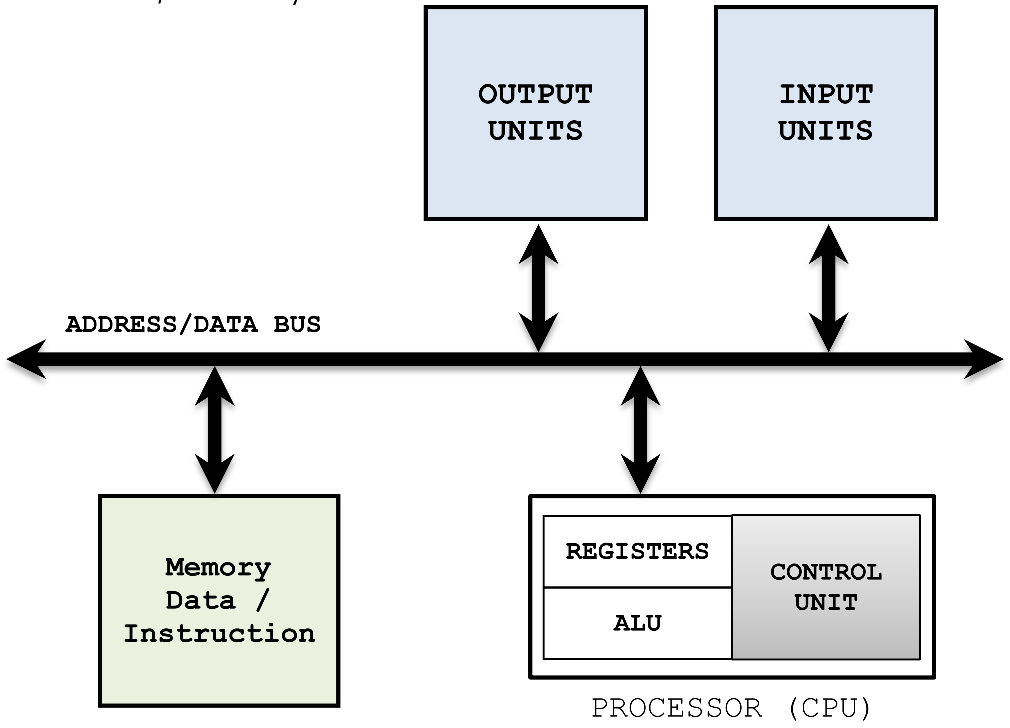
Microprocessor Design

INTRODUCTION

- Abstraction Layers in Computer Systems Design: Transistor Circuits → Logic Gates → **Register Transfers** → Microarchitecture → Instruction Set Architecture → Operating Systems → Programming Languages → Algorithm
- Typical devices used to implement digital systems (they can be implemented with a hardware-description language):
  - ✓ ASICs, FPGAs: For dedicated hardware implementation. It requires highly specialized design.
  - ✓ General-Purpose Microprocessors, Microcontrollers (e.g. embedded). Software development.
  - ✓ Specialized uPs: PDSPs (programmable digital signal processor).
- ASICs or uPs? Performance vs. flexibility. ASIC design requires high development cost, not reprogrammable.
- FPGAs: Intermediate option between ASICs and uP. Not commonly used for processor implementation. Operating frequencies can be relatively low compared to uP, but can achieve higher performance for specific tasks. They are reconfigurable.
- PSoCs (Programmable System-on-Chip). They integrate reconfigurable logic (like an FPGA), a hard-wired microprocessor, and peripherals. With proper software/hardware co-design, high performance solutions can be attained.

COMPUTER HARDWARE ORGANIZATION

- **Computer**: Processor + I/O + Memory



CENTRAL PROCESSING UNIT

- Also called Processor. It consists of a Datapath and Control Unit
  - ✓ Datapath:
    - ▫ **Register File** (set of Registers): They hold data and memory address values during the execution of an instruction.
    - ▫ **Arithmetic Logic Unit (ALU)**: Shared operation unit that performs arithmetic (e.g., addition, subtraction, division) and bit-wise logic (e.g., AND, OR, operations).
  - ✓ **Control Unit:** To control operations performed on the Datapath and other components (e.g. memory). It is in charge of executing instructions. Instructions are read from memory. To execute a particular instruction, this unit asserts specific signals at certain times to control the registers, ALU, memories and ancillary logic. A Control Unit is usually implemented as a large Finite State Machine (FSM) with ancillary logic.
- Complex CPU: Multiple control units and datapaths.

