Processor Design Fundamentals

NYU

EL6463 2021

What makes a “computer”?

- Turing Machines, by Alan Turing in *On Computable Numbers* (1936)

  (simplified:)

  - A device with infinite storage (tape)
  - A computational unit (head on the tape)
  - The head can move forwards, backwards, write data, or erase data
  - The head’s behaviour is governed by a FSM

- In other words, a machine with *memory* and *computation*

Universal Turing Machine

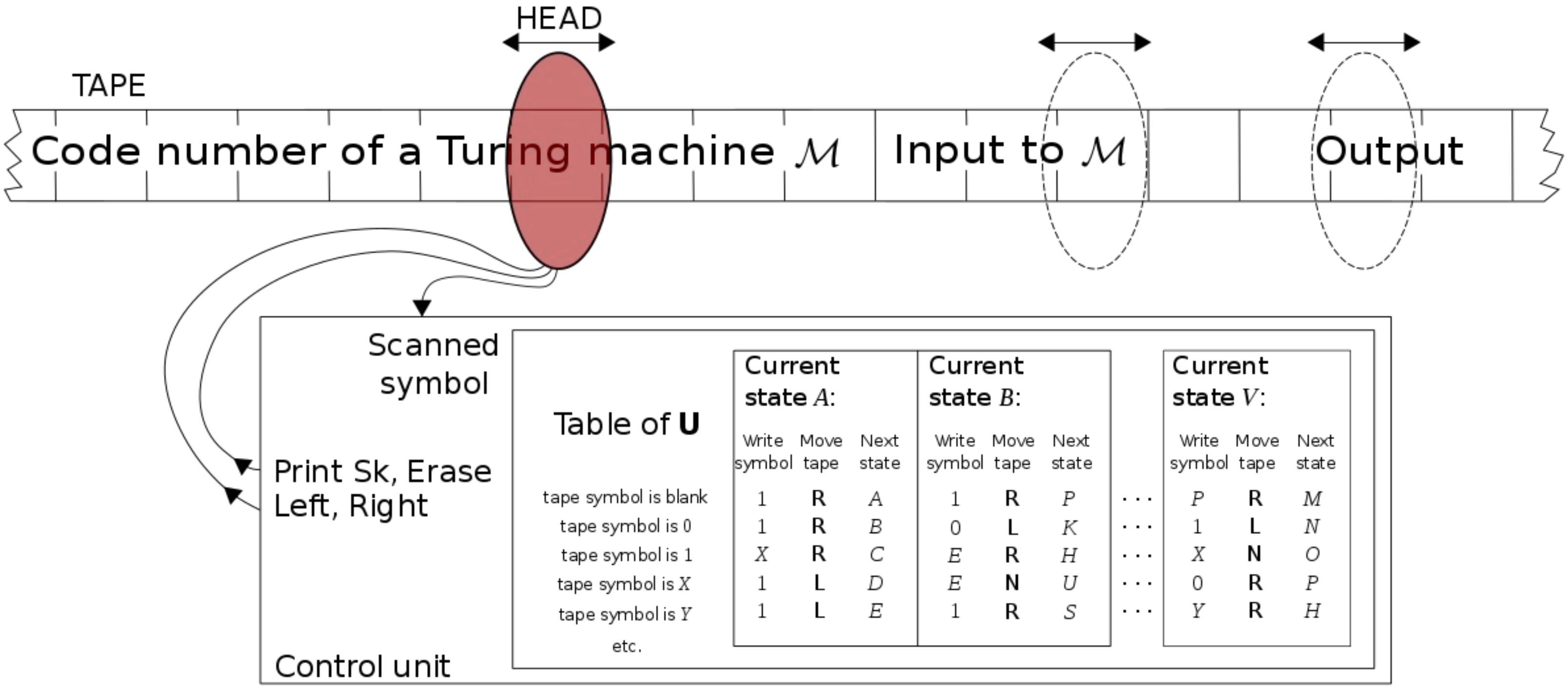


Table of **U**

	Current state A:			Current state B:				Current state V:		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state		Write symbol	Move tape	Next state
tape symbol is blank	1	**R**	A	1	**R**	P	. . .	P	**R**	M
tape symbol is 0	1	**R**	B	0	**L**	K	. . .	1	**L**	N
tape symbol is 1	X	**R**	C	E	**R**	H	. . .	X	**N**	O
tape symbol is X	1	**L**	D	E	**N**	U	. . .	0	**R**	P
tape symbol is Y	1	**L**	E	1	**R**	S	. . .	Y	**R**	H
etc.										

By Cbuckley - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3097974

Universal Turing Machine

- A Turing machine computes a fixed function.
- However, we can encode the action table of any Turing machine in a string.
- Thus we make a Turing machine that first reads a program then executes it.

"It is possible to invent a single machine which can be used to compute any computable sequence. If this machine U is supplied with a tape on the beginning of which is written the S.D ["standard description" of an action table] of some computing machine M, then U will compute the same sequence as M."

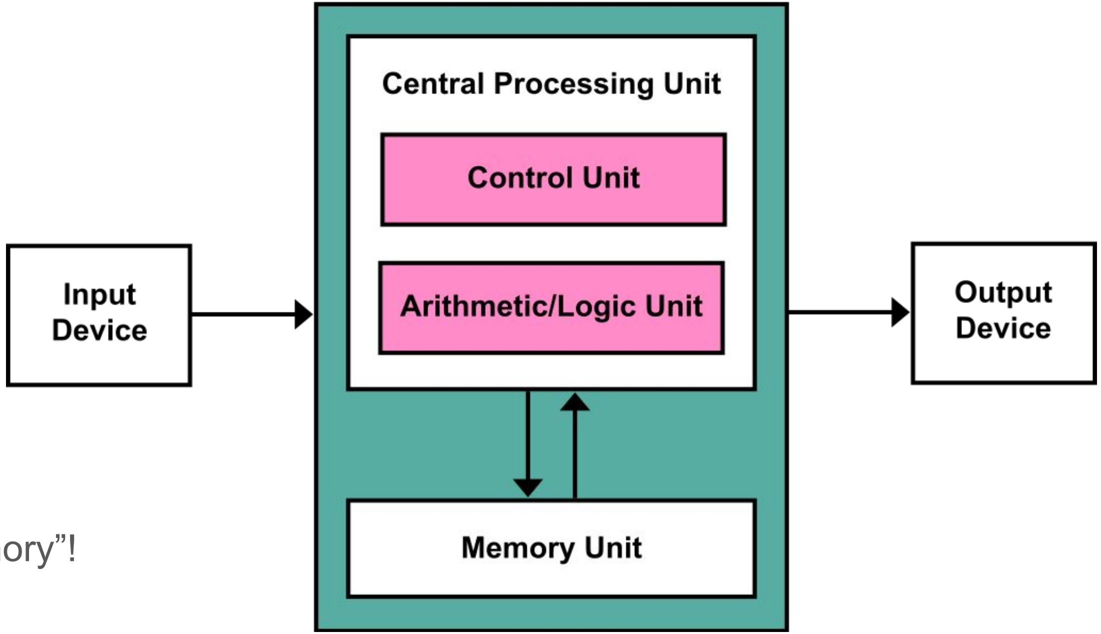
- Alan Turing, 1936

What makes a computer?

- A memory
- A computation unit
- A program (instructions)

You might also consider

- Inputs and Outputs
  - These can be part of the "memory"!



By Kapooht - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=25789639

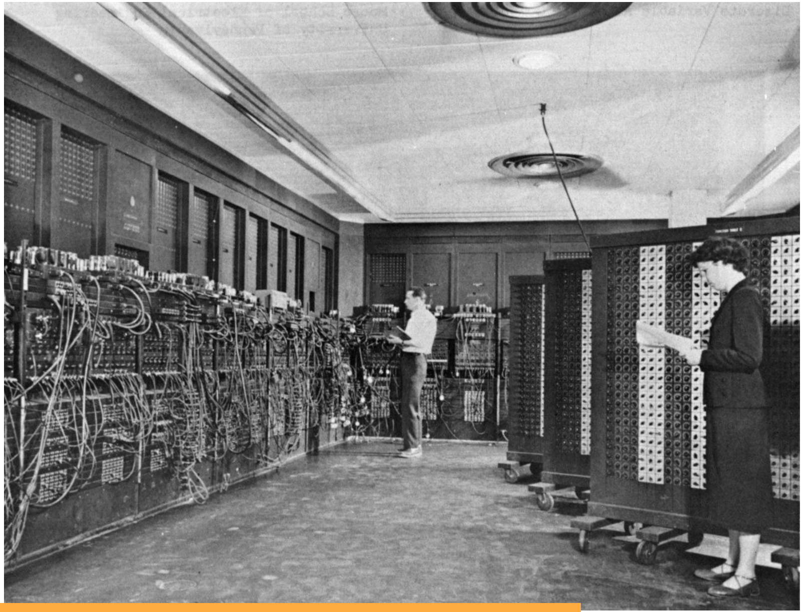
Implementing a Computer / A Turing Machine

- In this course we’ve actually taught you all the essential components to make simple Computers / Turing Machines

- Registers - for memory (data and program storage)
- ALUs, LUTs - for arbitrary computation
- FSMs - for governing the behaviour of the system (or a fixed program)

What's under the hood? Simple Computers

- The first digital computers were built with the ideas provided by Turing
- ENIAC (1946) - Turing machine, EDVAC (1949) - 1945

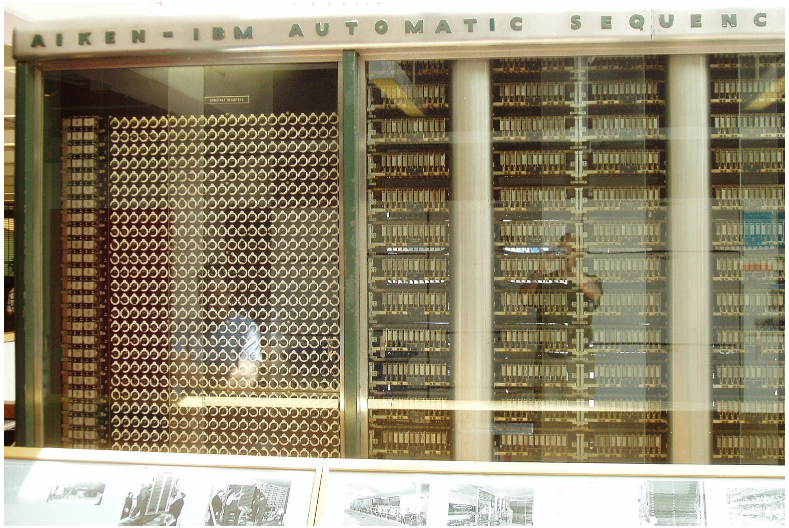


"Programmed" by re-wiring modules

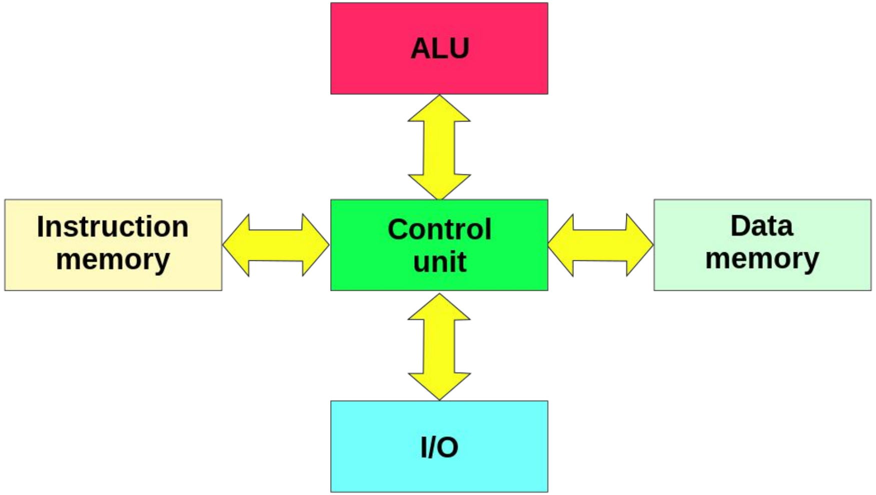
First "stored program" computer
- Von Neumann architecture

Harvard Architecture

- Von Neumann architecture has one memory unit
  - places severe limits on execution speed due to memory contention
- Harvard Architecture separates the memories into “instruction” and “data”
  - Origin: Harvard Mark 1 (1944) Relay-based computer



By Daderot CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1831682

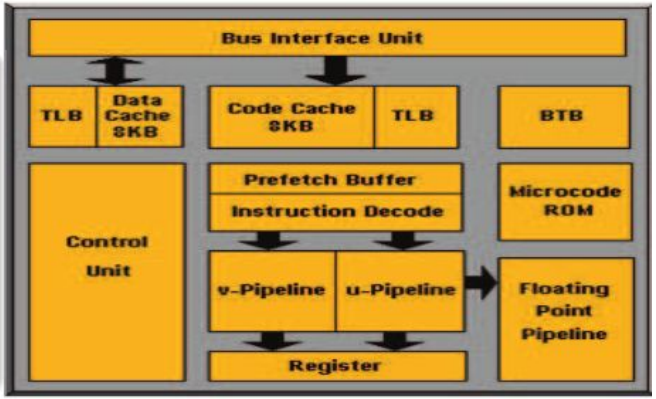


By Nessa los - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=10303637

Examples of Real Architectures

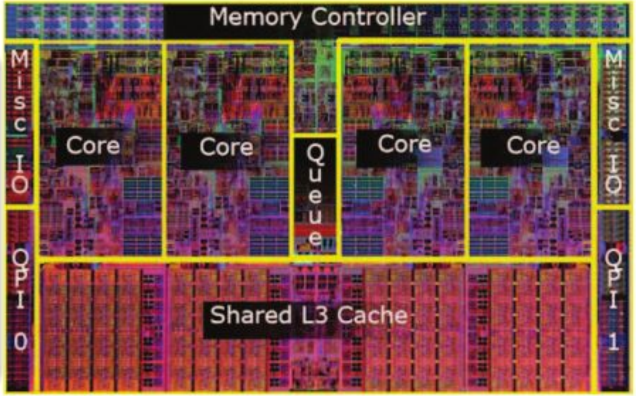
Pentium Processor Architecture (simplified)

An example of architectural innovations made by processor designers (to improve computer systems performance) (Source: Intel)



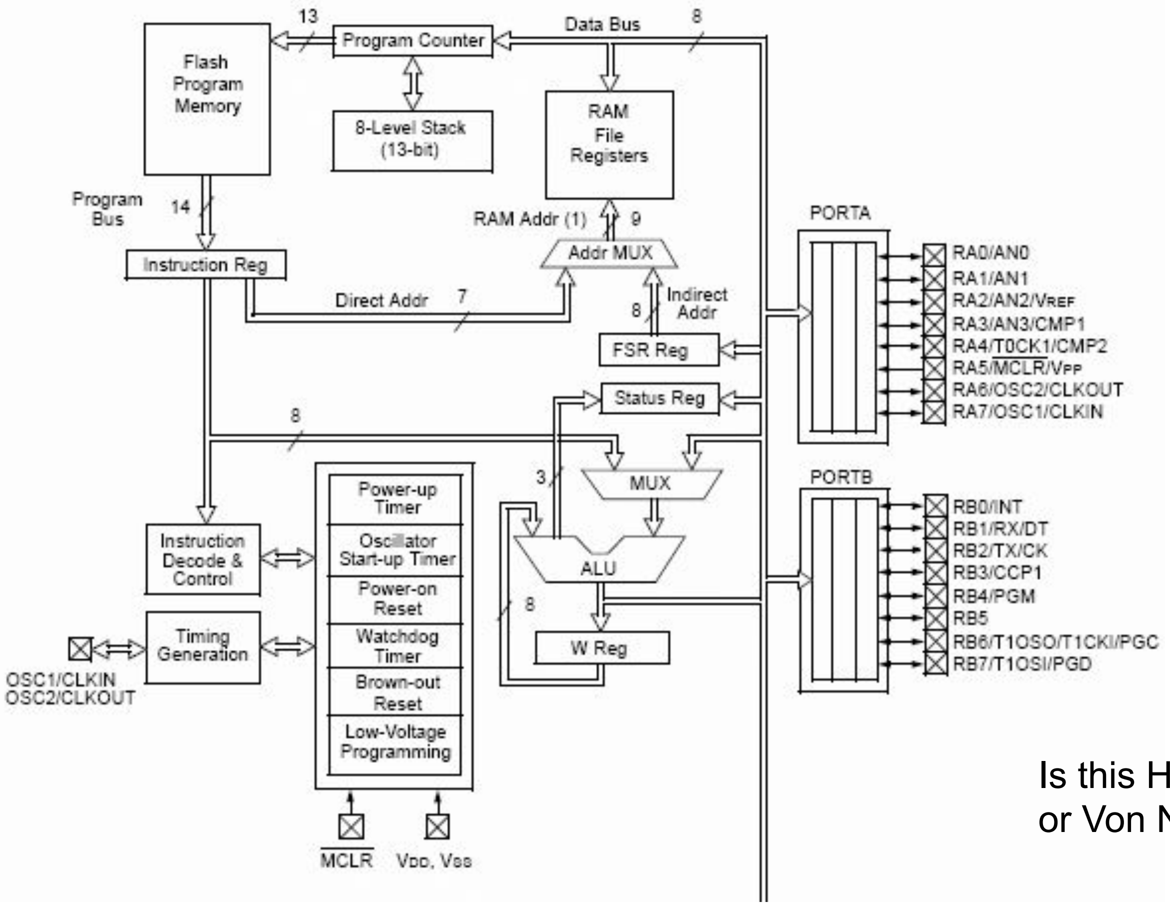
Intel Core i7

The Core i7 die and major components. (more than 700 million transistors in a die with area of $263mm^2$ in 45nm technology). (Source: Intel)



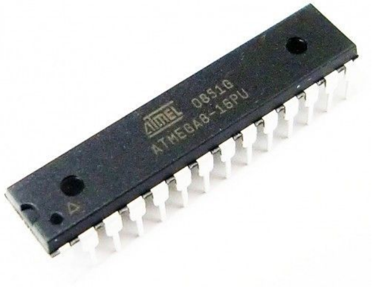
CS302 Resources - Morteza Biglari-Abhari

