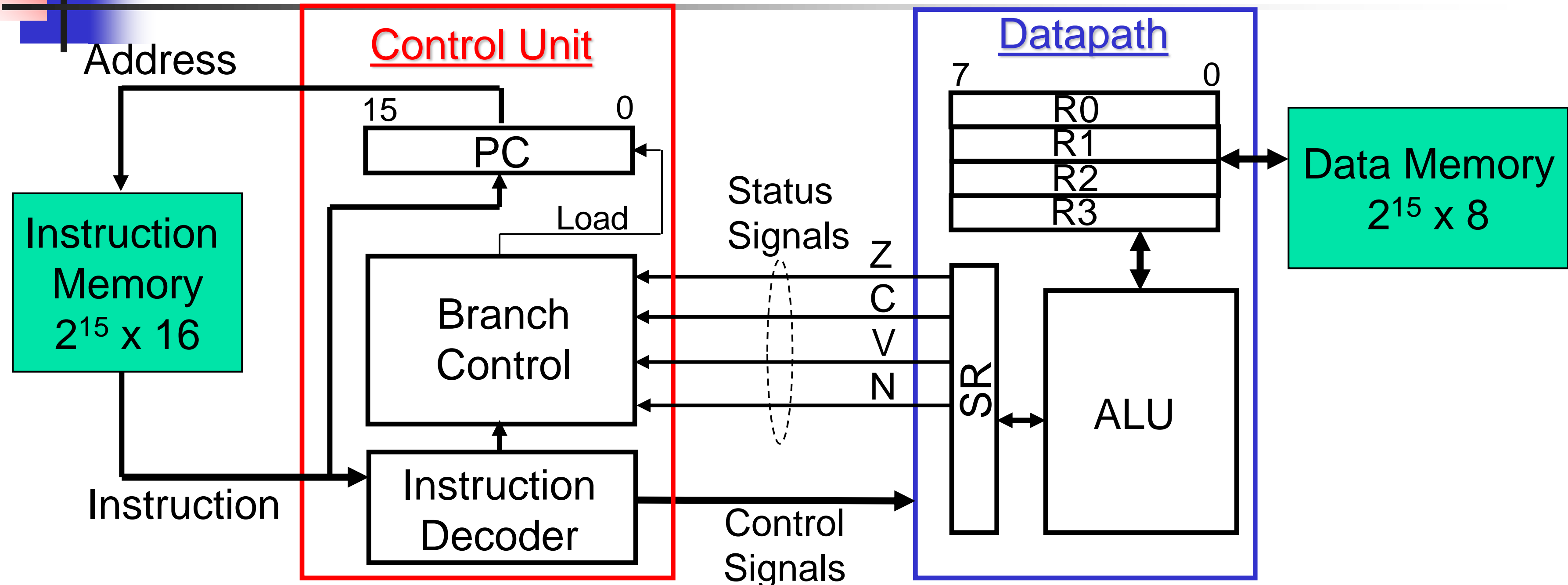
- Further, we will consider a processor with Harvard architecture.

Example of a Simple 8-bit Processor



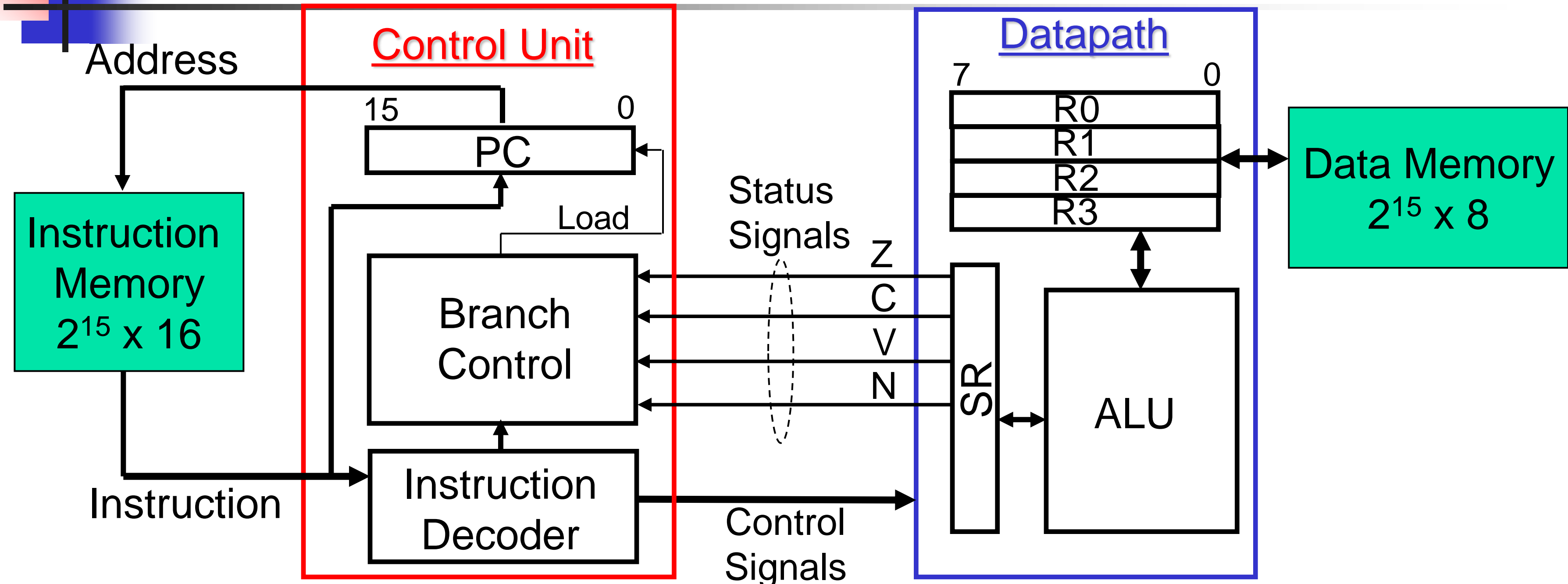
- The processor has a Harvard Architecture, i.e., separate Instruction and Data memory.
- This is an 8-bit processor because all operations performed by the Arithmetic and Logic Unit (ALU) are on 8-bit operands.
- The operands are stored in the Register File that consists of four 8-bit general purpose registers (R0 to R3).