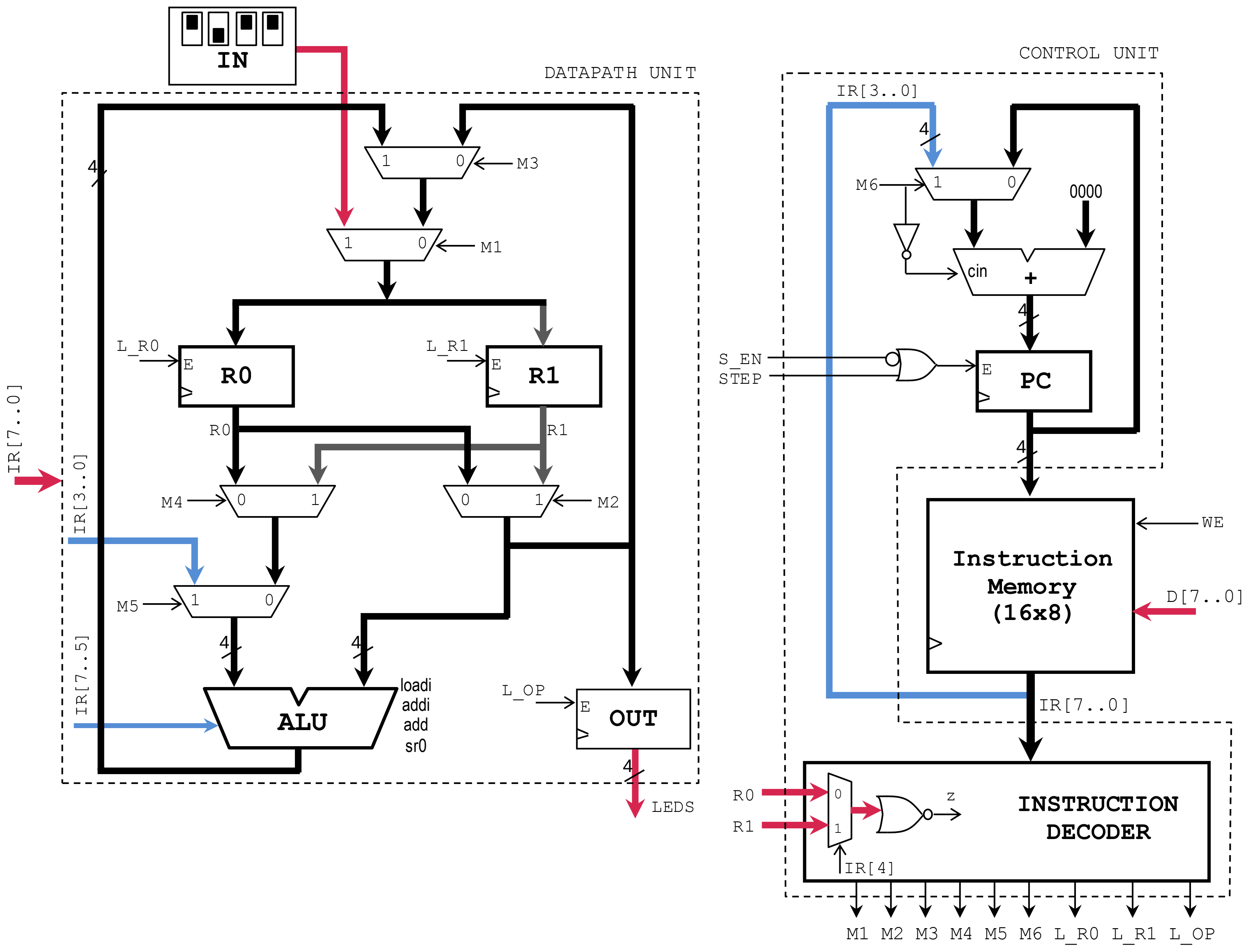
Harvard vs. Von Neumann

Harvard:	▫ Instruction memory and Data memory ▫ Operands usually placed in registers in the CPU: register-to-register architecture
Von Neumann:	▫ One memory for both instruction and data ▫ Operands placed in an accumulator register or in the instruction memory: register-memory architecture

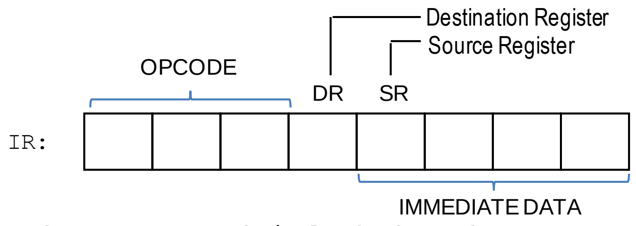
SINGLE-CYCLE HARDWIRED CONTROL

BASIC PROCESSOR

- Only one instruction memory. No data memory.
- Data can be loaded onto the ALU on one clock cycle.



- **Available Registers**: `R0` (register 0, 4 bits), `R1` (register 1, 4 bits), `OUT` (output register, 4 bits), `PC` (program counter, 4 bits), `IR` (instruction register, 8 bits).
- **Instruction Memory**: Stores up to 16 8-bit instructions
- **Program Counter (PC)**: To execute the instructions in sequence, it is necessary to provide the address in memory of the instruction to be executed. In a computer, this address comes from a register called PC.
- **Instruction Decoder**: Converts instructions into control bits. This is a combinational circuit.
- **Instruction Set**: Instructions are specified in the Instruction Register (`IR`).