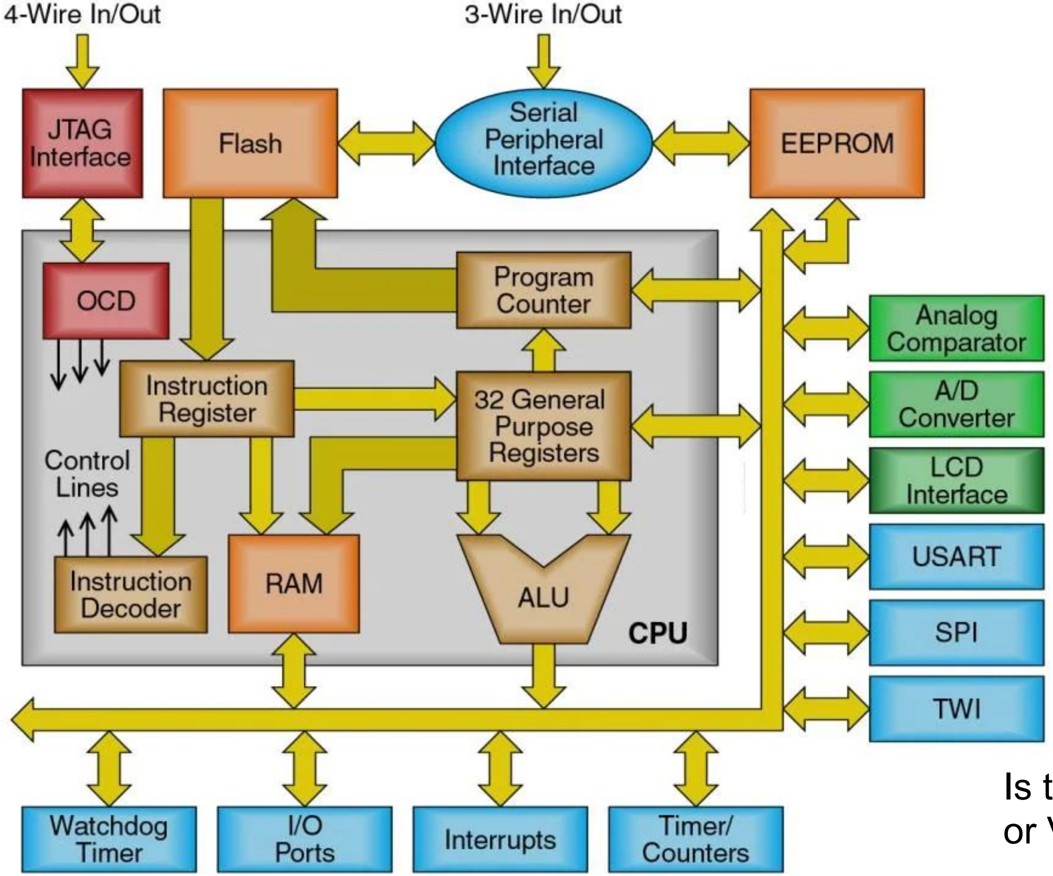
Microchip PIC16F628A PIC architecture



Is this Harvard
or Von Neumann ?

Microchip/Atmel Atmega8 AVR architecture





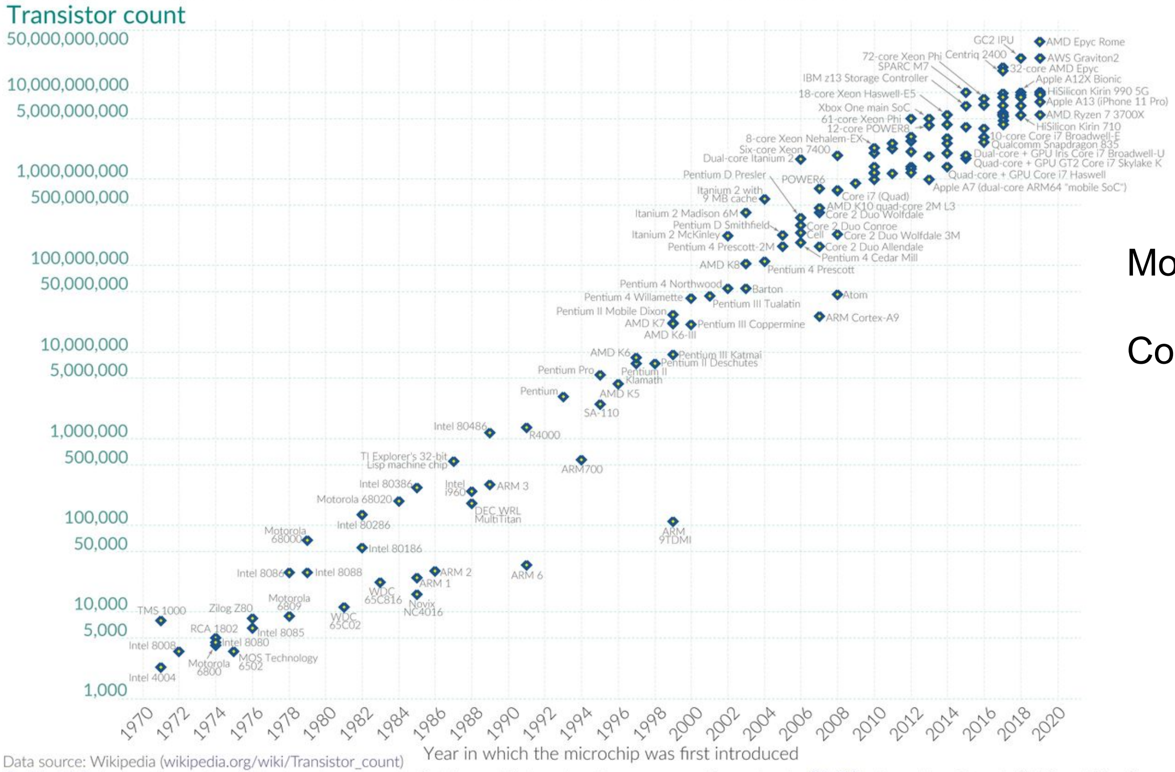
Is this Harvard
or Von Neumann ?

https://www.arrow.com/en/research-and-events/articles/avr-microcontrollers-for-high-performance-and-power-efficient-8-bit-processing

Transistor count & Moore’s Law

Moore’s Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.



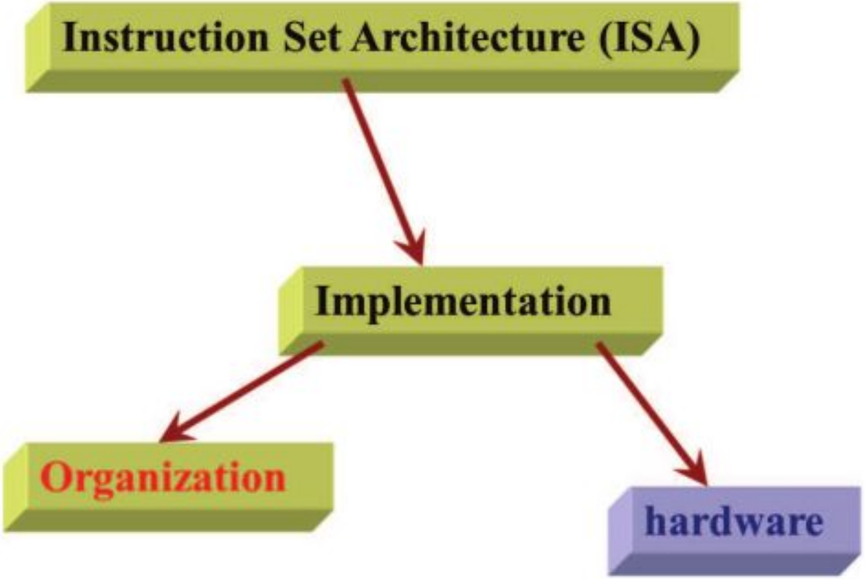
Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world’s largest problems.

Modern CPUs: > 1,000,000,000 transistors

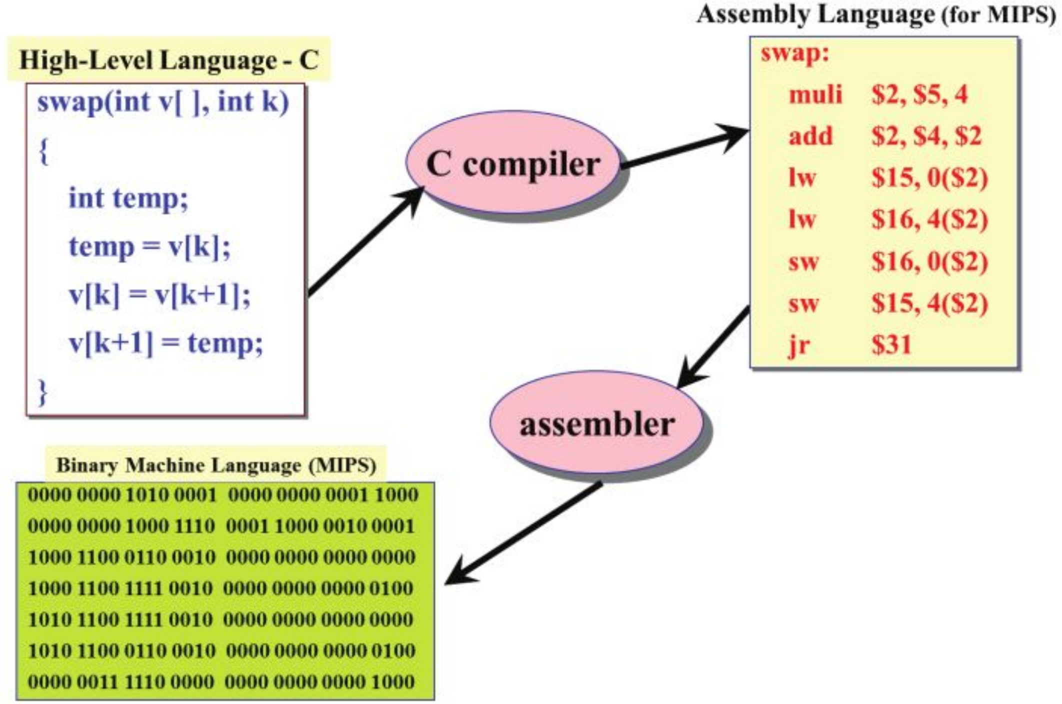
Common uC’s: <100,000 transistors

Architecting computing machines



- A program is specified as Instructions described via an ISA
- The ISA is then implemented as a computing machine
- Organisation: high level aspects, bus design, memory system ...

Compiling to ISAs



High-Level Language - C

```
swap(int v[ ], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Assembly Language (for MIPS)

```
swap:
   muli  $2, $5, 4
   add   $2, $4, $2
   lw    $15, 0($2)
   lw    $16, 4($2)
   sw    $16, 0($2)
   sw    $15, 4($2)
   jr    $31
```

Binary Machine Language (MIPS)