Example of a Simple 8-bit Processor (cont.)



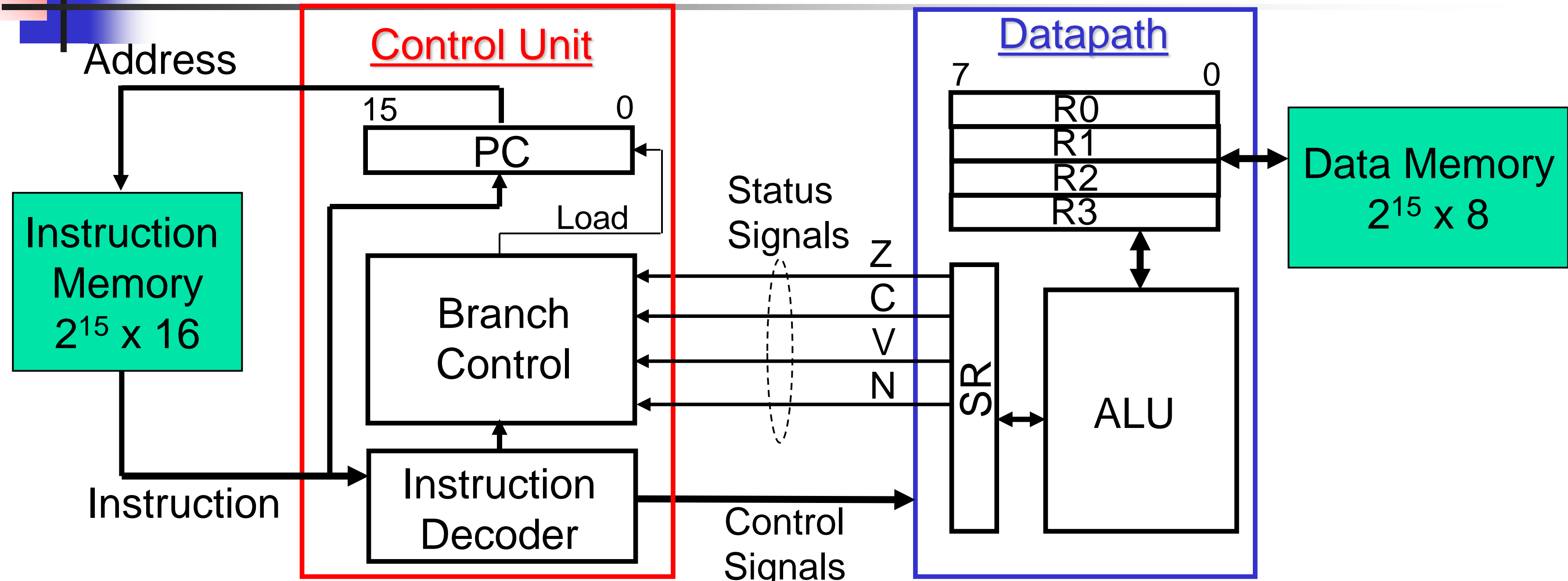
- The register file is very small but very fast temporary storage needed by the ALU for operands and intermediate results.
- A larger but slower storage is the Data Memory. The processor can move data from the register file to the data memory and back.
- When operations like "addition", "subtraction", etc. are performed by the ALU the following may happen:
  - Carry or Overflow situation may occur.
  - The result may be Zero or Negative.
- Thus, the ALU generates status bits stored in the Status Register (SR) to indicate if carry, overflow, zero, or negative situation has occurred.

Example of a Simple 8-bit Processor (cont.)



- The Control Unit fetches instructions from the Instruction Memory to determine what operation needs to be executed.
- A register called Program Counter keeps the address of the instruction to be executed.
- After an instruction is fetched from the Instruction Memory it is decoded by the Instruction Decoder and control signals are sent to instruct the Datapath what operation to perform.
- Next, the Program Counter is loaded with the address of the next instruction to be executed.

Example of a Simple 8-bit Processor (cont.)



- There are two possibilities to load the PC:
  - Simply the PC is incremented, i.e., PC = PC + 1 (Hence the name “Program Counter”).
  - The PC is loaded with a number taken from the instruction or some of the registers in the register file. In this case we say that the processor makes a branch/jump in the program.
- The Branch Control unit determines whether the PC should be incremented or loaded to make a branch/jump.
  - The decision depends on the current instruction (whether it is a branch/jump instruction) and the value of the status bits.