`DR=0` ⇒ `R0` is the Destination register, `DR=1` ⇒ `R1` is the Destination register.
`SR=0` ⇒ `R0` is the Source register, `SR=1` ⇒ `R1` is the Source register.

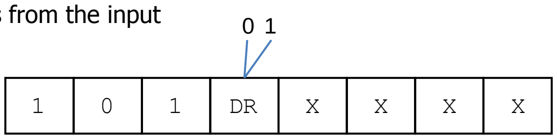
OPCODE (IR[7..5])	Instruction	Operation
000	MOV DR, SR	DR ← SR
001	LOADI DR, DATA	DR ← DATA, DATA = IR[3..0]
010	ADD DR, SR	DR ← DR + SR
011	ADDI DR, DATA	DR ← DR + DATA, DATA = IR[3..0]
100	SR0 DR, SR	DR ← 0&SR[3..1]
101	IN DR	DR ← IN
110	OUT DR	OUT ← DR

111	JNZ DR, ADDRESS	PC ← PC + 1 if DR=0
		PC ← IR[3..0] if DR≠0
		* ADDRESS = IR[3..0]

- opcode: IR[7..5]: This is the *operation code* of an instruction. This group of bits specifies an operation (such as add, subtract, shift, complement in the ALU). If it has m bits, there can be up to 2^m distinct operations.
- Immediate Data: IR[3..0]. This is called an immediate operand since it is immediately available in the instruction.

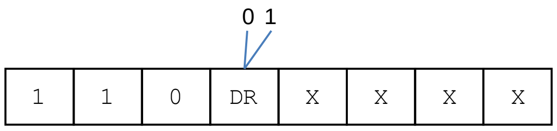
Executing Instructions:

✓ IN DR: DR grabs the contents from the input



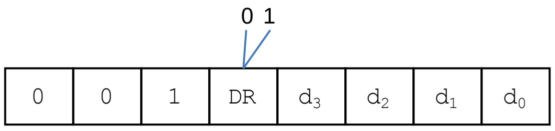
IN R0: 1010XXXX ⇒ M1 ← 1, L_R0 ← 1, M6 ← 0
IN R1: 1011XXXX ⇒ M1 ← 1, L_R1 ← 1, M6 ← 0

✓ OUT DR: Places the contents of DR on the output register



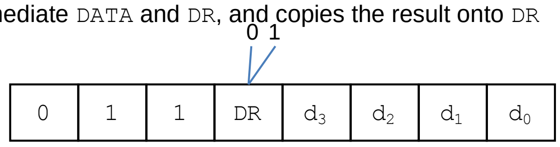
OUT R0: 1100XXXX ⇒ M2 ← 0, L_OP ← 1, M6 ← 0
OUT R1: 1101XXXX ⇒ M2 ← 1, L_OP ← 1, M6 ← 0

✓ LOADI DR, DATA: Copies immediate DATA onto DR



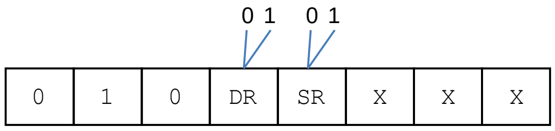
LOADI R0, DATA: 0010$d_3d_2d_1d_0$ ⇒ M5 ← 1, M3 ← 1, M1 ← 0, L_R0 ← 1, M6 ← 0
LOADI R1, DATA: 0011$d_3d_2d_1d_0$ ⇒ M5 ← 1, M3 ← 1, M1 ← 0, L_R1 ← 1, M6 ← 0

✓ ADDI DR, DATA: Adds immediate DATA and DR, and copies the result onto DR



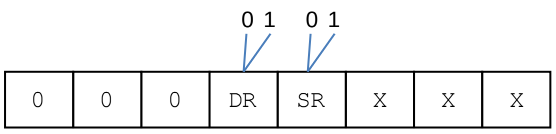
ADDI R0, DATA: 0110$d_3d_2d_1d_0$ ⇒ M2 ← 0, M5 ← 1, M3 ← 1, M1 ← 0, L_R0 ← 1, M6 ← 0
ADDI R1, DATA: 0111$d_3d_2d_1d_0$ ⇒ M2 ← 1, M5 ← 1, M3 ← 1, M1 ← 0, L_R1 ← 1, M6 ← 0