```
0000 0000 1010 0001  0000 0000 0001 1000
0000 0000 1000 1110  0001 1000 0010 0001
1000 1100 0110 0010  0000 0000 0000 0000
1000 1100 1111 0010  0000 0000 0000 0100
1010 1100 1111 0010  0000 0000 0000 0000
1010 1100 0110 0010  0000 0000 0000 0100
0000 0011 1110 0000  0000 0000 0000 1000
```

- ISAs abstract away underlying hardware, enabling *program environments*
- As such, we design an architecture FOR an ISA **(The ISA comes first!)**

Instruction Set Architectures (ISAs)

- ISAs are the interface between hardware and software
- There are many different existing ISAs
  - 80x86, i386, AMD64, ARM, Itanium, AVR, PIC, PowerPC, MIPS, SPARC, HP, RISC-V...
- ISAs may have many different implementations
  - These can have different memory sizes, speeds, costs ....
  - E.g. both Intel and AMD make compatible processors implementing the same ISAs
  - E.g. many many manufacturers implement ARM-compatible processors
- Sometimes, an ISA can enable or prevent innovation
  - Our computer processors are still largely compatible with computers made two decades ago!

What is specified in an ISA?

For each instruction,

- What operation does the CPU need to perform?
  - Arithmetic/logic operations, data transfer, control transfer, …
- How is the data for the operation provided?
  - Through memory? Through registers? Through a constant in the instruction?
- How should the result be stored?
  - In memory or in a register?

In general, two broad classifications

- RISC (Reduced Instruction Set Computers)
    - Sizes of instructions are all the same
    - Fewer possible instructions each doing a general task
    - Simpler hardware
    - Longer program times (e.g. many instructions required for certain complex operations)
- CISC (Complex Instruction Set Computers)
    - May have differently sized instructions
    - Has many possible instructions which may do application-specific tasks
    - Complex hardware
    - Faster program times (E.g. you may have one given instruction for your exact operation)

Four main ISA types

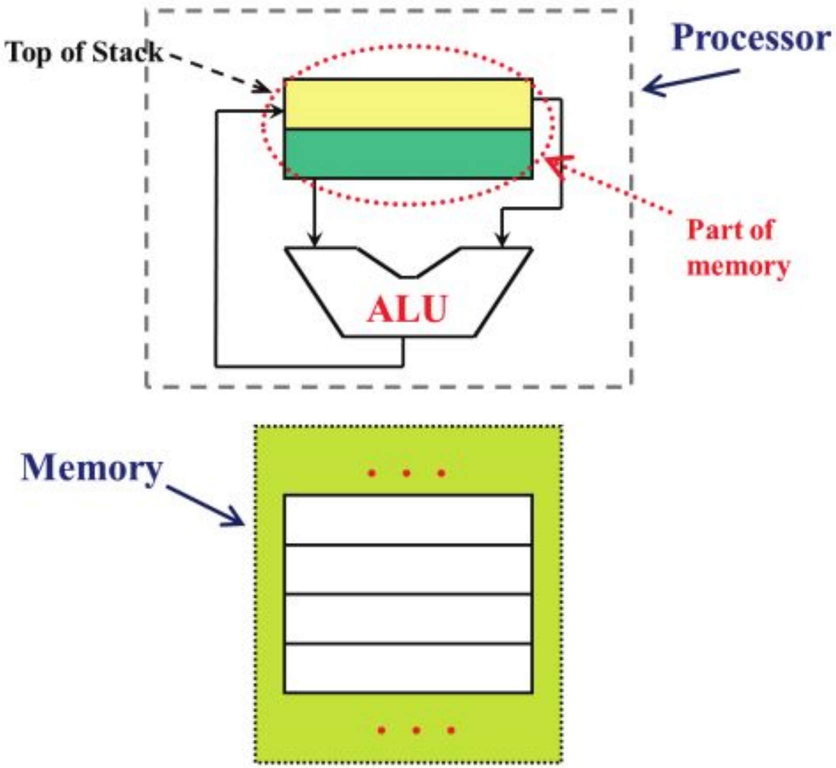
1. Stack-based

2. Accumulator-based

3. Register-memory based

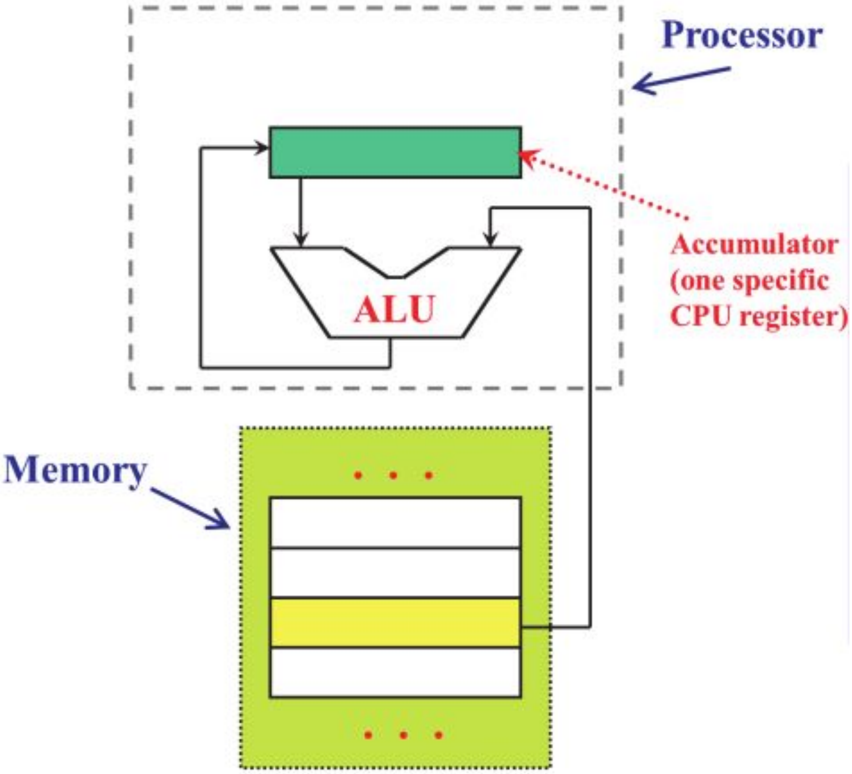
4. Register-register based

Stack-based architectures



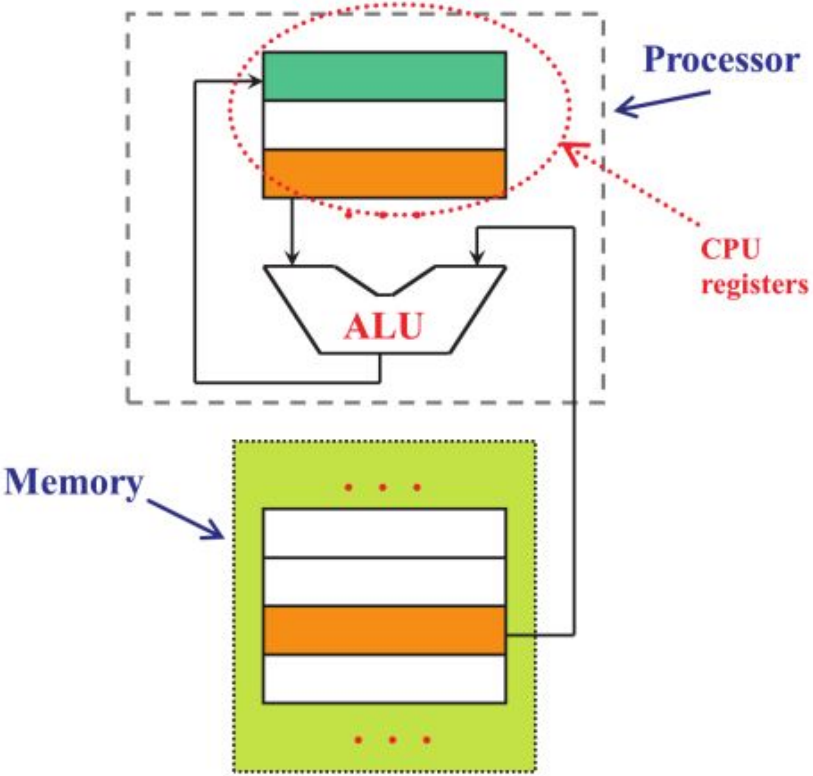
- Stack Pointer for addressing
- Special instructions POP, PUSH
  - POP: read element off the stack
  - PUSH: enter element to the stack
- Operations read from and store in the stack/memory
  - E.g. ADD
  - POPs top two, sum, PUSH result

Accumulator-based architectures



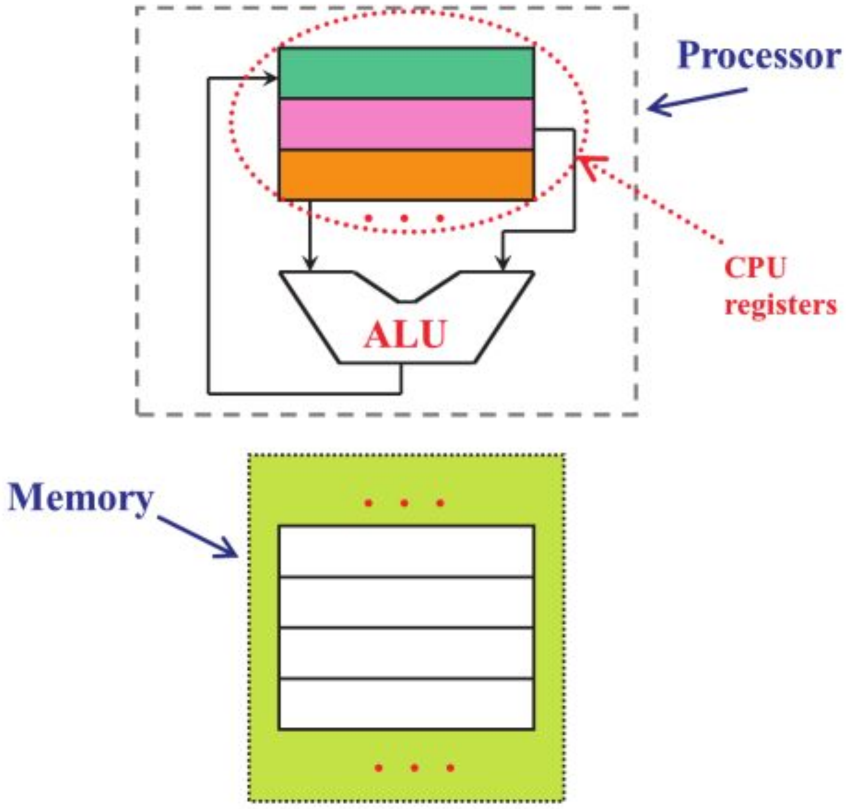
- Special register for computation
- Special instructions LOAD, STORE
- Operations read from memory and work with accumulator
  - E.g. ADD mem_address
  - Take accumulator, sum with memory element, return result to accumulator