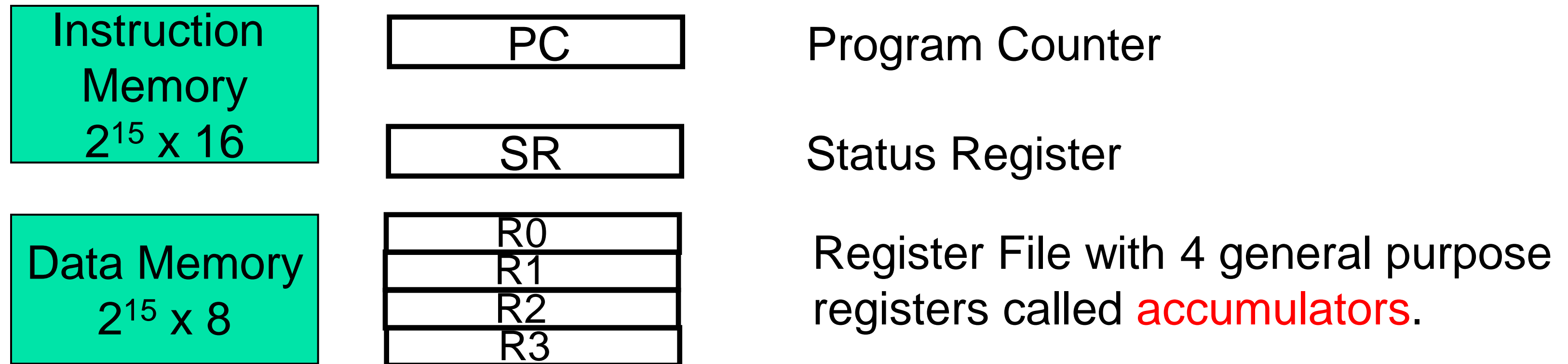
Some Basic Terminology

- The first step in a processor design is to specify the operations that the processor should perform, i.e.,
  - to specify the instruction set of the processor.
- An instruction is a collection of bits that instructs the processor to perform a specific operation.
- An instruction set is the collection of all instructions for a processor.
- A program is a list of instructions that specifies
  - the operations to be performed by the processor
  - their sequence
- A thorough description of the instruction set for a processor is called instruction set architecture (ISA).

Instruction Set Architecture (ISA)

- Any instruction set architecture has the following three major components
- Programming Model:
  - This is the processor structure as viewed by a user programming the processor in a language that directly specifies the instructions to be executed.
  - Such language is called assembly language (more details later!)
- Instruction Specifications:
  - They describe each of the distinct instructions that can be executed by a processor.
- Instruction Formats:
  - They determine the meaning of the bits used to encode each instruction.

Programming Model for our Processor



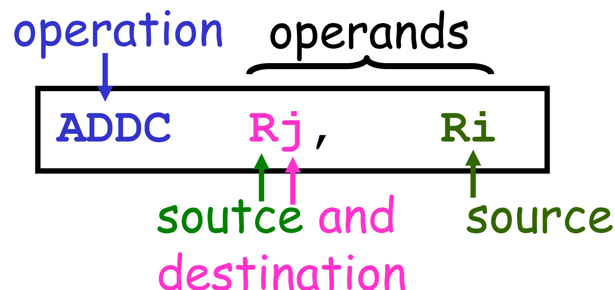
- It gives the resources of our processor that the user sees available for storing information.
  - The model has two memories,
    - one for storage of instructions (with size 32K 16-bit words)
    - one for storage of data (with size 32KB).
  - A register file with four 8-bit Registers (R0 to R3)
- Also, visible to the programmer are
  - the 16-bit Program Counter (PC)
  - the Status Register (SR) of our processor.

Instruction Specifications for our 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|Rj] ← 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	

Data Manipulation Instructions

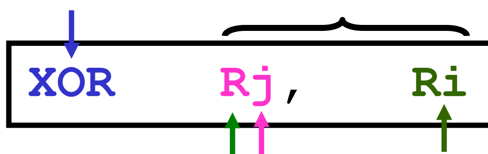
- Data manipulation instructions correspond to Arithmetic, Logic, and Shift operations.
- Consider instruction **ADDC** of our processor:
  - It corresponds to **arithmetic** operation *"addition with carry-in"*.
  - It has **two** operands.
  - It modifies the **C**arry, o**V**erflow, **Z**ero, and **N**egative bits in the processor's Status Register.



Register transfer notation:

Rj ← Rj + Ri + C

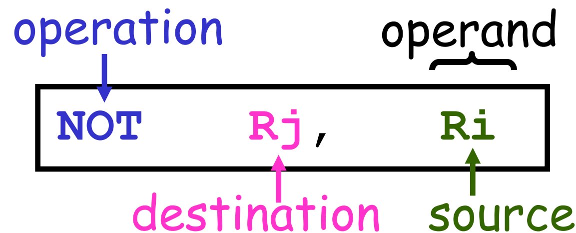
- Consider instruction **XOR** of our processor:
  - It corresponds to **logic** operation *"bitwise XOR"*.
  - It has **two** operands.
  - It modifies bits **Z** and **N** in the Status Register.



Rj ← Rj ⊕ Ri

Data Manipulation Instructions (cont.)

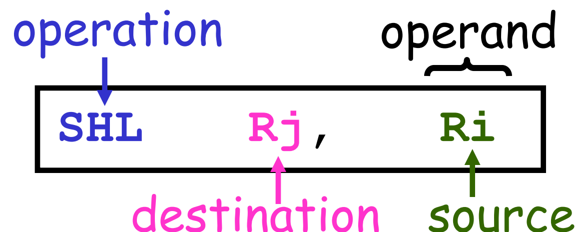
- Consider instruction **NOT** of our processor:
  - It corresponds to **<u>logic</u>** operation *"complement"*.
  - It has **<u>one</u>** operand.
  - It modifies bits **Z** and **N** in the Status Register.