✓ ADD DR, SR: Adds SR and DR, and copies the result onto DR



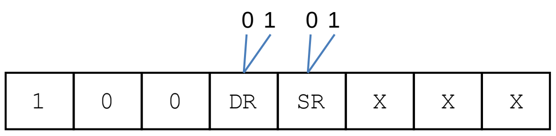
ADD R0,R0: 01000XXX ⇒ M4←0, M5←0, M2←0, M3←1, M1←0, L_R0←1, M6←0
ADD R0,R1: 01001XXX ⇒ M4←0, M5←0, M2←1, M3←1, M1←0, L_R0←1, M6←0
ADD R1,R0: 01010XXX ⇒ M4←0, M5←0, M2←1, M3←1, M1←0, L_R1←1, M6←0
ADD R1,R1: 01011XXX ⇒ M4←1, M5←0, M2←1, M3←1, M1←0, L_R1←1, M6←0

✓ `MOV DR, SR`: Copies the contents of `SR` onto `DR`



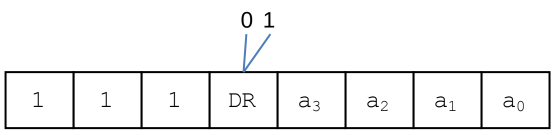
```
MOV R0, R0: 00000XXX ⇒ M2 ← 0, M3 ← 0, M1 ← 0, L_R0 ← 1, M6 ← 0
MOV R1, R1: 00011XXX ⇒ M2 ← 1, M3 ← 0, M1 ← 0, L_R1 ← 1, M6 ← 0
MOV R0, R1: 00001XXX ⇒ M2 ← 1, M3 ← 0, M1 ← 0, L_R0 ← 1, M6 ← 0
MOV R1, R0: 00010XXX ⇒ M2 ← 0, M3 ← 0, M1 ← 0, L_R1 ← 1, M6 ← 0
"MOV R0,R0", "MOV R1,R1"(can be used as NOP instruction)
```

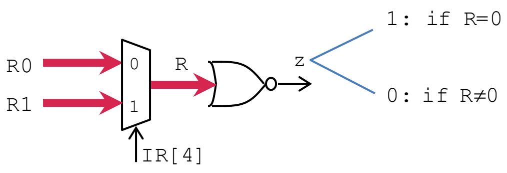
✓ `SR0 DR, SR`: Shifts (to the right) the contents of `SR` and places the result onto `DR`



```
SR0 R0,R0: 10000XXX ⇒ M4←0, M5←0, M2←0, M3←1, M1←0, L_R0←1, M6←0
SR0 R0,R1: 10001XXX ⇒ M4←0, M5←0, M2←1, M3←1, M1←0, L_R0←1, M6←0
SR0 R1,R0: 10010XXX ⇒ M4←0, M5←0, M2←1, M3←1, M1←0, L_R1←1, M6←0
SR0 R1,R1: 10011XXX ⇒ M4←1, M5←0, M2←1, M3←1, M1←0, L_R1←1, M6←0
```

✓ `JNZ DR, ADDRESS`: Jumps to a certain instruction if DR≠0. This is how computers implement loops.





JNZ R0, ADDRESS: 1110$a_3a_2a_1a_0$ ⇒ M6 ← 0 if z = 1, M6 ← 1 if z = 0

JNZ R1, ADDRESS: 1111$a_3a_2a_1a_0$ ⇒ M6 ← 0 if z = 1, M6 ← 1 if z = 0

* M6 ← 0 ≡ PC ← PC + 1; M6 ← 1 ≡ PC ← IR[3..0]

Example:

▪ Write an assembly program for a counter from 1 to 5: 1, 2, 3, 4, 5, 1, 2, 3, …. The count must be shown on the output register (OUT).

```
start: loadi R0,1
       out R0     → OUT = 1
       addi R0,1
       out R0     → OUT = 2
       addi R0,1
       out R0     → OUT = 3
       addi R0,1
       out R0     → OUT = 4
       addi R0,1
       out R0     → OUT = 5
       jnz R0, start
```

Example:

- Write an assembly program for a counter from 2 to 13: 2,3,…, 13,2,3,… The count must be shown on the output register (OUT). Use labels to specify any address where your program jumps. Note that you can have only up to 16 instructions.
- Provide the contents of the Instruction Memory.

address	INSTRUCTION MEMORY
0000	00100010
0001	00110100
0010	11000000
0011	01100001
0100	01110001
0101	11110000
0110	00100001
0111	11100010
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

```
start: loadi R0,2
       loadi R1,4
loop:  out R0 → OUT: shows the count
       addi R0,1
       addi R1,1
       jnz R1, loop
       loadi R0,1
       jnz R0, start
```

Single-Cycle Computer Shortcomings:

- ALU operations that might require more than one cycle to execute (e.g. multiplication, division) cannot be executed.
- Lower limit on the clock period based on a long worst-case delay path. Pipelining of the datapath is required to reduce the combinational delay between registers. This requires multiple-cycle control.