Register-memory-based architectures



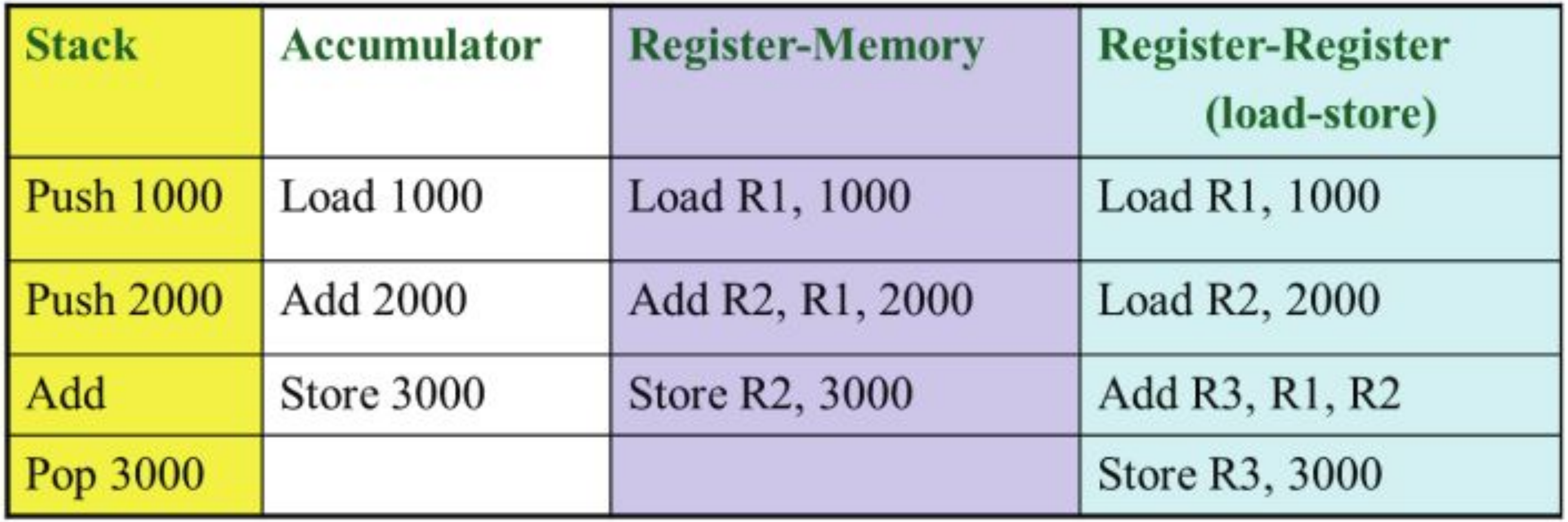
- Many registers for computation
- Special instructions LOAD, STORE
- Operations work over registers and memory
- E.g. ADD reg1, mem_addr, reg2
  - reg2 = reg1+mem_addr

Register-register architectures (load/store)



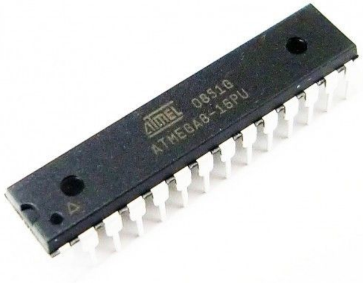
- Many registers for computation
- Special instructions LOAD, STORE
- Operations work only over registers
- E.g. ADD reg1, reg2, reg3
    - reg3 = reg1+reg2

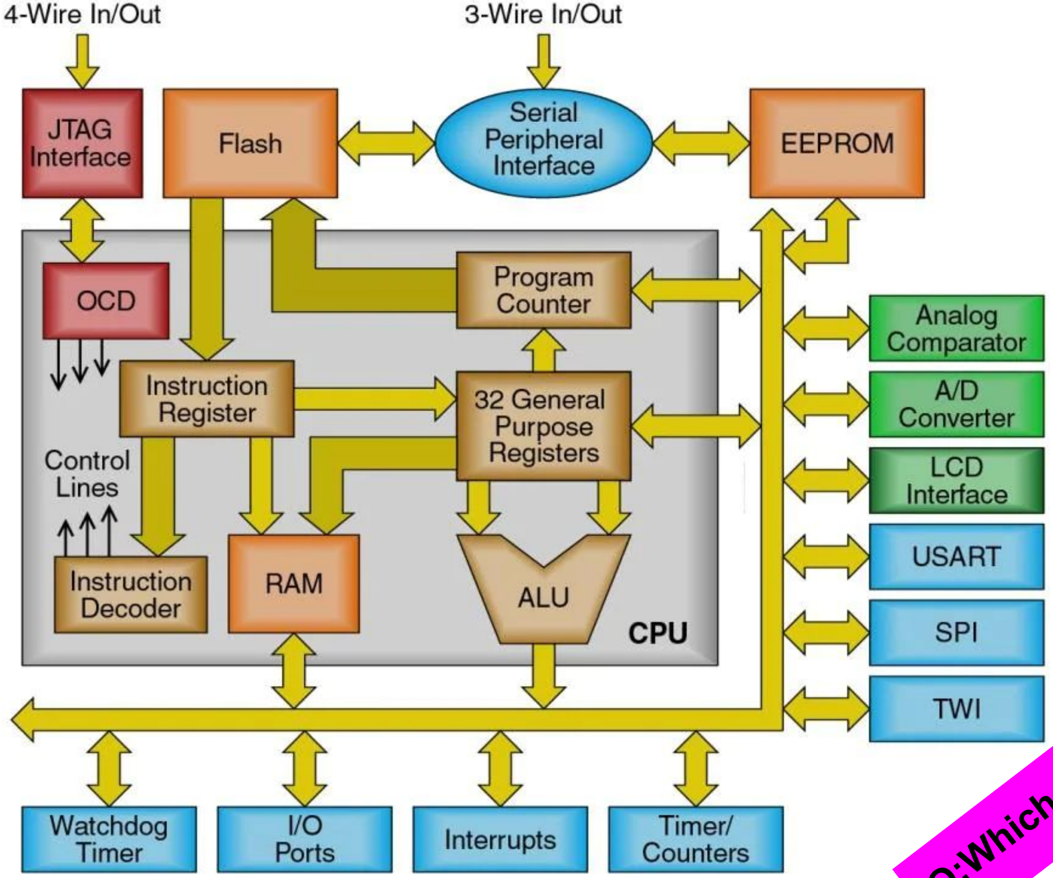
Example ISAs



Stack	Accumulator	Register-Memory	Register-Register (load-store)
Push 1000	Load 1000	Load R1, 1000	Load R1, 1000
Push 2000	Add 2000	Add R2, R1, 2000	Load R2, 2000
Add	Store 3000	Store R2, 3000	Add R3, R1, R2
Pop 3000			Store R3, 3000

Microchip/Atmel Atmega8 AVR architecture

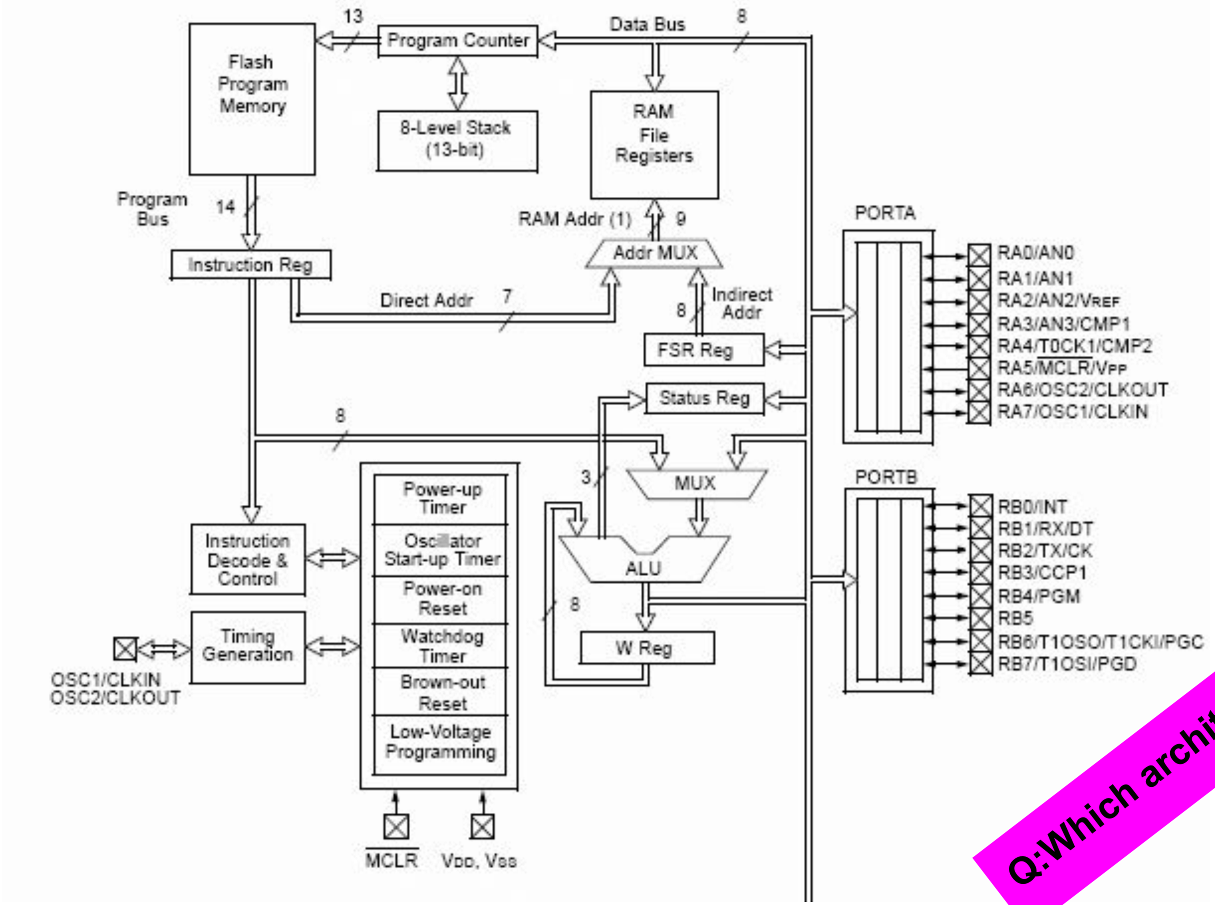




https://www.arrow.com/en/research-and-events/articles/avr-microcontr igh-performance-and-power-efficient-8-bit-processing

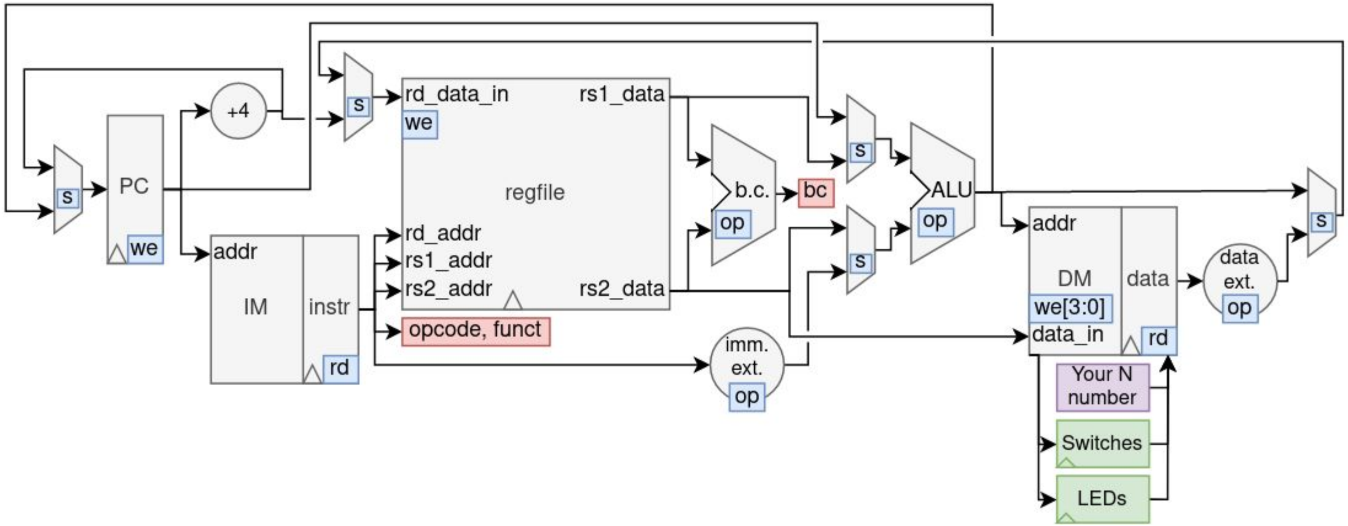
Q:Which architecture type is this?

The PIC16F628A



Q:Which architecture type is this?

The NYU-6463-RV32I

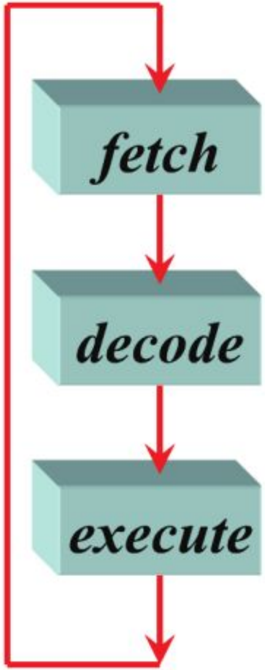


Q:Which architecture type is this?

Summary

- Within each instruction of an ISA there will be an opcode defining
  - Type of operation
  - Zero to “n” operands specifying data source and result destination
- The type of source/destination storage is an easy differentiator for ISAs
  - Operands from/to memory
  - Operands from/to register(s)
  - Operands from/to memory/register(s) both
- Which is best?

Executing an ISA



Fetching

Access memory to get the next instruction: Activate memory read signal, place the right memory address on address bus, read the content of memory pointed by the address

Decoding

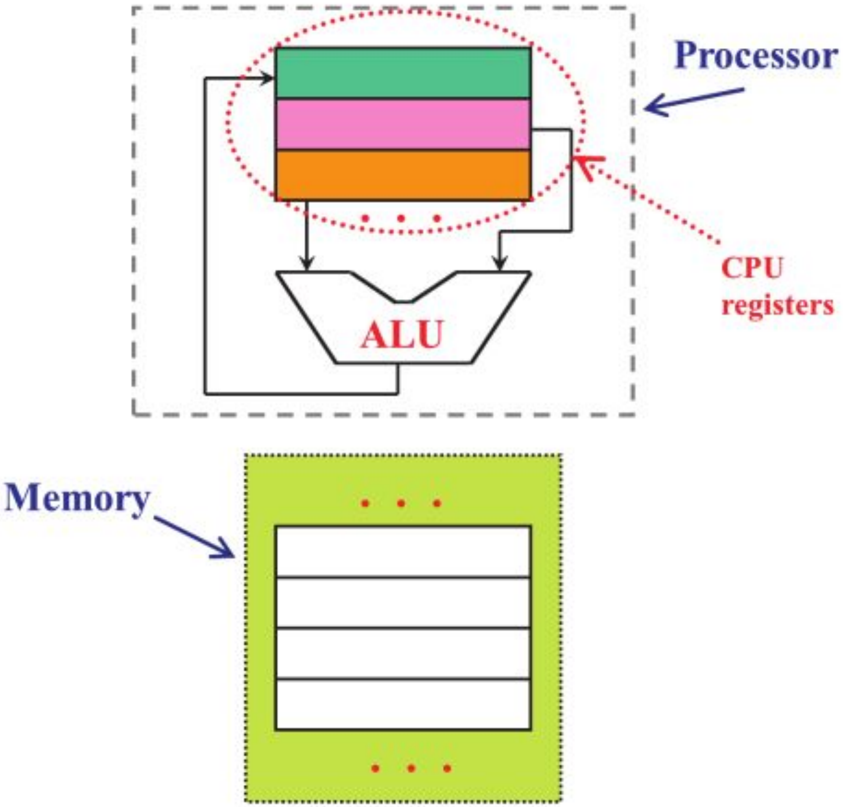
Interpret the bits of the instruction word: to identify the operation and its data (which might be taken from memory or registers)

Execution

Perform that specific operation: Use the processor resources to perform the operation and write the result into memory or register if necessary

CS302 Resources - Morteza Biglari-Abhari

Designing a simple RISC CPU: NYU-6463-RV32I



- Let’s look at the steps required to actually build one of these CPUs using our digital design knowledge

- We’ll need to specify many different types of instruction, and as we do this we’ll see how we build up a block diagram CPU

What sort of instructions are needed?

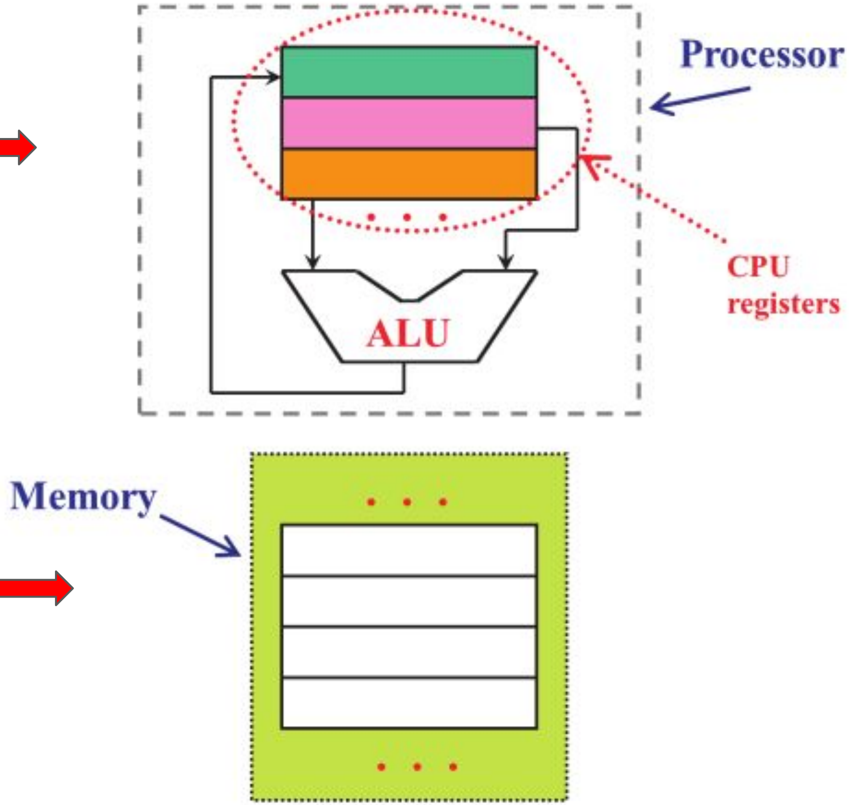
- Arithmetic
  - Addition, subtraction, maybe multiplication, division, etc
- Logical
  - AND, OR, XOR, left shift, right shift, etc
- Memory accesses
  - Load, store, move
- Control transfers
  - Conditional and unconditional branch
- Special-purpose