Register transfer notation:

`Rj ← Ri'`

- Consider instruction **SHL** of our processor:
  - It corresponds to **<u>shift</u>** operation *"shift-left with one bit and fill with 0"*.
  - It has **<u>one</u>** operand.
  - It does NOT modify bits in the Status Register.

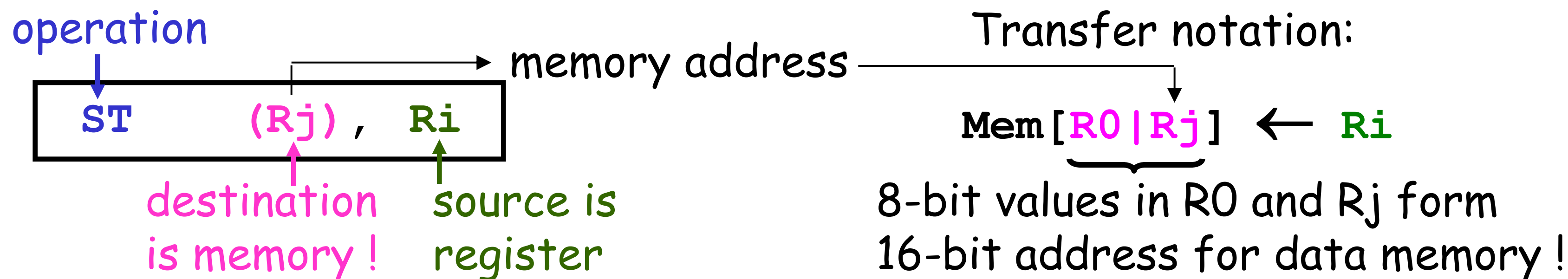


Register transfer notation:

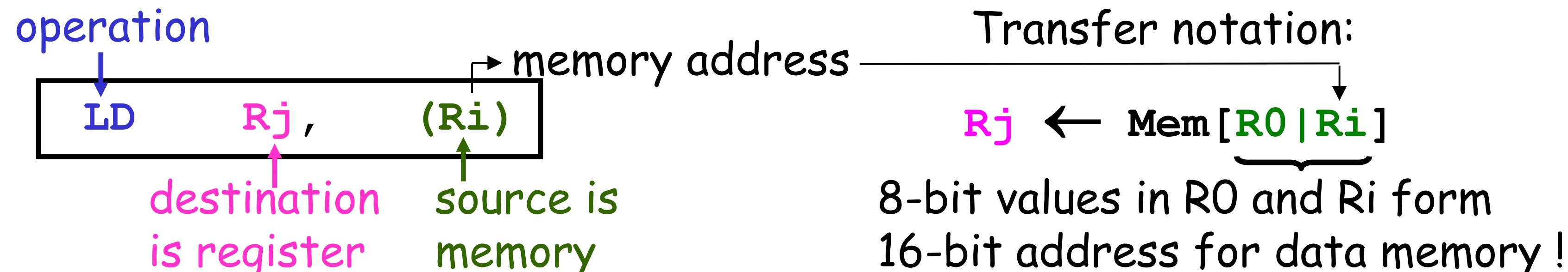
`Rj ← Ri << 1`

Data Movement Instructions

- Data movement instructions correspond to transfer operations of data between the Data Memory and the Register File. Also, operations that load/store a constant in a register/memory cell are considered here.
- Consider instruction **ST** of our processor:
  - It copies (stores) data from a register to a memory cell.
  - It does NOT modify bits in the Status Register.

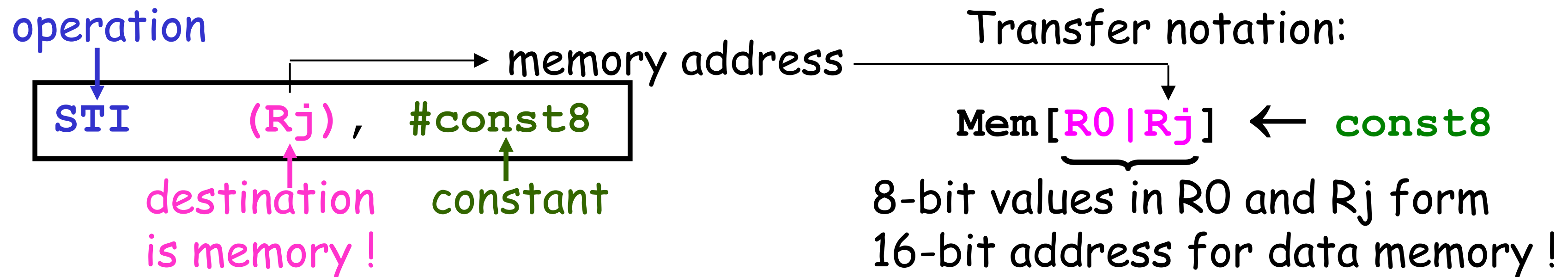


- Consider instruction **LD** of our processor:
  - It copies (loads) data to a register from a memory cell.
  - It does NOT modify bits in the Status Register.