MULTIPLE-CYCLE HARDWIRED CONTROL

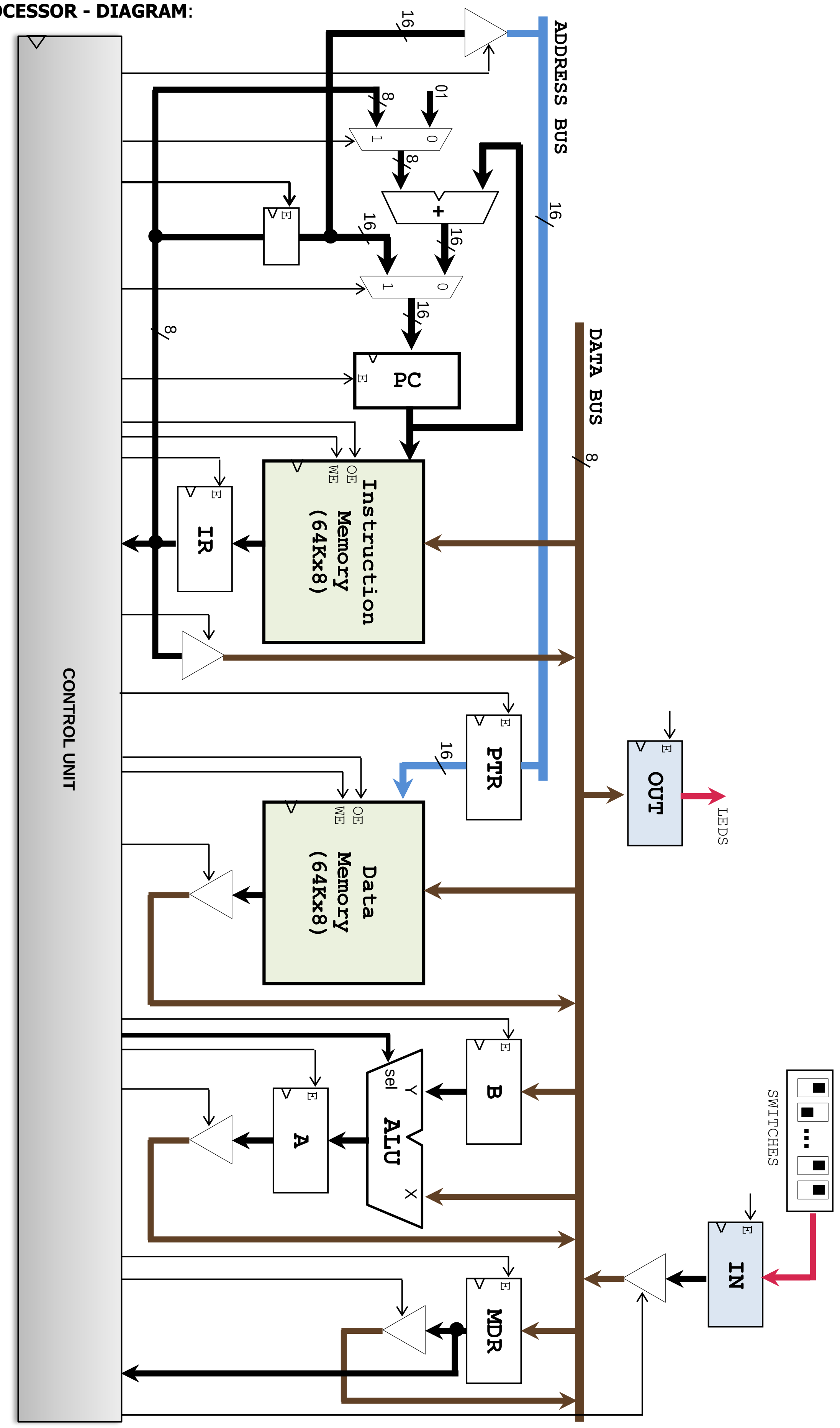
SIMPLE PROCESSOR

- Instruction (or Program) Memory and Data Memory separated (Harvard architecture).
- A handful of registers: A, PC, PTR, MDR, IR
- 8-bit processor: Data bus is 8-bits wide
- Data and Instruction memories: Memory contents are 8-bits wide. The size of the memories is 64 KB, this requires 16 bits to address all bytes. Address bus is then 16-bits wide.
- **Program Counter (PC)**: Keeps track of address of the instruction to be executed next. It points to the Instruction Memory. It can be i) incremented by 1, ii) incremented by an 8-bit offset, iii) modified to an arbitrary 8-bit or 16-bit address.
- PTR Register (16 bits): Points to data memory.
- **Control Unit:** It includes an Instruction Decoder and logic to update the Program Counter.
  - ✓ Instruction Decoder: Since this is multiple-cycle control, the Instruction Decoder includes a FSM along with ancillary logic. Look at the '*Simple Processor*' in **Unit 4** for an FSM example: instructions are processed sequentially, once an instruction finishes its execution, we can load the next instruction.