  - Interrupts, power control, frequency control … we consider as needed

A bit of nomenclature

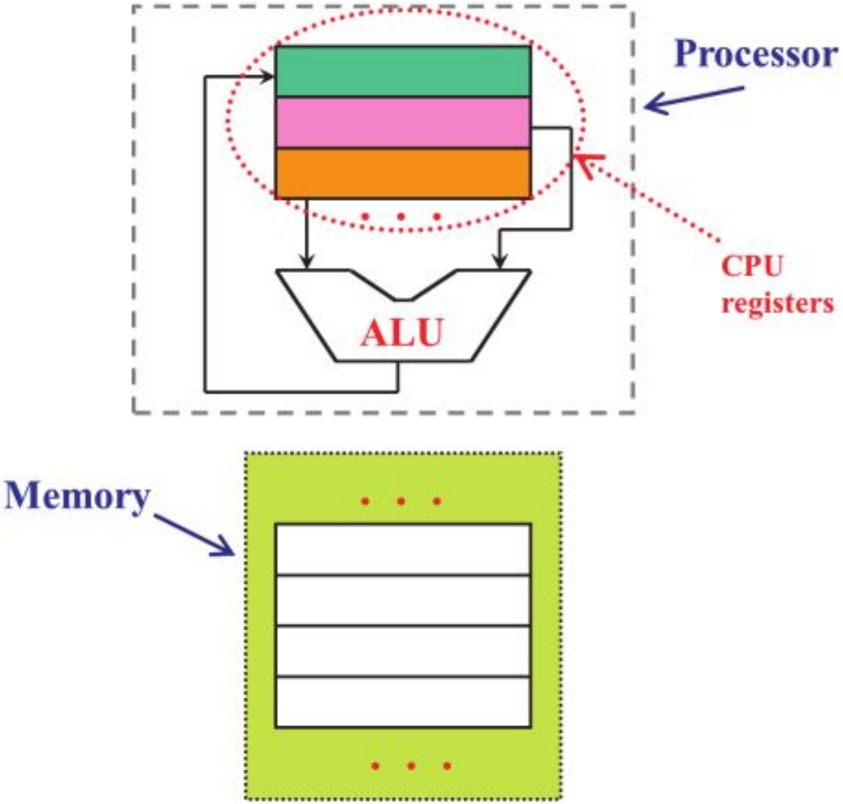
Let’s call this the “register file”

Let’s call this the Data Memory

A location might be DM[3] for the third value



An example operation



Let x1 (register 1) = 1

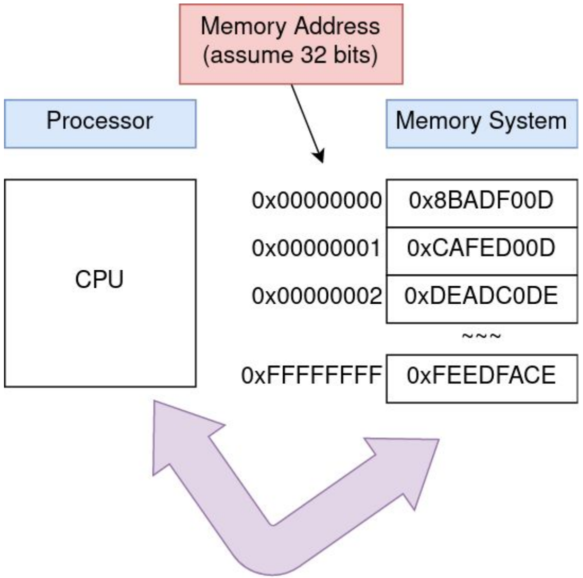
Let DM = {5,6,5,4}

Instructions:

1. Load x2 from DM[1]
2. Add x1 and x2 and store in x3
3. Store x3 to DM[2]

What is value of:
X1 ?
x2 ?
DM ?

Let’s talk memory



Memory Address (assume 32 bits)

Processor

Memory System

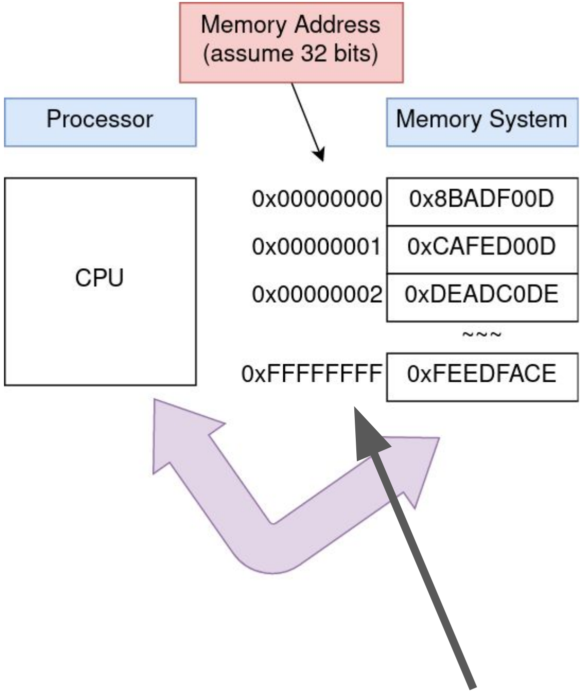
CPU

0x00000000	0x8BADF00D
0x00000001	0xCAFED00D
0x00000002	0xDEADC0DE
	~~
0xFFFFFFFF	0xFEEDFACE

- Memory is a large 1-d array
  - Fixed width (e.g. 32 bits) and length
- A memory address is an index to the array
- Index may point to word of memory
- Accessing larger/smaller amounts (8 bits, 64 bits) requires more work

Memory access time is the amount of time required to read (usually one word) or write data (usually one word) from/to memory.

Let’s talk memory: Byte addressing



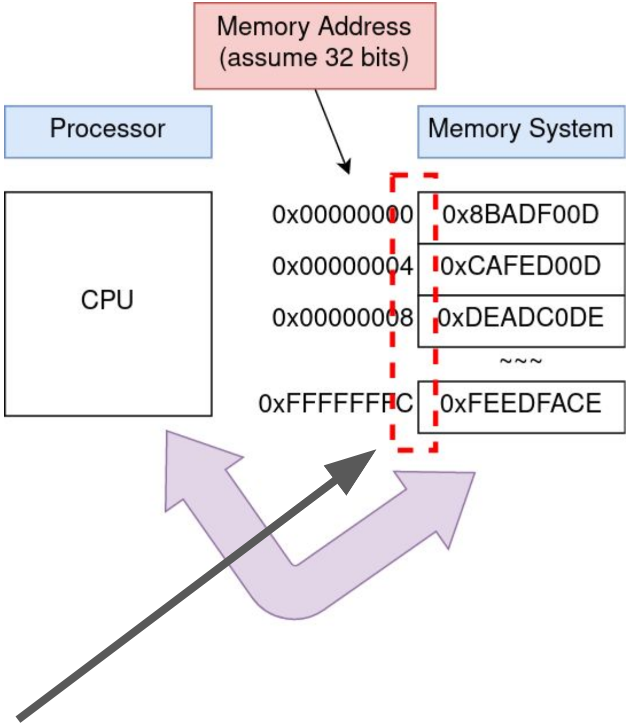
Memory Address (assume 32 bits)

Processor

Memory System

CPU

0x00000000	0x8BADF00D
0x00000001	0xCAFED00D
0x00000002	0xDEADC0DE
	~~
0xFFFFFFFF	0xFEEDFACE



Memory Address (assume 32 bits)

Processor

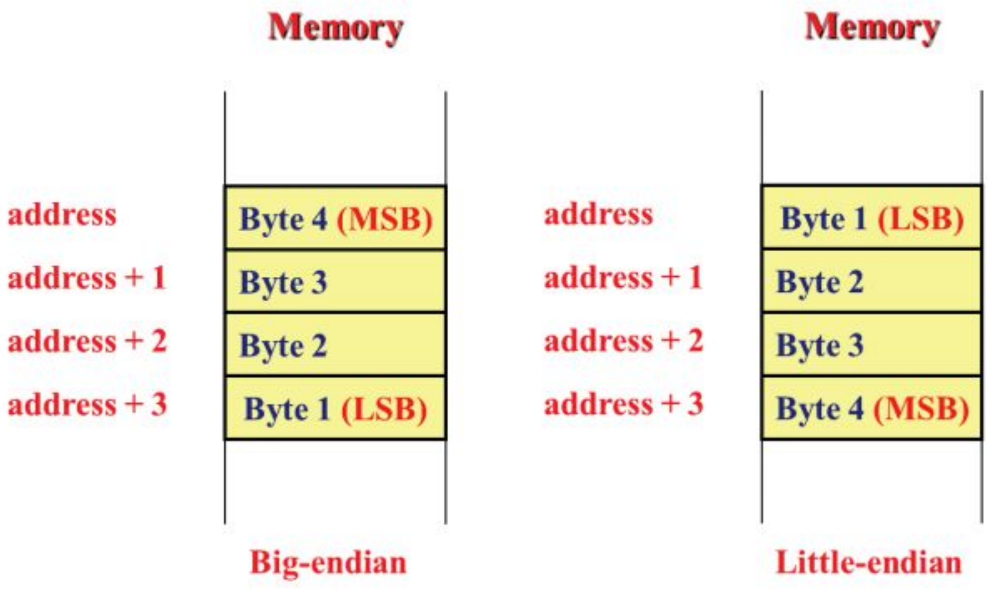
Memory System

CPU

0x00000000	0x8BADF00D
0x00000004	0xCAFED00D
0x00000008	0xDEADC0DE
	~~
0xFFFFFFFC	0xFEEDFACE

Word addressed != Byte addressed

Let’s talk memory: Byte addressing (Endianness)



	Memory		Memory
address	Byte 4 (MSB)	address	Byte 1 (LSB)
address + 1	Byte 3	address + 1	Byte 2
address + 2	Byte 2	address + 2	Byte 3
address + 3	Byte 1 (LSB)	address + 3	Byte 4 (MSB)
	Big-endian		Little-endian

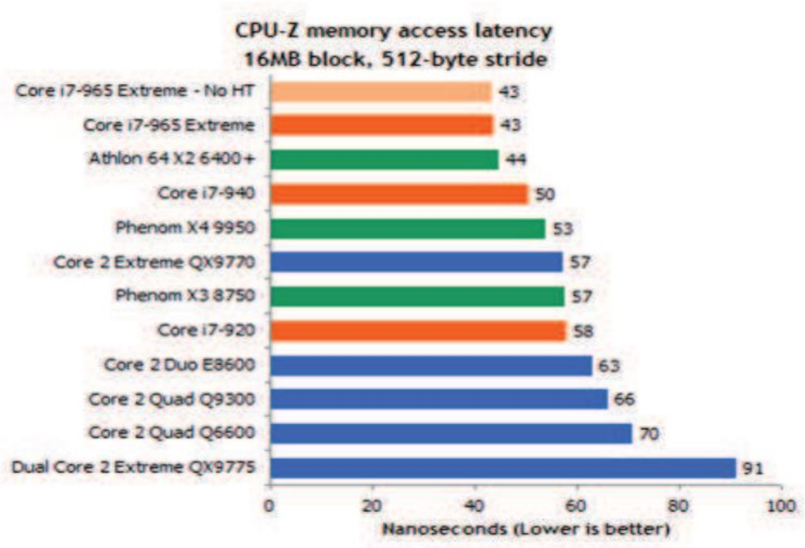
The byte order of memory contents is known as *endianness*

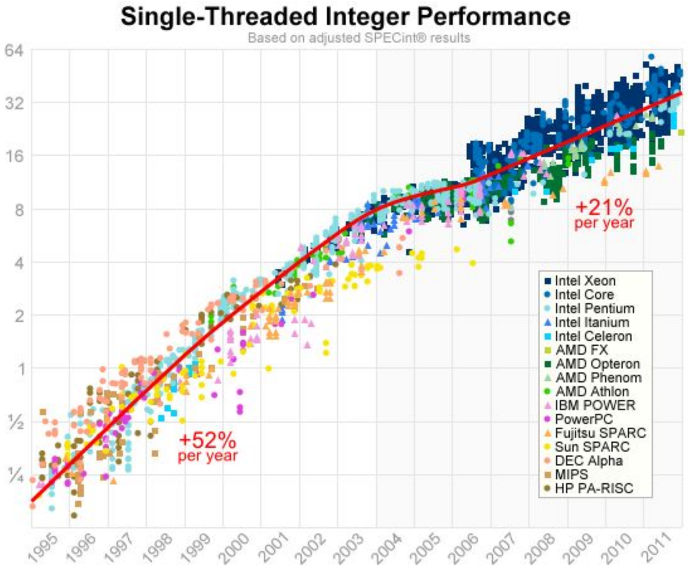
Endianness example

- Consider unsigned hexadecimal number 0x12345678
- This requires at least four bytes to represent.
- In a big-endian ordering they would be [0x12, 0x34, 0x56, 0x78]
- In a little-endian ordering, the bytes would be arranged [0x78, 0x56, 0x34, 0x12]

- **NYU-6463-RV32I is Little-Endian**

Aside: Memory performance

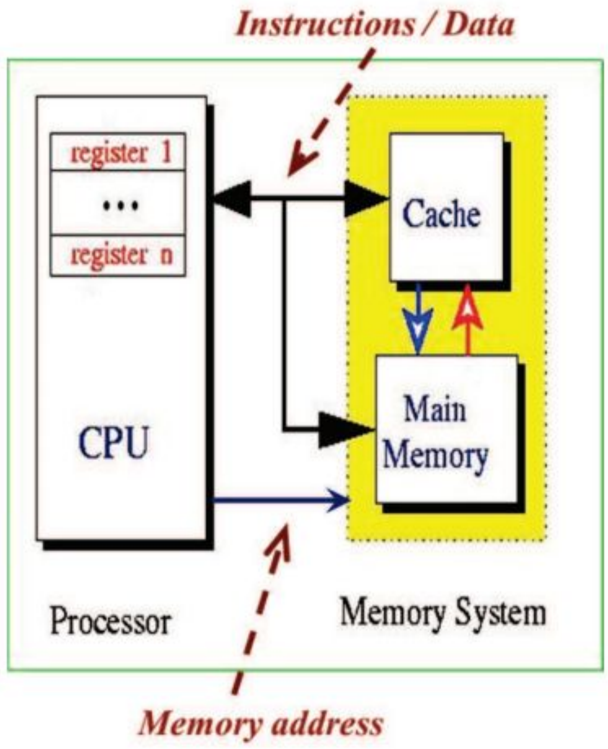




https://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/

- Memory speed has not scaled as well as CPUs
- Solution: memory hierarchies

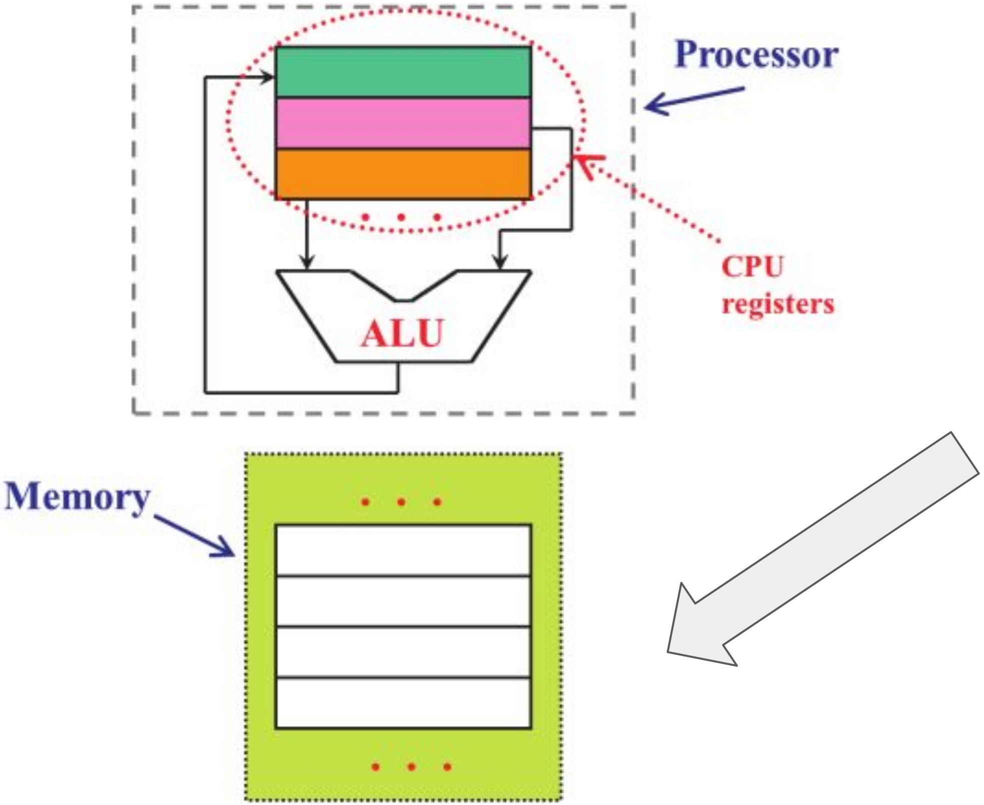
Aside: Memory hierarchies



- Processor registers
  - Access time < nanoseconds
  - 32 or 64 * 32 or 64 bits
- Cache memory
  - Access time several nanoseconds
  - <512KBytes
- Main memory (RAM)
  - Access time in tens of nanoseconds
  - Many GBytes

Our simple design will just use on-chip SRAM and single-cycle memory!