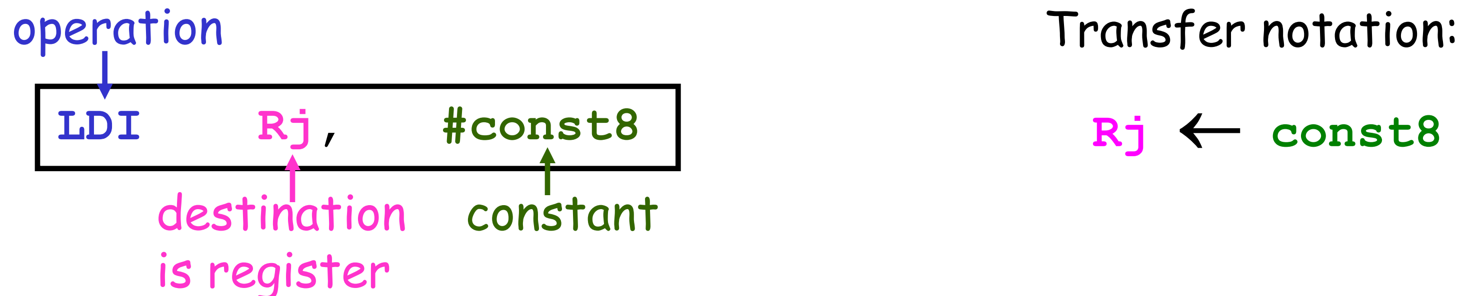


Data Movement Instructions (cont.)

- Consider instruction **STI** of our processor:
  - It stores an 8-bit constant to a memory cell.
  - It does NOT modify bits in the Status Register.



- Consider instruction **LDI** of our processor:
  - It loads a register with an 8-bit constant.
  - It does NOT modify bits in the Status Register.



Addressing Modes

- We have seen several instructions containing the # or ( ) symbols.
- The # and ( ) are important!
- These symbols are ways of specifying different addressing modes.
- For our simple processor we use three addressing modes: direct mode, index mode () and immediate mode #.
  - Direct mode - when we address a register (no specific symbol is used!)

```
LDR R3, R1          // R3 ← R1
SUB R2, R0          // R2 ← R2 + R0' + 1
```

  - Index mode - when we address a memory cell by using the content of registers as address. Here we use the symbol **( )** !

```
ST  (R3) , R1       // Mem[R0|R3] ← R1
LD  R3  ,(R1)       // R3 ← Mem[R0|R1]
```

  - Immediate mode - when we refer to a constant we use the symbol **#** !

```
LDI  R2 , #35       // R2 ← 35
STI (R3), #10       // Mem[R0|R3] ← 10
```

Control Flow Instructions

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

```
...
Addr 0x0010:  LDI R0, #0x25   // R0 ← 0x25
Addr 0x0011:  LDI R1, #0xfc   // R1 ← 0xfc
Addr 0x0012:  LD  R3, (R1)    // R3 ← Mem[0x25fc]
Addr 0x0013:  INC R3, R3      // R3 ← R3 + 1
Addr 0x0014:  ST  (R1),R3      // Mem[0x25fc] ← 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.

- Control Flow Instructions change the value stored in the Program Counter.
- Thus, the programmer can use them when it is necessary to change the sequence of executions of instructions in the program.
- The change of the instruction sequence can be unconditional or conditional.
- An unconditional change is accomplished by using **Jump** instructions.
- A conditional change is accomplished by using **Branch** instructions.

Jump Instructions

- A jump instruction ***always*** changes the value of the PC.
  - The operand specifies exactly how to change the PC.
- For example, a program can skip certain instructions.

```
0x0010:  LDI R1, #0x00    // R1 ← 0x00
0x0011:  LDI R2, #0x15    // R2 ← 0x15
0x0012:  JMP R1, R2       // PC ← 0x0015 (go to address 0x0015)
0x0013:  LDI R1, #0x20    // These two instructions
0x0014:  LDI R2, #0x04     // would be skipped
0x0015:  SUB R3, R0       // R3 ← R3 + R0′ + 1
0x0016:  ADD R3, R3       // R3 ← R3 + R3
```

- You can also use jumps to repeat instructions.