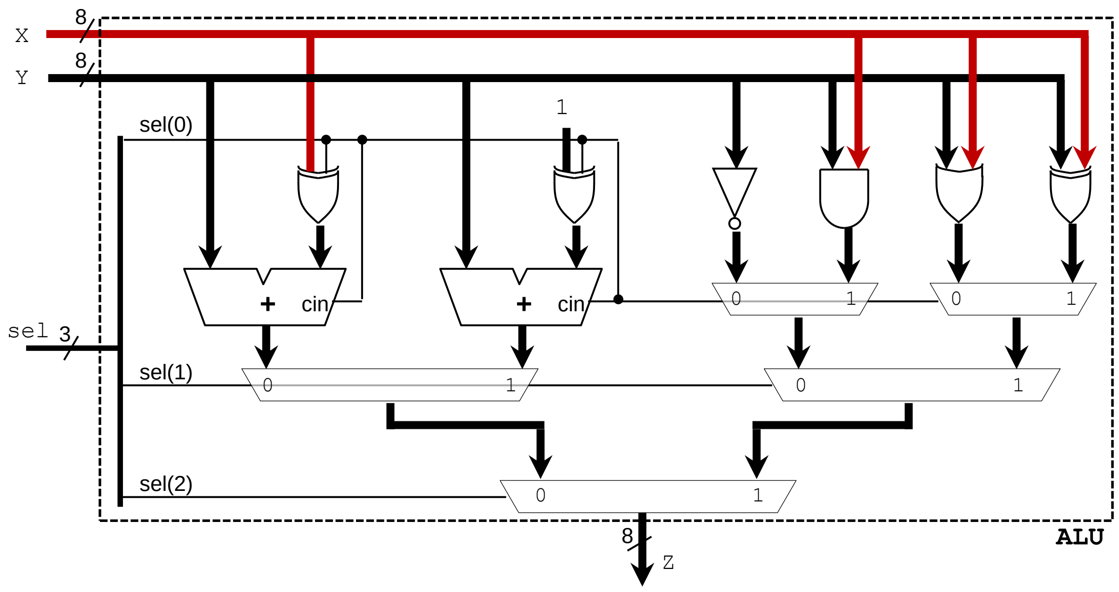
PROGRAM EXECUTION

- The processor reads instructions from the Instruction Memory (one after the other) and executes them. The instructions are a string of bytes that specify a particular operation to be carried out. The Control Unit asserts specific signals at the proper times so as to execute a particular instruction. The results of the operation can be stored in a register, data memory, and/or and output interface. Some instructions might modify the Program Counter at will.
- At power up, usually the processor reads the instruction from the first memory address. This is done by asserting the reset input of the Program Counter Register at power up.

SIMPLE PROCESSOR - DIAGRAM:



ARITHMETIC LOGIC UNIT:



sel	Operation	Function	Unit
000	Z <= X + Y	Add X and Y	Arithmetic
001	Z <= X - Y	Subtract Y from X	
010	Z <= X + 1	Increment X	
011	Z <= X - 1	Decrement X	
100	Z <= not(X)	Complement X	Logic
101	Z <= X and Y	Bitwise AND	
110	Z <= X or Y	Bitwise OR	
111	Z <= X xor Y	Bitwise XOR	

INSTRUCTION SET:

- The format of the instructions is as follows: OPCODE (1 byte) | (optional fields: up to 2 bytes)
  - ✓ OPCODE: Unique identifier that specifies a particular instruction.
  - ✓ Other fields: usually they are address and data values
- An instruction may take up to 3 bytes in memory.
- Collection of available instructions:

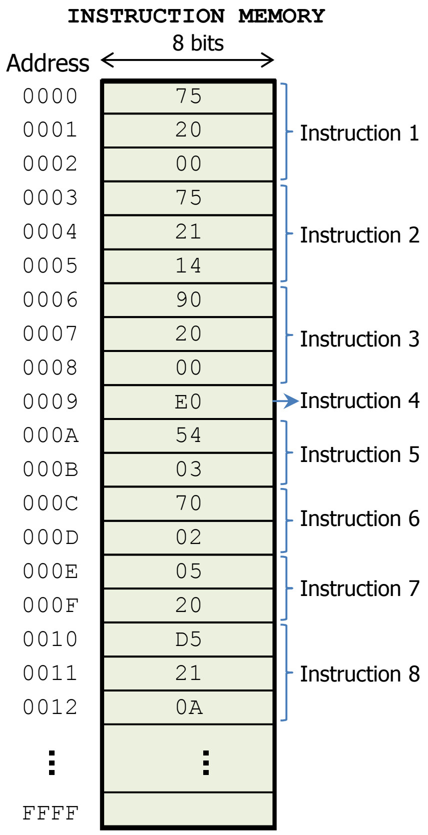
Assembly Instruction mnemonic	Machine code	Meaning
`ld addr,#val`	`75 aa xx`	Load 8-bit value (`val`) into memory location at `addr`.
`ld ptr, #data`	`90 yy yy`	Load 16-bit value (`data`) into register `ptr`.
`ld A, @ptr`	`E0`	Load contents of memory location pointed to by `ptr` into `A`.
`and A, #val`	`54 xx`	Bit-wise AND: `A ← A and val`
`bnz A, target`	`70 zz`	If `A≠0`, then branch to the address provided by target If `A=0`, then go to next instruction
`inc addr`	`05 aa`	Increments the contents of memory location at `addr`.
`dbnz addr, target`	`D5 aa zz`	Decrements the contents of memory location at `addr` and branch if the result is not zero.