What about system I/O?



Where does system I/O exist in this system?

Actually, special memory locations can represent the system I/O

E.g. a register for UART

A register for a keyboard

Etc...

E.g. The 16F628A memory map

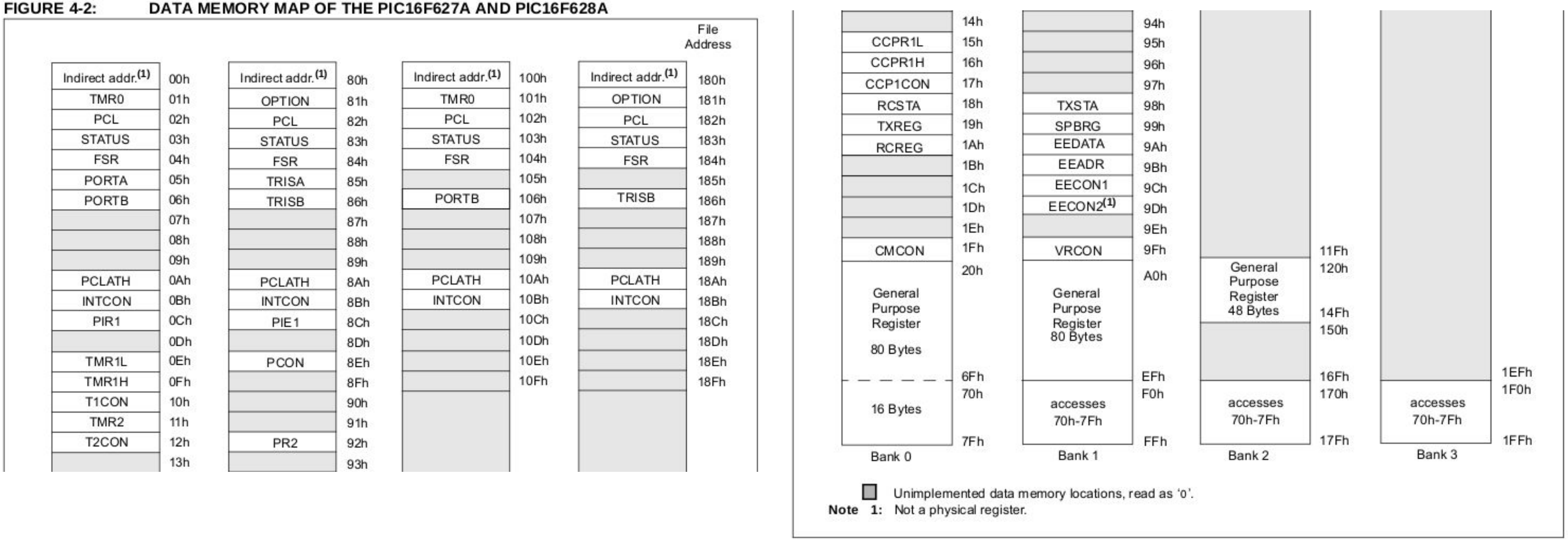


FIGURE 4-2: DATA MEMORY MAP OF THE PIC16F627A AND PIC16F628A

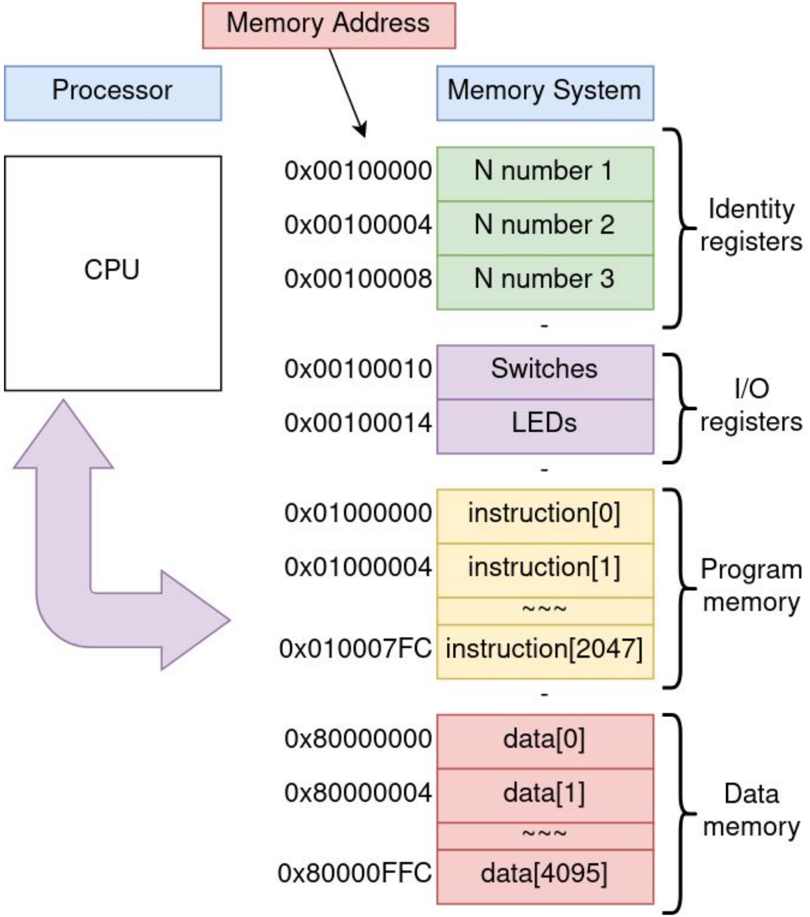
							File Address
Indirect addr.(1)	00h	Indirect addr.(1)	80h	Indirect addr.(1)	100h	Indirect addr.(1)	180h
TMR0	01h	OPTION	81h	TMR0	101h	OPTION	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h	▨	105h	▨	185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
▨	07h	▨	87h	▨	107h	▨	187h
▨	08h	▨	88h	▨	108h	▨	188h
▨	09h	▨	89h	▨	109h	▨	189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	▨	10Ch	▨	18Ch
▨	0Dh	▨	8Dh	▨	10Dh	▨	18Dh
TMR1L	0Eh	PCON	8Eh	▨	10Eh	▨	18Eh
TMR1H	0Fh	▨	8Fh	▨	10Fh	▨	18Fh
T1CON	10h	▨	90h	▨		▨	
TMR2	11h	▨	91h	▨		▨	
T2CON	12h	PR2	92h	▨		▨	
▨	13h	▨	93h	▨		▨	
▨	14h	▨	94h	▨		▨	
CCPR1L	15h	▨	95h	▨		▨	
CCPR1H	16h	▨	96h	▨		▨	
CCP1CON	17h	▨	97h	▨		▨	
RCSTA	18h	TXSTA	98h	▨		▨	
TXREG	19h	SPBRG	99h	▨		▨	
RCREG	1Ah	EEDATA	9Ah	▨		▨	
▨	1Bh	EEADR	9Bh	▨		▨	
▨	1Ch	EECON1	9Ch	▨		▨	
▨	1Dh	EECON2(1)	9Dh	▨		▨	
▨	1Eh	▨	9Eh	▨		▨	
CMCON	1Fh	VRCON	9Fh	▨	11Fh	▨	
General Purpose Register 80 Bytes	20h	General Purpose Register 80 Bytes	A0h	General Purpose Register 48 Bytes	120h 14Fh	▨	
	6Fh		EFh	▨	150h 16Fh	▨	1EFh
16 Bytes	70h 7Fh	accesses 70h-7Fh	F0h FFh	accesses 70h-7Fh	170h 17Fh	accesses 70h-7Fh	1F0h 1FFh
Bank 0		Bank 1		Bank 2		Bank 3	

▨ Unimplemented data memory locations, read as '0'.

Note 1: Not a physical register.

- Most microcontrollers have tons of special memory registers for I/O!

NYU-6463-RV32I memory map



Memory Address

Processor

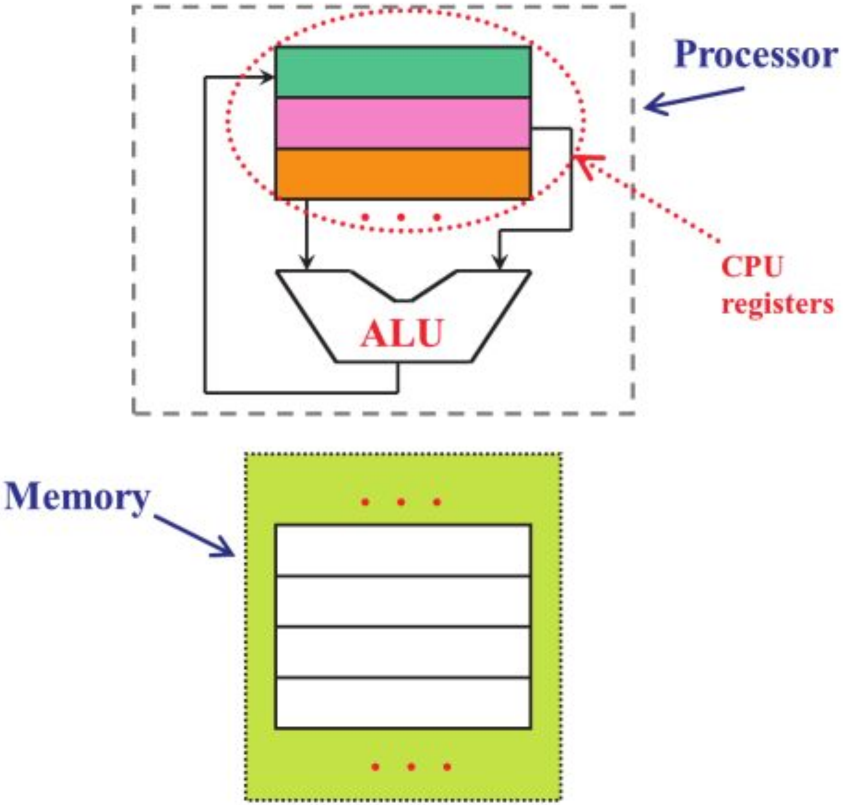
CPU

Memory Address	Memory System	
0x00100000	N number 1	Identity registers
0x00100004	N number 2	
0x00100008	N number 3	
	-	
0x00100010	Switches	I/O registers
0x00100014	LEDs	
	-	
0x01000000	instruction[0]	Program memory
0x01000004	instruction[1]	
	~~	
0x010007FC	instruction[2047]	
	-	
0x80000000	data[0]	Data memory
0x80000004	data[1]	
	~~	
0x80000FFC	data[4095]	

Summary

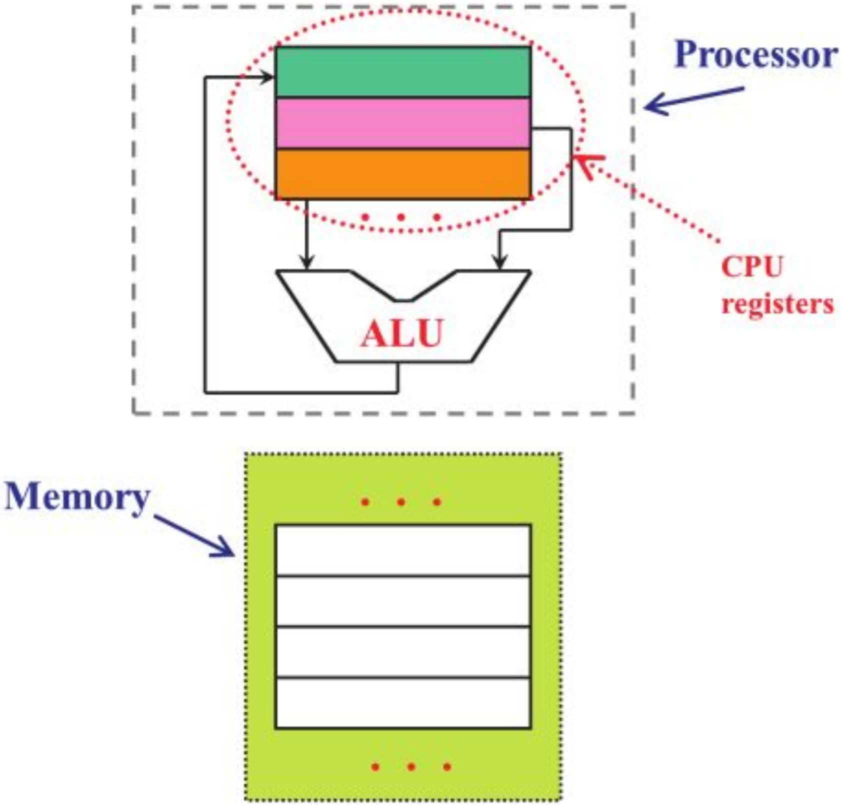
- ISAs define operations upon registers and memory
- Broad classifications on architectures:
    - Stack, Accumulator, Register-Memory, Register-Register
    - RISC v CISC
- Memory is organised as large 1-d arrays
- In complex architectures there are memory hierarchies
- Simple microcontrollers just use simple single-cycle uniform memories
- Often, special registers are used for system I/O
- NYU-6463-RV32I is a register-register RISC architecture

Example operations



- It’s easy to see that many kinds of operation could be encoded here
- E.g. many math/logic operations