```
0x0010:  LDI R1, #0x00   // R1 ← 0x00
0x0011:  LDI R2, #0x12   // R2 ← 0x12
0x0012:  ADD R0, R3      // R0 ← R0 + R3
0x0013:  JMP R1, R2      // PC ← 0x0012 (go to address 0x0012)
```

Branch Instructions

- A branch instruction ***may*** change the PC, depending on whether a given condition is true.
- Branch conditions are often based on the result of Arithmetic and Logic operations.
- This is what the Status Register bits C, V, Z and N are used for. With them we can implement various branch instructions like the ones below for our simple processor.

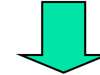
Instruction Mnemonic	Condition	Status bit
BV	Branch on overflow	V = 1
BNV	Branch on no overflow	V = 0
BC	Branch if carry set	C = 1
BNC	Branch if carry clear	C = 0
BN	Branch if negative	N = 1
BNN	Branch if positive	N = 0
BZ	Branch if zero	Z = 1
BNZ	Branch if non-zero	Z = 0

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

Branch Instructions Example

- Let us write the program on the right for our simple processor.
- Below, you see that we need many instructions for this simple program but that is how processors work.

if Mem[0x5a38] ≥ R2 ***then***
Mem[0x5a38] = Mem[0x5a38] + 1;
else
Mem[0x5a38] = R2 + 1;
end if;



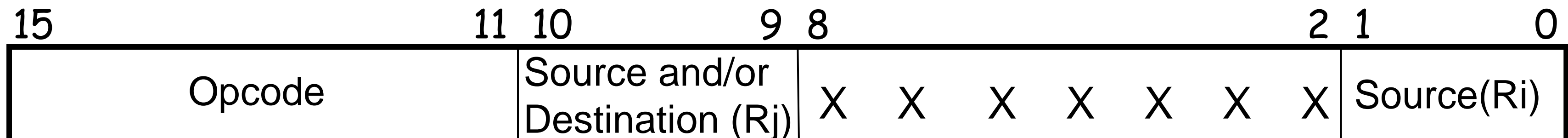
```
...
0x0010:  LDI  R0, #0x5a   // R0 ← 0x5a
0x0011:  LDI  R1, #0x38   // R1 ← 0x38
0x0012:  LD   R3, (R1)    // R3 ← Mem[0x5a38]
0x0013:  SUB  R3, R2      // R3 ← R3 + R2′ + 1
0x0014:  BNC  #0x005      // if Carry = 0 then PC ← PC + 0x005
                             (go to address 0x0019).
0x0015:  INC  R3, R3      // R3 ← R3 + 1
0x0016:  ST   (R1),R3     // Mem[0x5a38] ← R3
0x0017:  SUB  R3, R3      // This two lines are equal to a jump
0x0018:  BZ   #0x003      // instruction because always PC ← PC + 0x003
                             (always go to address 0x001B).
0x0019:  INC  R2, R2      // R2 ← R2 + 1
0x001A:  ST   (R1),R2     // Mem[0x5a38] ← R2
0x001B:  ...
```

Instruction Formats

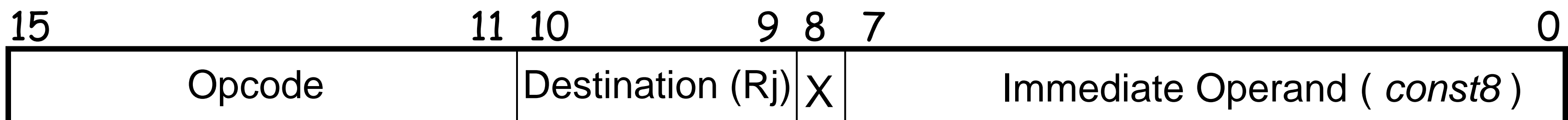
- So far, we have discussed two major components of an Instruction Set Architecture, namely *Programming Model* and *Instruction Specifications*.
- The last major component we have to discuss here is the Instruction Formats.
- Recall, an instruction is a collection of bits that instructs the processor to perform a specific operation.
- Thus, each instruction has to be encoded uniquely using bits (we will discuss this in another lecture!).
- The bits encoding an instruction have a meaning given by the Instruction Formats.

Instruction Formats for our Processor

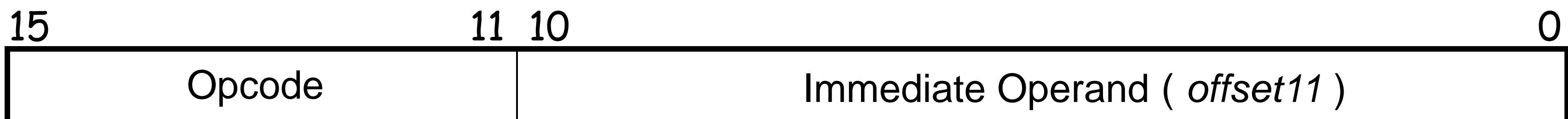
- We have 3 instruction formats because the instructions of our processor can be divided into 3 groups:
  - Instructions that require two registers (**Rj** and **Ri**) from the register file to be specified. See **Format1** below.
  - Instructions that require one register (**Rj**) from the register file and one 8-bit constant (***const***) to be specified. See **Format2** below.
  - Instructions that require one 11-bit constant (***offset***) to be specified. See **Format3** below.
- **Format1:** for Arithmetic&Logic, Shift, Memory, and Jump instructions:



- **Format2:** for Immediate Transfer instructions:



- **Format3:** for Branch instructions:

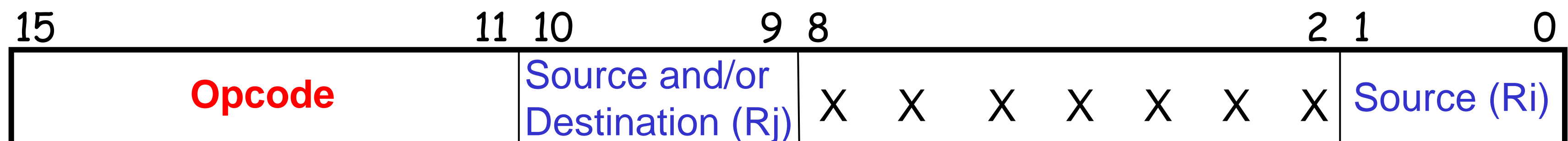


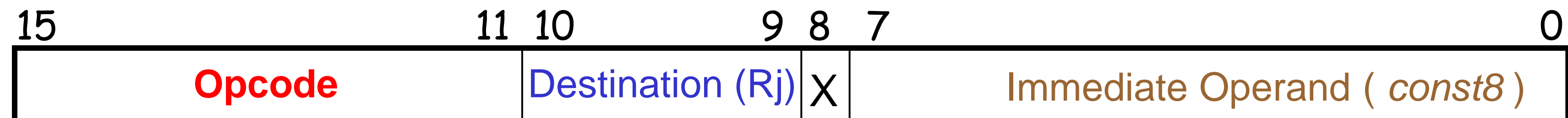
Recall the Instruction Specifications …

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 $\wedge$ Ri	Z, N	
		OR Rj, Ri	Rj ← Rj $\vee$ Ri	Z, N	
		XOR Rj,Ri	Rj ← Rj $\oplus$ 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	

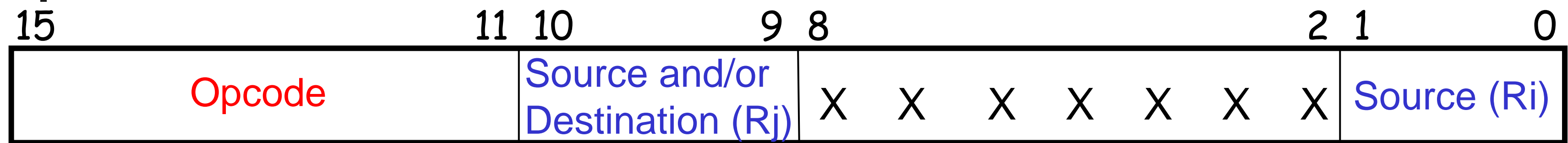