Observations:

- Most of the instructions in the table above only allow 8 bits to specify a memory address.
- PTR is a 16-bit register. To load a 16-bit value on it, we require two read cycles from Instruction Memory. This is taken care of by an intermediate register that grabs 8 bits at a time.
- `bnz, dbnz`: A label specifies the instruction to branch to. The assembler determines the address to branch to and specifies it in the machine code (here we can only branch to the first 256 memory positions as only one byte is available to specify the address to branch to).
  Alternative assembler approach: the assembler determines an offset. This way we can branch (up or down) 256 positions from the current instruction. This approach requires specification of whether to add or subtract the offset from the PC value, thereby increasing the machine code and requiring the PC circuitry to allow for addition and subtraction.

SAMPLE PROGRAM:

Assembly Instruction mnemonic	Start Address	Machine Code (hex)	Comments
`ld 0x20,#0`	`0x0000`	`75 20 00`	
`ld 0x21,#20`	`0x0003`	`75 21 14`	20 is a decimal value
`ld ptr,#0x2000`	`0x0006`	`90 20 00`	
`loop: ld A, @ptr`	`0x0009`	`E0`	Only 1 byte required
`and A,#0x03`	`0x000A`	`54 03`	
`bnz A, next`	`0x000C`	`70 10`	
`inc 0x20`	`0x000E`	`05 20`	
`next: dbnz 0x21, loop`	`0x0010`	`D5 21 09`	



1) `ld 0x20,#0`: Load `0x00` at address `0x0020`. Instruction requires 3 bytes.
The OPCODE (`75`) is read from Instruction Memory. The next byte (`0x20`) is read and placed on PTR (left-appending zeros) via the address bus. The following byte (`0x00`) is read and placed on the data bus. Then we write `0x00` in Data Memory at address `0x0020` (alternatively, we can store `0x00` on MDR, this allows for flexibility to choose when to write on Data Memory).
Start of Instruction: PC: `0x0000` End of Instruction: PC: `0x0003`

2) `ld 0x21,#20`: load `0x14` at address `0x0021`. Instruction requires 3 bytes.
We write `0x14` in Data Memory at address `0x0020` (as in instruction 1)
Start of Instruction: PC: `0x0003` End of instruction: PC: `0x0006`

3) `ld ptr,#0x2000`: load `0x2000` on PTR. Instruction requires 3 bytes.
The OPCODE (`90`) is read from Instruction Memory. The next byte (`0x20`) is placed on the least significant byte of PTR. The following byte (`0x00`) is placed on the most significant byte of PTR.
Start of Instruction: PC: `0x0006` End of Instruction: PC: `0x0009`

4) `ld A, @ptr`: contents of the memory location pointed to by PTR are loaded into A. This instruction requires 1 byte.
The OPCODE (`E0`) is read from Instruction Memory. The data located at the address pointed to by PTR (`0x2000` in this example) is read and placed on the ALU where we transfer it to register A (this ALU transfer can be done by setting the register B to `0x00` and performing a bit-wise OR)
Start of Instruction: PC: `0x0009` End of Instruction: PC: `0x000A`

5) `and A,#0x03`: `A ← A AND 0x03`. This instruction requires 2 bytes.
The OPCODE (`54`) is read from Instruction Memory. The next byte (`0x03`) is placed on register B via the data bus. Then a bitwise AND with A is performed on the ALU, the result is stored in A.
Start of Instruction: PC: `0x000A` End of Instruction: PC: `0x000C`

6) `bnz A, next`: Branch to `next` (`0x0010`) if A is nonzero. This instruction requires 2 bytes.
The OPCODE (`70`) is read from Instruction Memory. The next byte (`0x10`) is read and placed on IR. To compare A with zero, A is placed on MDR. If A=0, PC is incremented by 1 (as usual). If A≠0, the value at IR (`0x10`) is placed on PC.
Start of Instruction: PC: `0x000C`
End of Instruction: If we do not branch (A is zero), then PC: `0x000E`. If we branch (A is nonzero), then PC: `0x0010`