- Let’s examine some instructions

Example operations



`addi rd, rs1, imm`

- Add rs+i and store in rd

`add rd, rs1, rs2`

- Add rs1+rs2 and store in rd

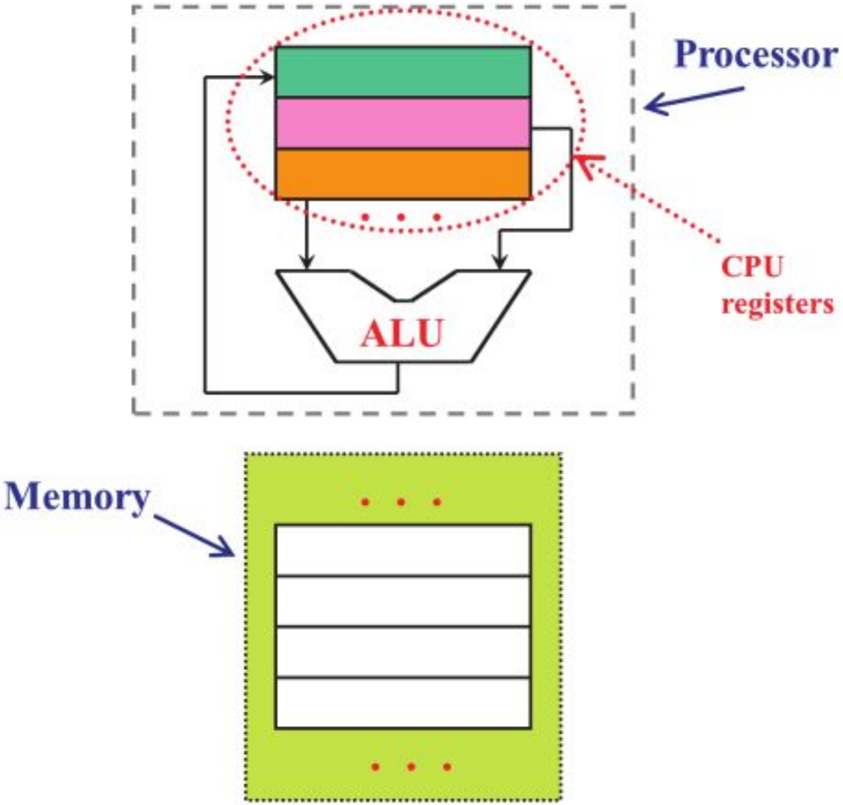
`sw rs2, offset(rs1)`

- Store rs2 in DM[rs1+offset]

`lw rd, offset(rs1)`

- Load r2 from DM[rs1+offset]

What’s the output?

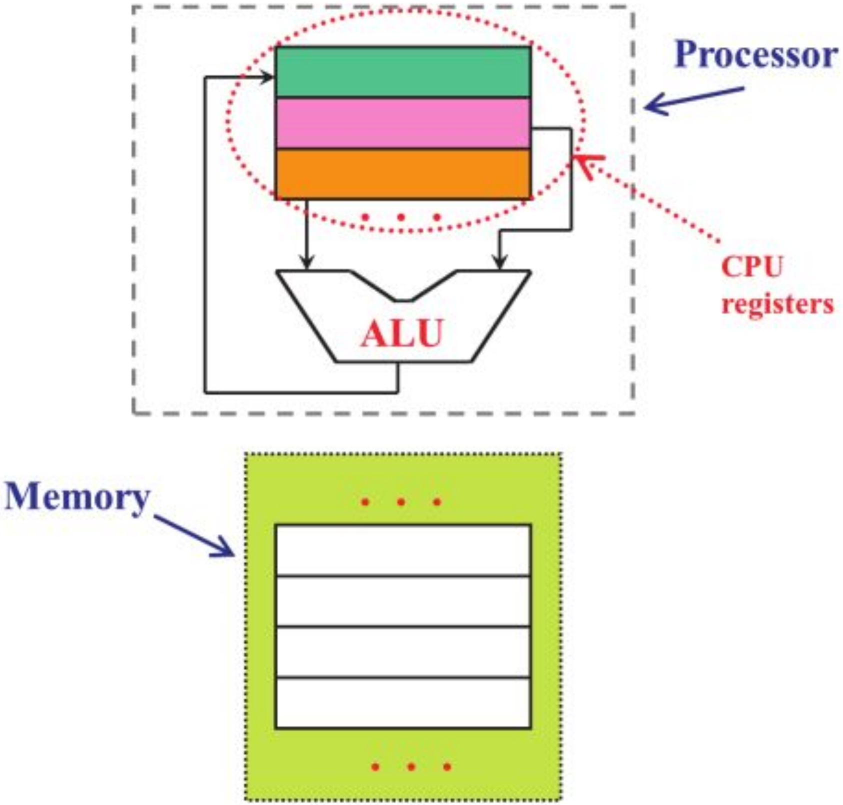


1. ADDI x1, x0, 0x02
2. SW x1, 0(x0)
3. ADDI x1, x1, 0x01
4. SW x1, 4(x0)
5. LW x3, 0(x0)
6. ADD x4, x1, x3

After this program,

Q. What is the value of x0, x1, x2, x3, DM[0], DM[1]?

What’s the output?



1. ADDI x1, x0, 0x02
2. SW x1, 0(x0)
3. ADDI x1, x1, 0x01
4. SW x1, 4(x0)
5. LW x3, 0(x0)
6. ADD x4, x1, x3

After this program,

Q. What is the value of x0, x1, x2, x3, DM[0], DM[1]?

1. x1 = 0x00000002
2. DM[0] = 0x00000002
3. x1 = 0x00000003
4. DM[1] = 0x00000003
5. x3 = 0x00000002
6. x4 = 0x00000005

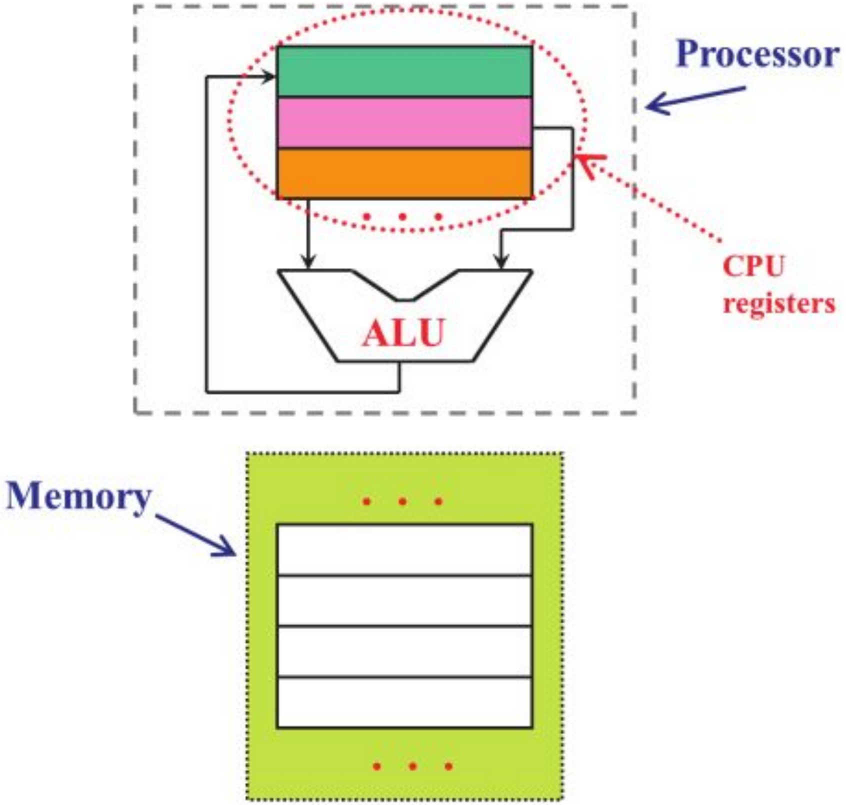
= 0, 0x03, ??, 0x02, 0x02, 0x03

Where does the program come from?

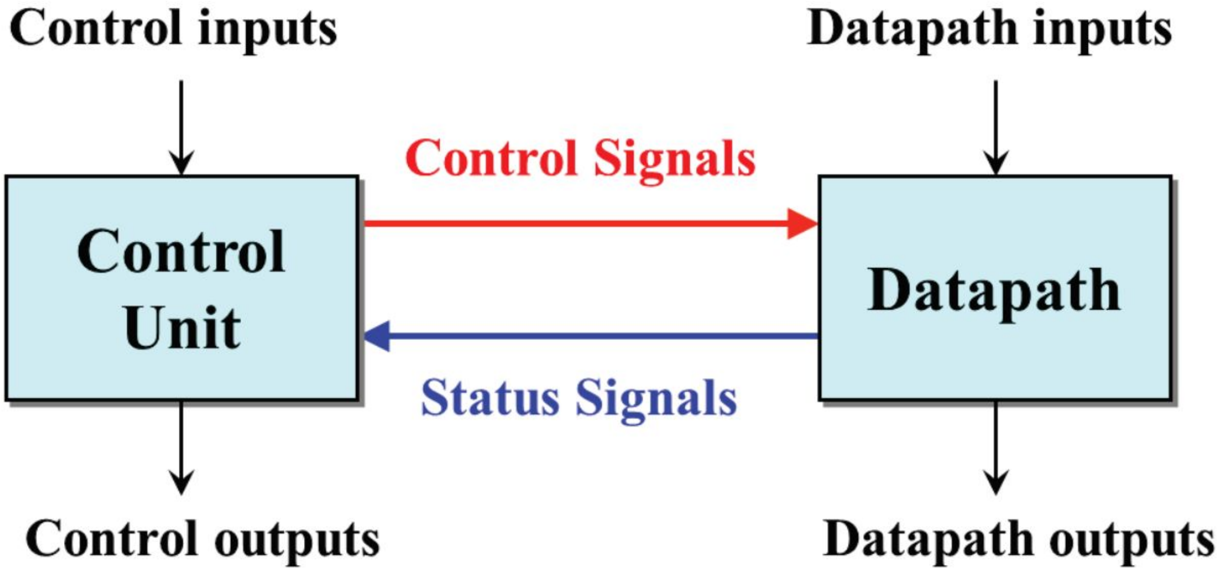
- This block diagram only shows the "datapath"

- It does not show the control unit or the control signals

- Where will they live?



Control and Datapath

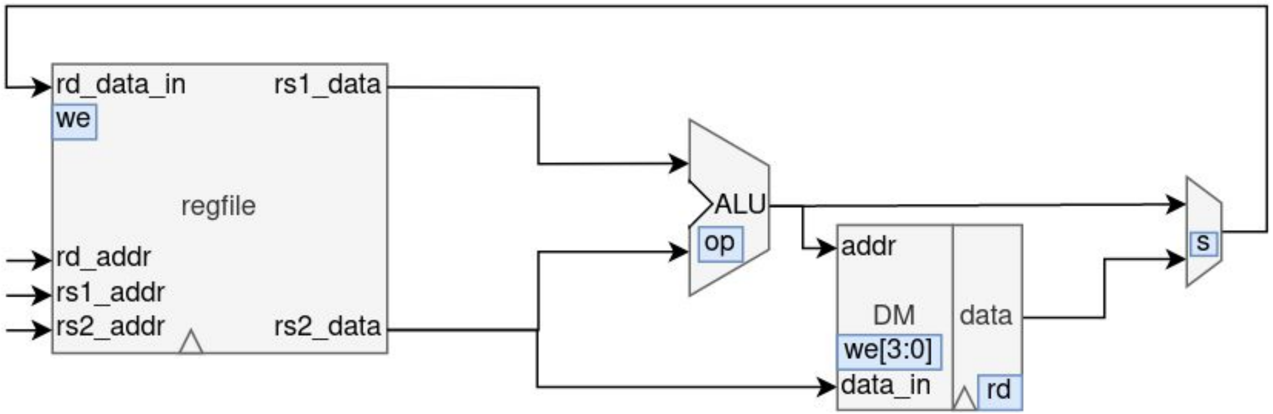


- Datapath performs computations directed by the control unit
- Control unit reads program instructions and status signals to direct datapath
- This might seem similar to you! RC5 FSM?

Let’s examine our datapath components

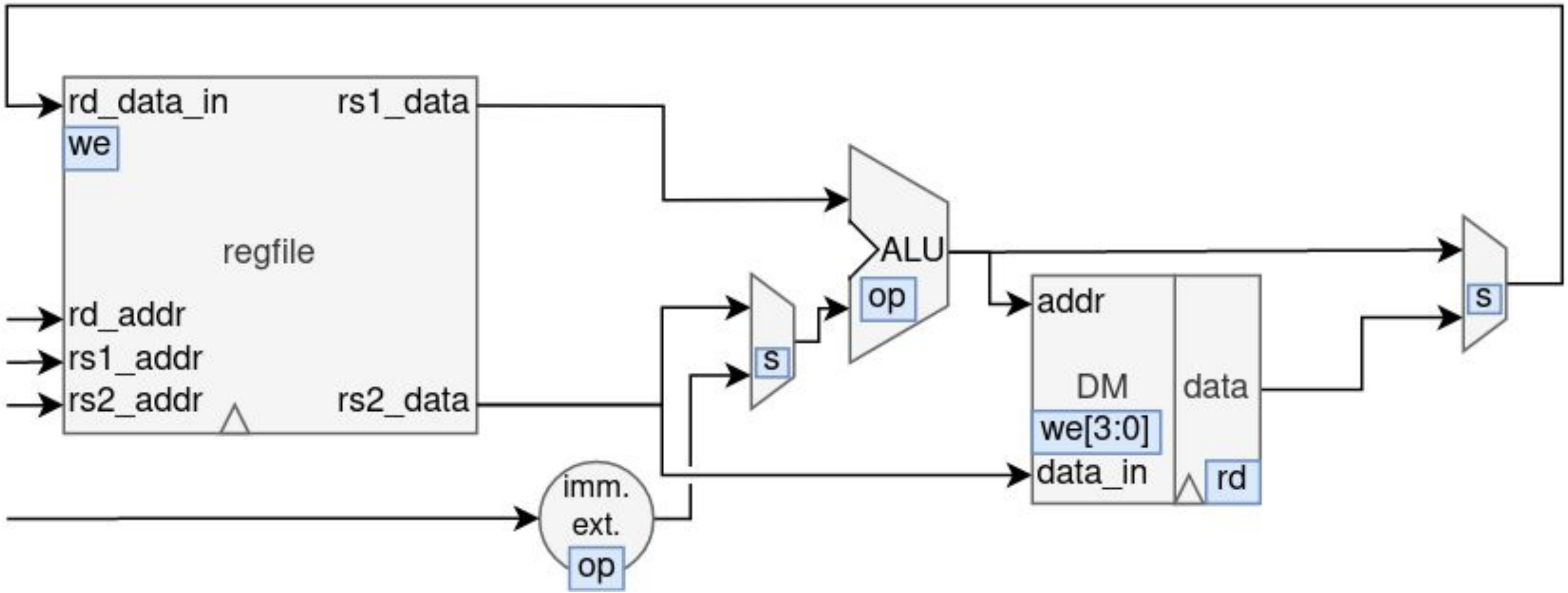
- Register file
- Memory unit
- ALU: performs computation over W and F
- Will we ever need status signals?
    - Usually, these affect control signals
    - For instance “If x > 5”
        - Implement with comparison unit

A start to the datapath



Can it add numbers / (do “math”) ?
Can it store values?
Can it load values?

A start to the datapath

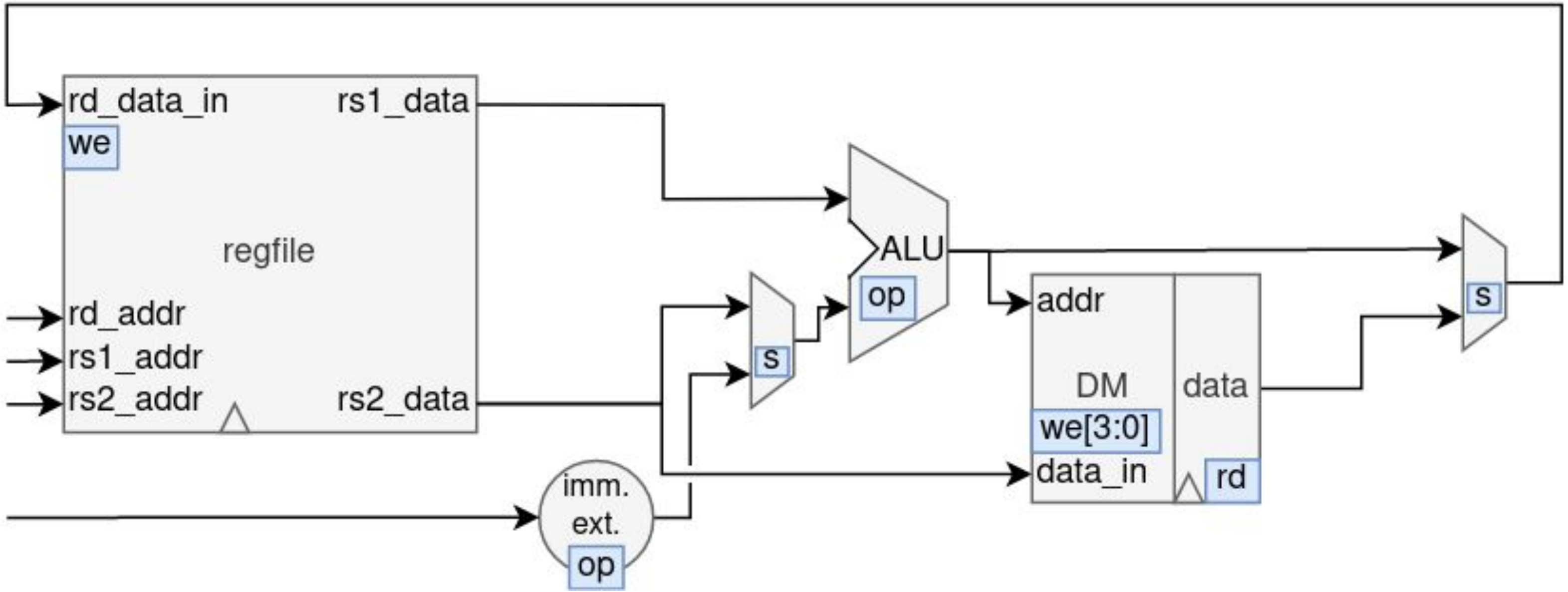


Can it add numbers / (do “math”) ?
Can it store values?
Can it load values?

Example

For each step of the program, what are the signals?

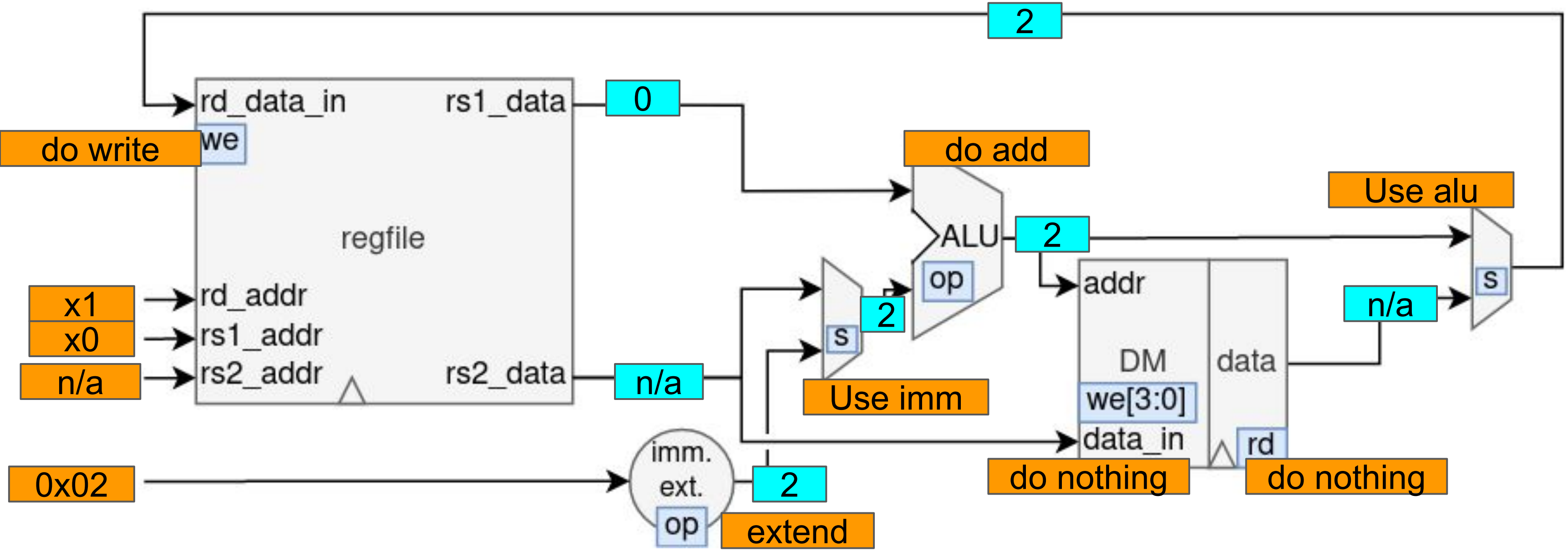
1. **ADDI x1, x0, 0x02**
2. **SW x1, 0(x0)**
3. **ADDI x1, x1, 0x01**
4. **SW x1, 4(x0)**
5. **LW x3, 0(x0)**
6. **ADD x4, x1, x3**



Example

For each step of the program, what are the signals?

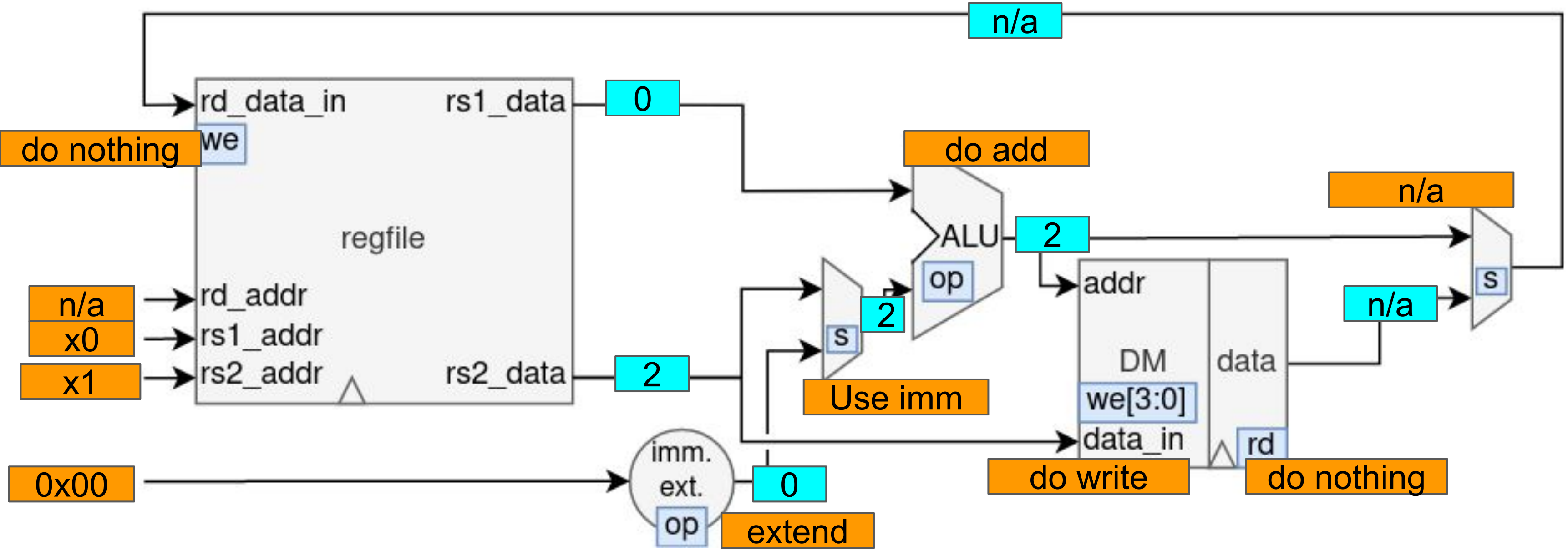
1. **ADDI x1, x0, 0x02**



Example

For each step of the program, what are the signals?

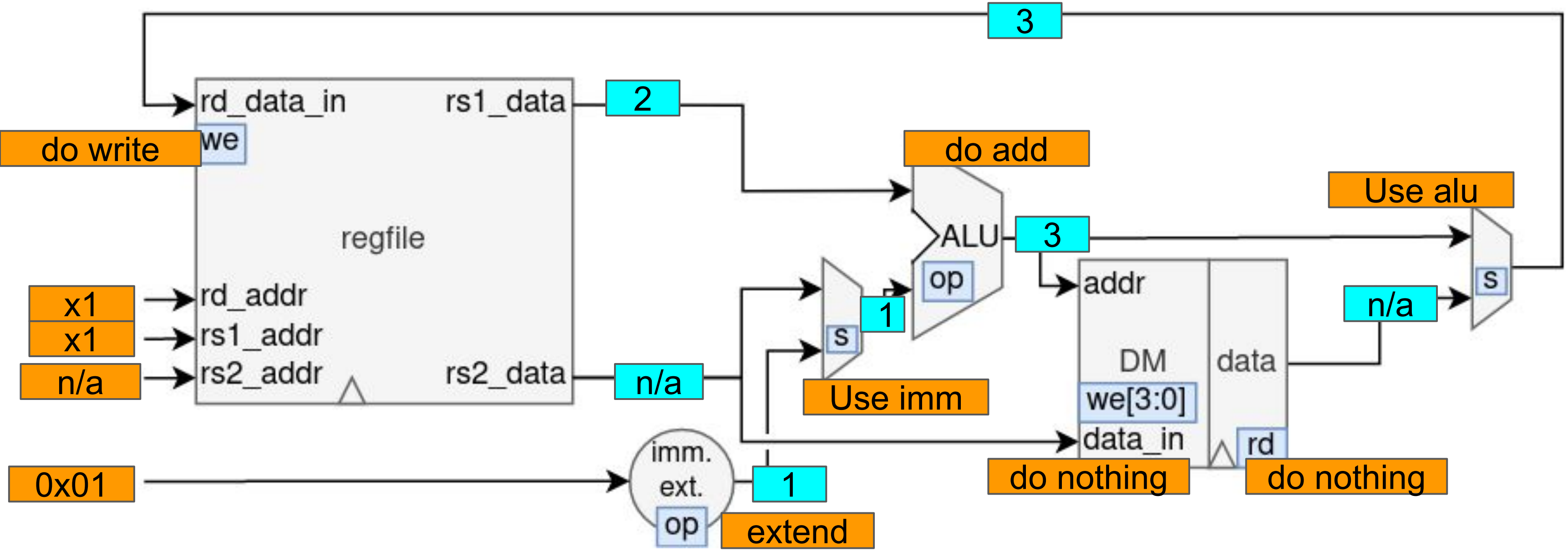
1. **ADDI x1, x0, 0x02**
2. **SW x1, 0(x0)**



Example

For each step of the program, what are the signals?

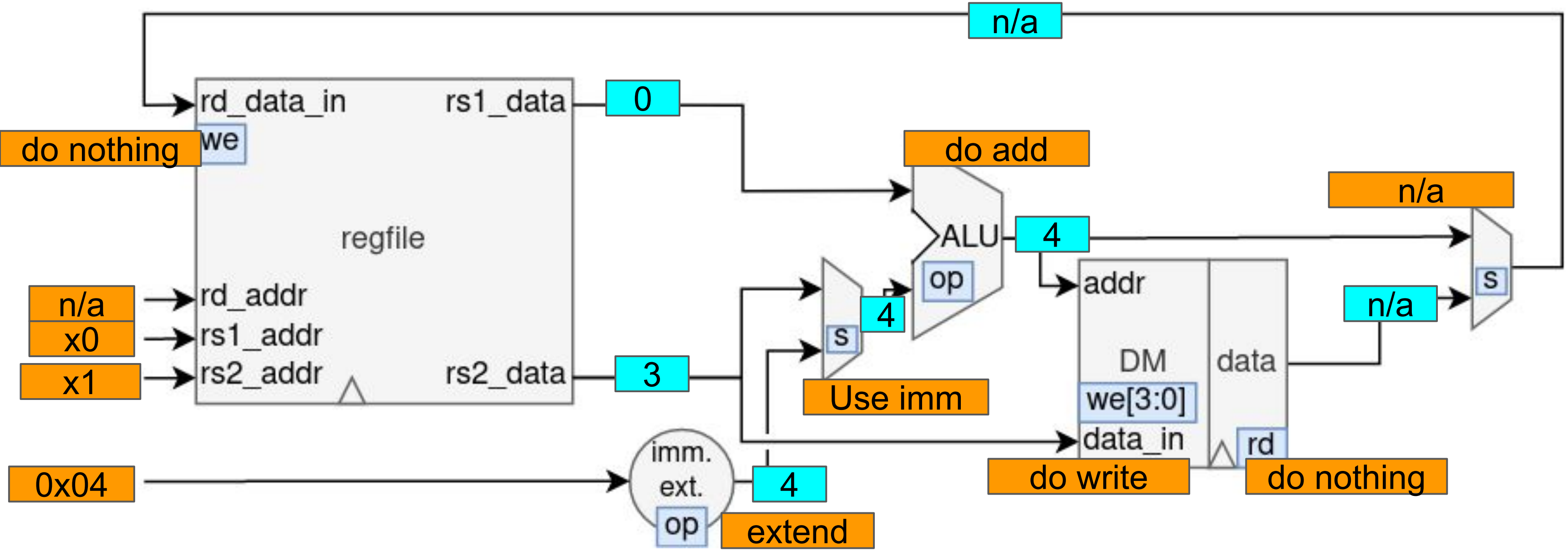
1. **ADDI x1, x0, 0x02**
2. **SW x1, 0(x0)**
3. **ADDI x1, x1, 0x01**



Example

For each step of the program, what are the signals?

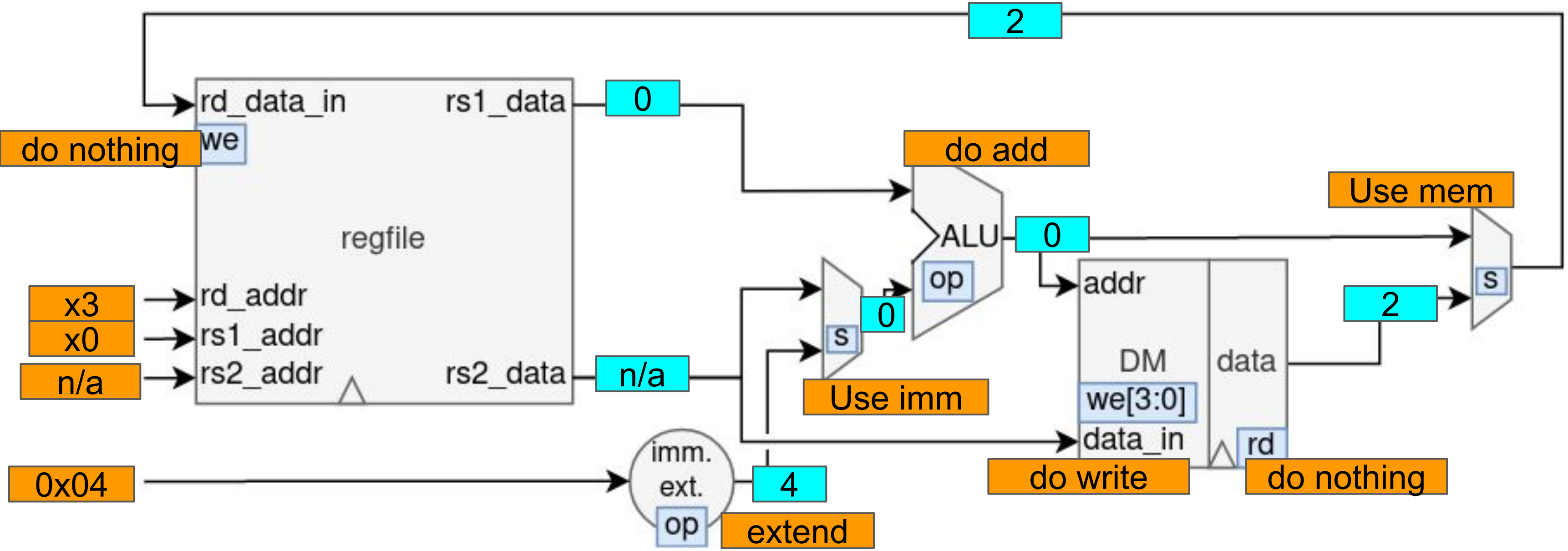
1. **ADDI x1, x0, 0x02**
2. **SW x1, 0(x0)**