Instruction Formats for our Processor (cont.)

- Each instruction format contains 16 bits because our program memory is 16-bit wide and 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 we need at least 5-bit Opcode in order to have a unique code for each operation.
- The rest of the bits (10 to 0) specify **registers** and/or **immediate operand**.
  - Each 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.





Format1: for Arithmetic & Logic, Shift, Memory, and Jump Instructions



Encoded with 5 bits

Encoded with 2 bits

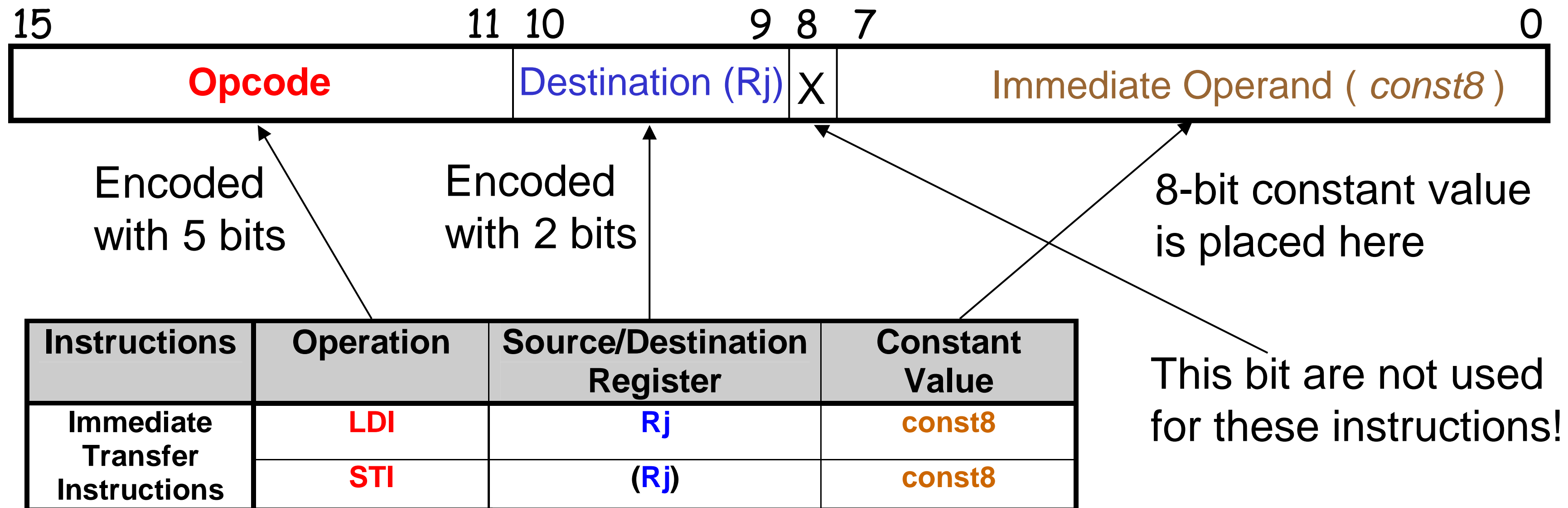
Encoded with 2 bits

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.

Format2: for Immediate Transfer Instructions



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.

Format3: for Branch Instructions

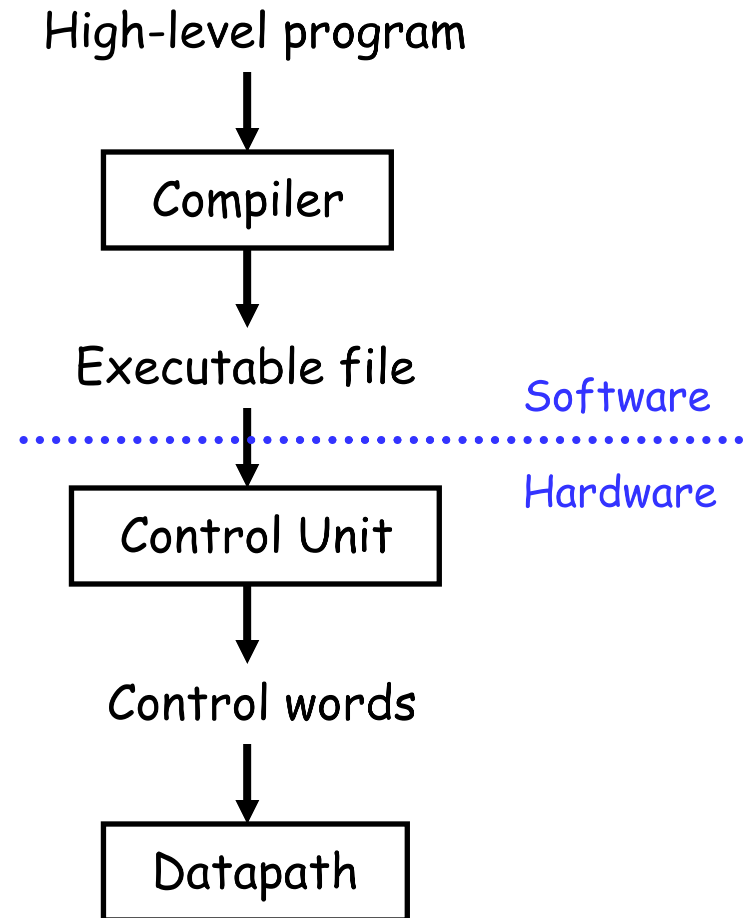
Encoded with 5 bits

11-bit constant value is placed here

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

How to Program our Processor

- Programs written in a high-level languages like C/C++ must be compiled to produce an executable program.
- The result is a processor-specific machine language program.
  - This can be loaded into the program memory and executed by the processor.
- The machine language serves as an interface between hardware and software.
- In this course we focus on stuff below the dotted blue line.
- Therefore, we will not build a compiler for our processor.
- So, we will use directly the machine language (instruction set) of our processor to program it.



High-level Languages

- High-level languages like C/C++, Java, Basic, Pascal, Fortran, Modula-2, Ada 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 and dynamic type checking
  - Garbage collection
- High-level languages are also relatively portable.
  - Theoretically, you can write one program and compile it on many different processors.
- It may be hard to understand what is so “high-level” here, until you compare these languages with...

Low-level Languages

- Each processor has its own low-level instruction set, or machine language, which closely reflects the processors' 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 not just declare new variables and use them!
  - Very little error checking is provided.
  - It is difficult to convert machine language programs to different processors.
- Later we will look at some rough translations from C to our simple processor's low-level language.

Assembly and Machine Languages

- Machine language instructions are sequences of bits in a specific order.

- To make things simpler, people typically use assembly language (as we have done so far in this lecture).
  - We assign "mnemonic" names to instructions and operands (see the instruction specifications of our processor!).
  - 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.
- In this lecture we have used assembly code for our processor to introduce the basic ideas. In your design project you will learn how to switch to machine language!

Compiling

- Processors cannot execute programs written in high-level languages directly,
  - a special program called a compiler is needed to translate high-level programs into low-level 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
    - hard for humans to write and maintain large, efficient machine language code
  - Processors are becoming more complex
    - difficult to write code that takes full advantage of a processor’s features

Simple High-level C Programs

- Consider the following simple C programs:
  - Program1: It contains a statement which executes only if some condition is true.