7) `inc 0x20`: The data located in address `0x0020` is incremented by 1. This instruction requires 2 bytes.
The OPCODE (`05`) is read from Instruction Memory. The next byte (`0x20`) is placed on PTR, and we read the data at `0x0020`. The data is placed on the ALU, where it is incremented by 1 and placed in A. Then, the contents of A are written on the Data Memory at memory location specified by PTR (`0x0020`).
Start of Instruction: PC: `0x000E` End of Instruction: PC: `0x0010`

8) `dbnz 0x21, loop`: Decrement the data located in address `0x0021`. If the result is nonzero, branch to `loop` (`0x0009`)
The OPCODE (`D5`) is read from Instruction Memory. This instruction requires 3 bytes. The next byte (`0x21`) is placed on PTR. The next byte (`0x09`) is read and placed on IR. Then we read data at `0x0021` (pointed to by PTR). The data is placed on the ALU, (where it is decremented by 1 and placed in A. A is then placed on MDR to compare A with zero. If it is zero, we increment PC by 1 (as usual). If A is nonzero, the value on IR (`0x09`) is placed on PC.
Start of Instruction: PC: `0x0010`
End of Instruction: If we do not branch, then PC: `0x0013`. If we branch, then PC: `0x0009`*b*

MORE COMPLEX PROCESSORS

- Instruction/Data Memory might be located in the same chip. The addressing control is more complex. Also, memory control is more involved as we are requires to deal with different technologies (e.g.: DRAM, SRAM, DDRRAM).
- ALU: It includes the common outputs V (overflow), C (carry out), N (sign), Z (zero); these are called **Status Bits** (usually stored in a register), and are very useful during the execution of instructions. Also, it includes a 'carry in' input for multi-operand addition/subtraction.
- Branching instructions: Conditional (e.g.: bz, bnz, bnn, bn, bv, bnv, bc, bnc) and unconditional branches (jmp).

- Pipelined control: So, far the FSM inside the control unit can only process one instruction once the previous one has been completed. By pipelining the datapath, we can execute the next instruction before the current instruction finishes. The design of the FSM is thus more complex.
  - ✓ Instruction Fetch (IF): Instruction is fetched from the instruction memory and the value of PC is updated.
  - ✓ Instruction Decode (ID): The instructions is turned into control signals
  - ✓ Operand Fetch (OF): The instruction operands are retrieved (from registers or memory) and placed on registers.
  - ✓ Execute Stage (EX): An ALU operation is executed.
  - ✓ Write Back (WB): Data might be written back into the Data memory.

- Addressing modes: They determine how the CPU instructions access operands (usually memory locations). For example:
  - ✓ Inherent: Instructions using this addressing mode have either no operands or the operands are CPU registers. CPU does not access memory locations. Example (HCS12):
    ```
    nop; no operation
    iny; Y <= Y+1
    clrb;  B <= 0
    tfr D, Y; Y <- D
    ```

  - ✓ Immediate: Operand values embedded in the instruction. Example (HCS12):
    `ldax #$4C32;` Register X gets the number `0x4C32`.

  - ✓ Direct: Instructions access operands from memory locations. Example (HCS12):
    `ldaa $E109;` Register A gets the contents of memory address `0xE109`.

  - ✓ Relative: Used for branch instruction. The distance of the branch (of jump) is called *offset*. Example (HCS12):
    `bmi start;` If status bit N is 1 then PC <-PC + offset; otherwise PC <- PC+1. *offset* is calculated by the compiler based on the address of the current instruction and the instruction address where we are supposed to branch.

  - ✓ Indexed: Instructions access operands from memory locations. However, the memory address is computed based on a based address (usually from a register) and an offset. Example (HCS12).
    `ldaa B, X;` Register A gets the contents of the memory address B+X.

- Numeric co-processors (e.g. pipelined CORDIC, square root) are included.
- Data and instruction cache are included.
- A Stack is included for managing subroutines and interrupts.

INSTRUCTION SET ARCHITECTURES

- Two main types of instruction set architectures that greatly differ in the relationship of hardware to software:
  - ✓ CISC (complex instruction set computer): It provides hardware support for high-level language operations and have compact programs.
  - ✓ RISC (reduced instruction set computer): It emphasizes simple instructions and flexibility.
- The instruction set of a CISC machines is rich, while a RISC machine typically supports fewer than 100 instructions.
- The word length of RISC instructions is fixed, and typically 32 bits long. In a CISC machine the word length is variable. In Intel machines we find instructions from 1 to 15 bytes long.
- The addressing modes supported by a CISC machine are rich, while in a RISC machine only very few modes are supported. A typically RISC machine supports just immediate and register based addressing.
- The operands for ALU operations in a CISC machine can come from instruction words, registers, or memory. In a RISC machine, memory access are restricted to load and store instructions; data manipulation instructions are register-to-register; thus, a RISC machine are called load/store architectures.

I/OS

- They are not considered part of the CPU. One of the most complex areas of computer design, due to the diversity of devices.
- Example: USB, Ethernet, SPI, I^{2}C, displays, hard drive, etc.