```
// Find the absolute value of X and increment it
int X;
...
if (X < 0) {
    X = -X; // This statement is not executed if X ≥ 0
}
X++;
```

  - Program2: It contains a loop that causes a statement to be executed many times.

```
// Sum the integers from 1 to 20 and store the result in X
int X = 0;
for (char i = 1; i <= 20; i++) {
     X = X + i;   // This statement is executed 20 times
}
```

Translating Program 1

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

```
// Find the absolute value of X and increment it
int X;
...
if (X < 0) {
    X = -X; // This statement is not executed if X ≥ 0
}
X++;
```

```
0x0010:  LDI R0, #0x00    } Load the address where X is stored in the
0x0011:  LDI R1, #0xff    } memory (we assume the address is 0x00ff)
0x0012:  LD  R3, (R1)     // load the value of X in register R3
0x0013:  ADD R3, R0       // X = X + 0
0x0014:  BNN #0x003       // if X ≥ 0 then go to address 0x0017
0x0015:  NOT R3, R3       } X = -X
0x0016:  INC R3, R3       }
0x0017:  INC R3, R3       } X++
0x0018:  ST  (R1), R3     }
```

Translating Program 2

- Here is a translation of the ***for*** loop, using branch and jump instructions.

```
// Sum the integers from 1 to 20 and store the result in X
int X = 0;
for (char i = 1; i <= 20; i++) {
      X = X + i;    // This statement is executed 20 times
}
```

```
0x0010:  LDI R1, #0x00   // X = 0
0x0011:  LDI R2, #0x01   // R2 stores the loop index i
0x0012:  LDI R3, #0x14  // R3 stores the upper loop bound 20
0x0013:  SUB R3, R2      } if i > 20 then go to address 0x001A
0x0014:  BNC #0x006      }
0x0015:  ADD R1, R2      // X = X + i
0x0016:  INC R2         // i++
0x0017:  LDI R0, #0x00
0x0018:  LDI R3, #0x12
0x0019:  JMP R0, R3     // Go back to loop test (address 0x0012)
0x001A:  LDI R0, #0x3c
0x001B:  LDI R3, #0xff
0x001C:  ST (R3), R1     // Store X in data memory at address 0x3cff
```

Summary

- The machine language is the interface between software and the processors (hardware).
- High-level programs must be translated into machine language before they can be run.
- There are three main categories of instructions.
  - Data manipulation instructions, such as adding or shifting.
  - Data movement instructions to copy data between registers and RAM.
  - Control flow instructions to change the execution order.
- Specifying Instruction Set Architecture is the first step in the design of a processor.
  - Datapath and Control Unit of a processor must be designed according to the processor's instruction set architecture.
  - The complexity of the instruction set architecture determines the complexity of the hardware needed to implement a processor.