3. **ADDI x1, x1, 0x01**
4. **SW x1, 4(x0)**



Example

For each step of the program, what are the signals?

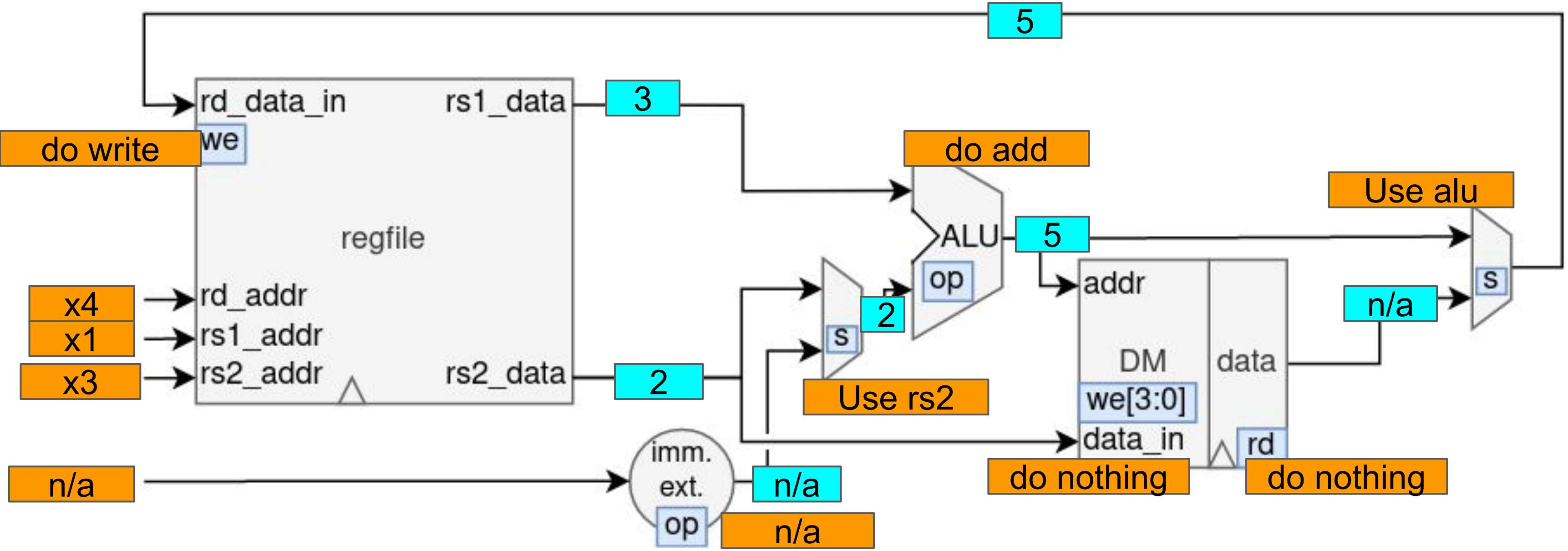
1. **ADDI x1, x0, 0x02**
2. **SW x1, 0(x0)**
3. **ADDI x1, x1, 0x01**
4. **SW x1, 4(x0)**
5. **LW x3, 0(x0)**



Example

For each step of the program, what are the signals?

1. **ADDI x1, x0, 0x02**
2. **SW x1, 0(x0)**
3. **ADDI x1, x1, 0x01**
4. **SW x1, 4(x0)**
5. **LW x3, 0(x0)**
6. **ADD x4, x1, x3**



Summary

- The DATAPATH performs the actual computation provided in a program

- The DATAPATH is driven by control signals

- These control signals need to be provided via a CONTROL UNIT

Decoding instructions

- Instructions are binary encoded with bit fields for each argument

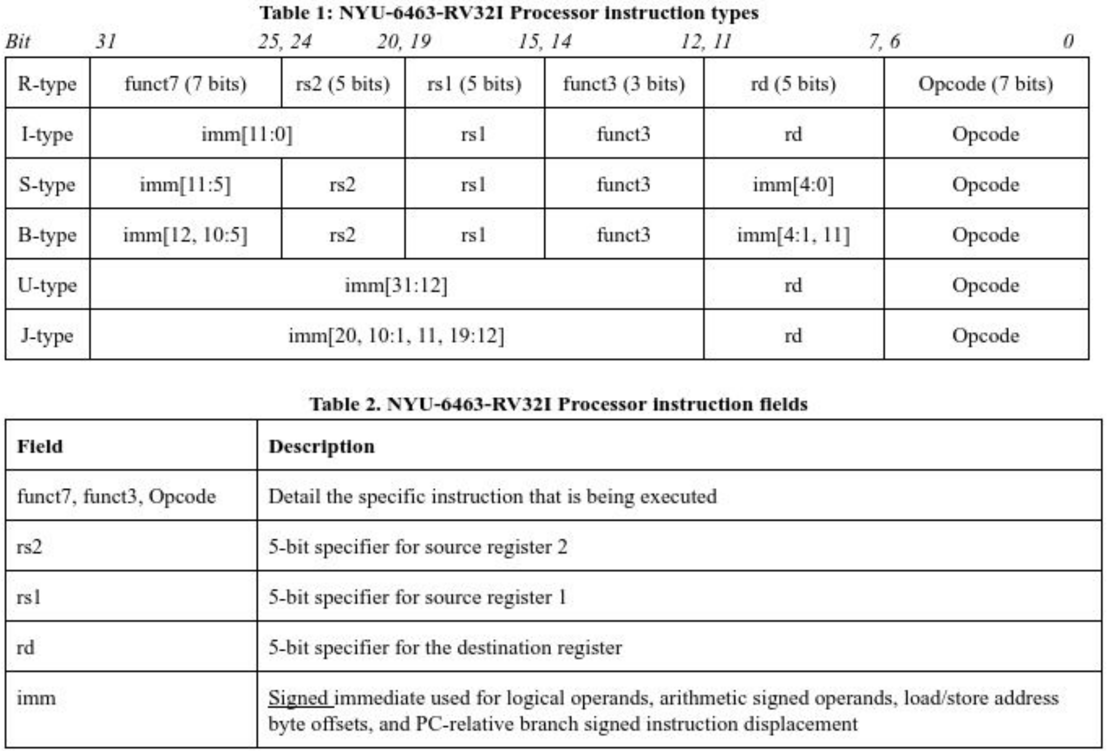


Table 1: NYU-6463-RV32I Processor instruction types

Bit 31 25, 24 20, 19 15, 14 12, 11 7, 6 0

R-type	funct7 (7 bits)	rs2 (5 bits)	rs1 (5 bits)	funct3 (3 bits)	rd (5 bits)	Opcode (7 bits)
I-type	imm[11:0]		rs1	funct3	rd	Opcode
S-type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	Opcode
B-type	imm[12, 10:5]	rs2	rs1	funct3	imm[4:1, 11]	Opcode
U-type	imm[31:12]				rd	Opcode
J-type	imm[20, 10:1, 11, 19:12]				rd	Opcode

Table 2. NYU-6463-RV32I Processor instruction fields

Field	**Description**
funct7, funct3, Opcode	Detail the specific instruction that is being executed
rs2	5-bit specifier for source register 2
rs1	5-bit specifier for source register 1
rd	5-bit specifier for the destination register
imm	Signed immediate used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacement

- Examples

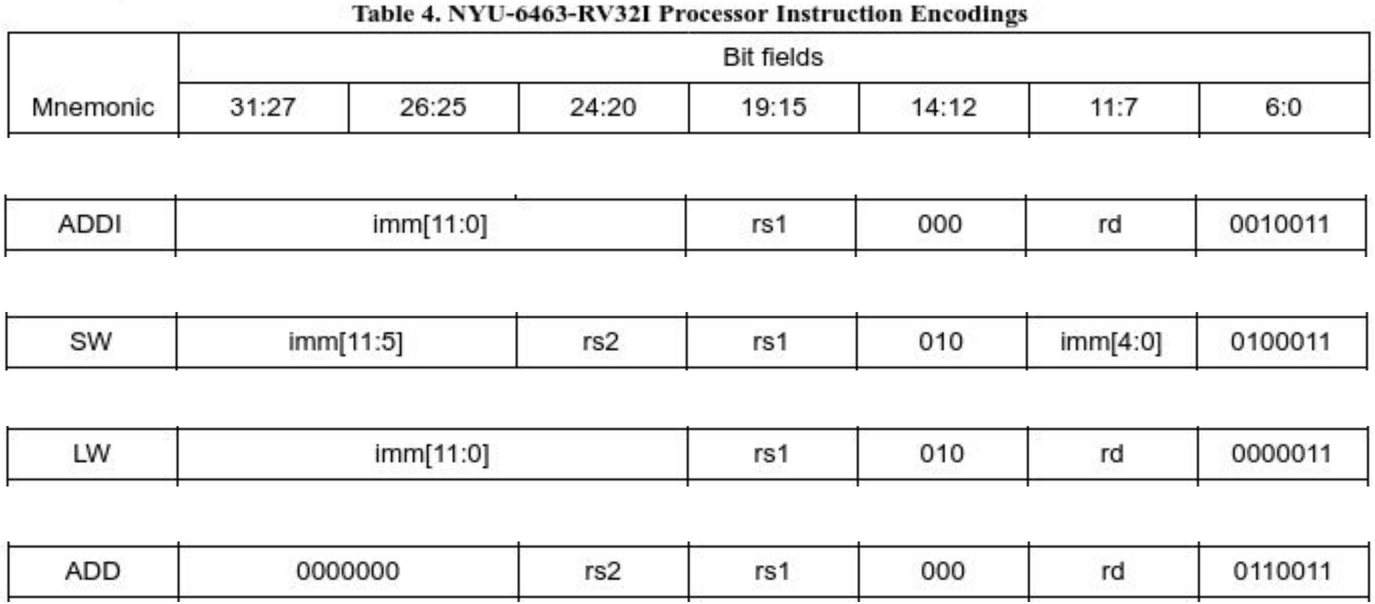
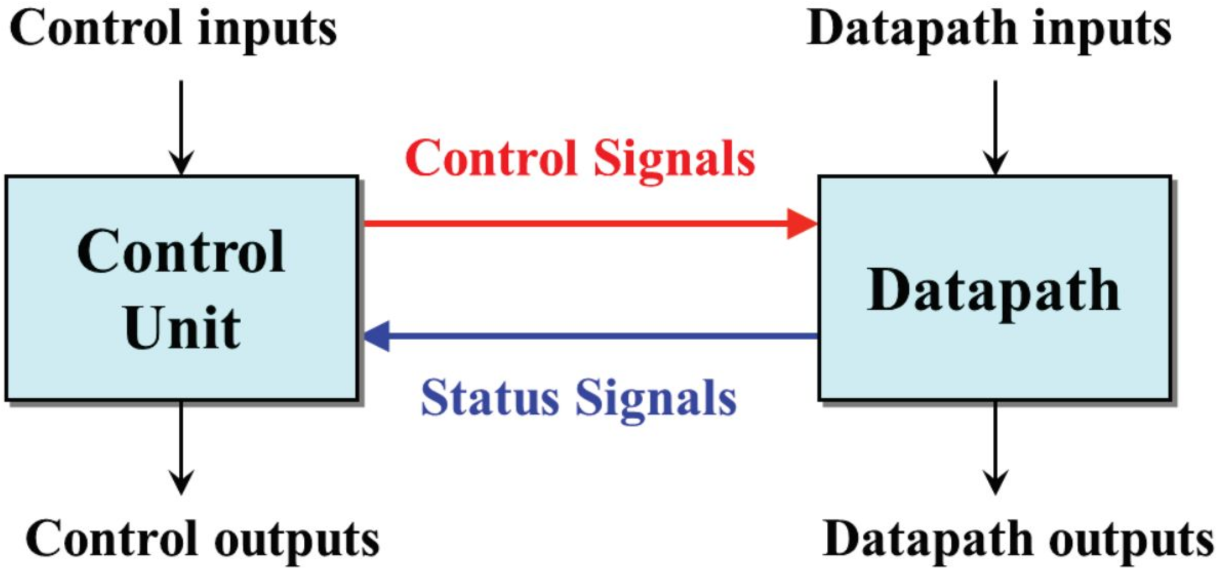


Table 4. NYU-6463-RV32I Processor Instruction Encodings

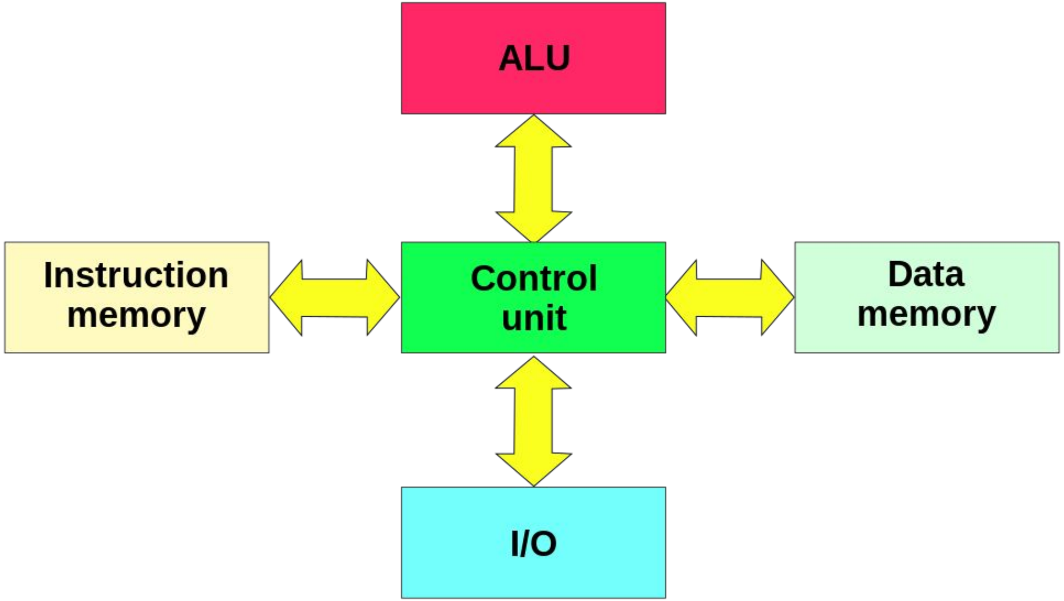
Mnemonic	Bit fields						
	31:27	26:25	24:20	19:15	14:12	11:7	6:0
ADDI	imm[11:0]			rs1	000	rd	0010011
SW	imm[11:5]		rs2	rs1	010	imm[4:0]	0100011
LW	imm[11:0]			rs1	010	rd	0000011
ADD	0000000		rs2	rs1	000	rd	0110011

Designing a control unit



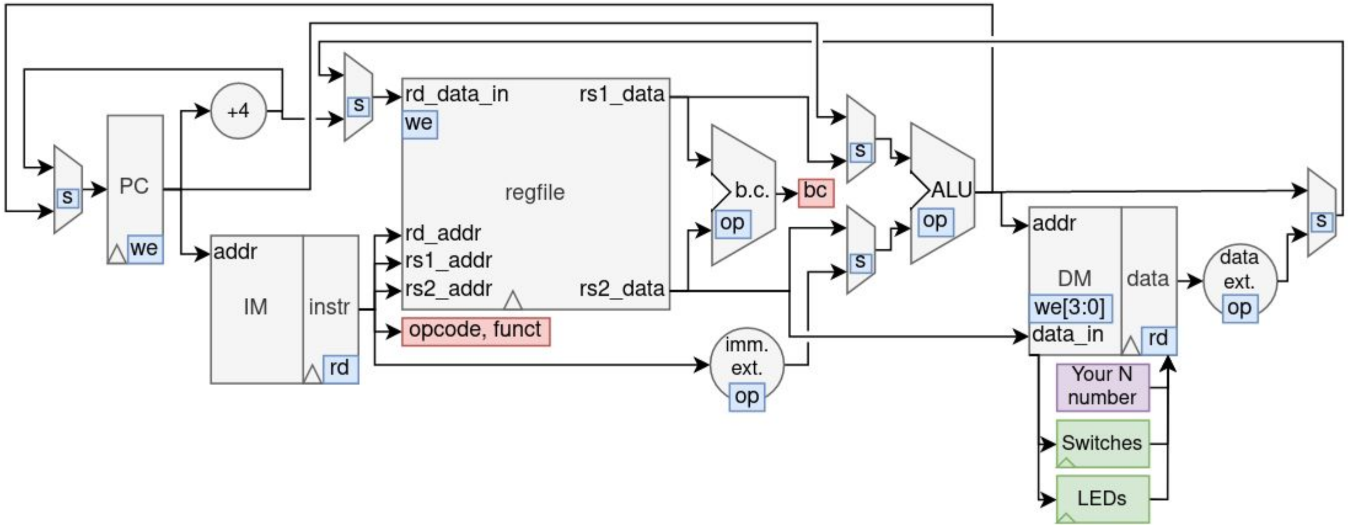
- Datapath performs computations directed by the control unit
- Control unit reads program instructions and status signals to direct datapath
- The Control Unit’s inputs come from the given program

Implementing control logic



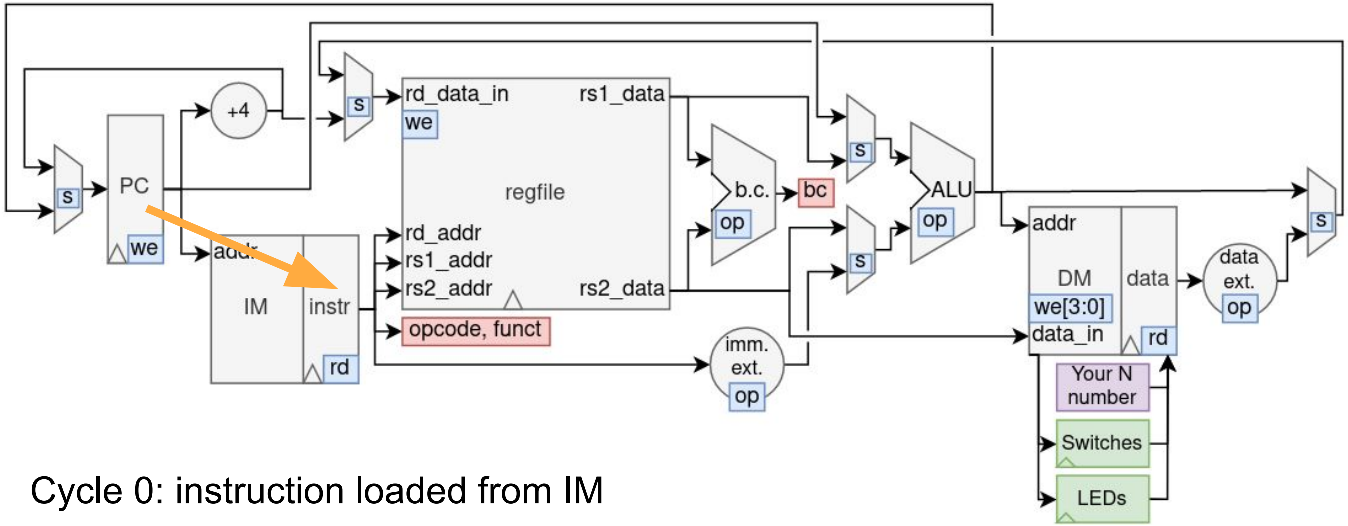
- We’re interested in a Harvard architecture
- That means we need another memory storage unit

Complete datapath - executing in stages



E.g.: **LW x3, 0(x0)**

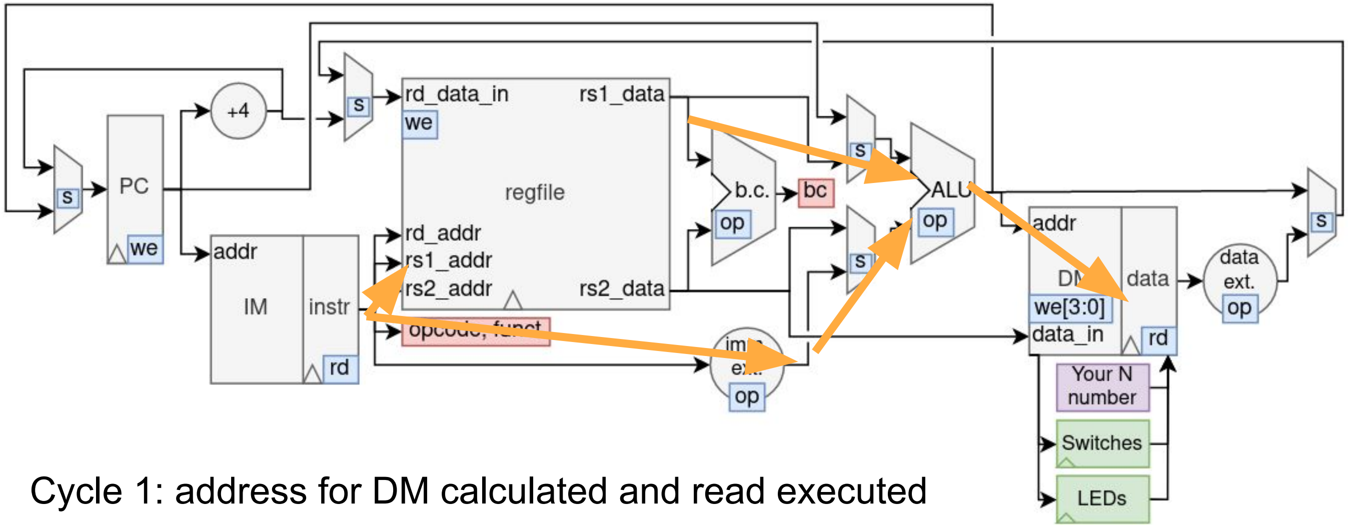
Complete datapath - executing in stages



Cycle 0: instruction loaded from IM

E.g.: **LW x3, 0(x0)**

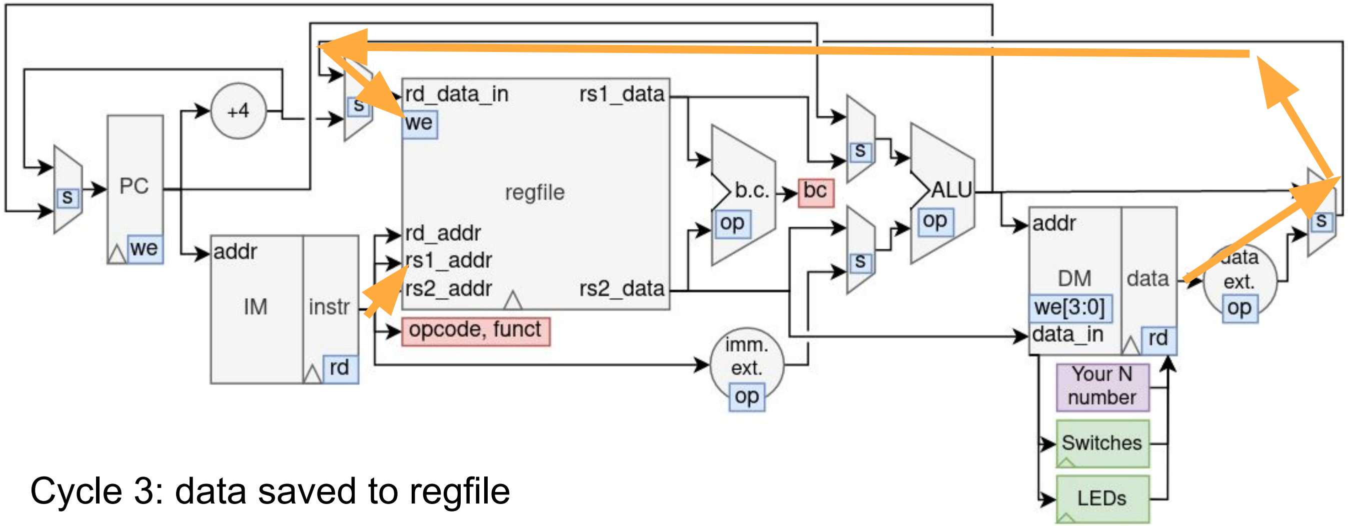
Complete datapath - executing in stages



Cycle 1: address for DM calculated and read executed

E.g.: **LW x3, 0(x0)**

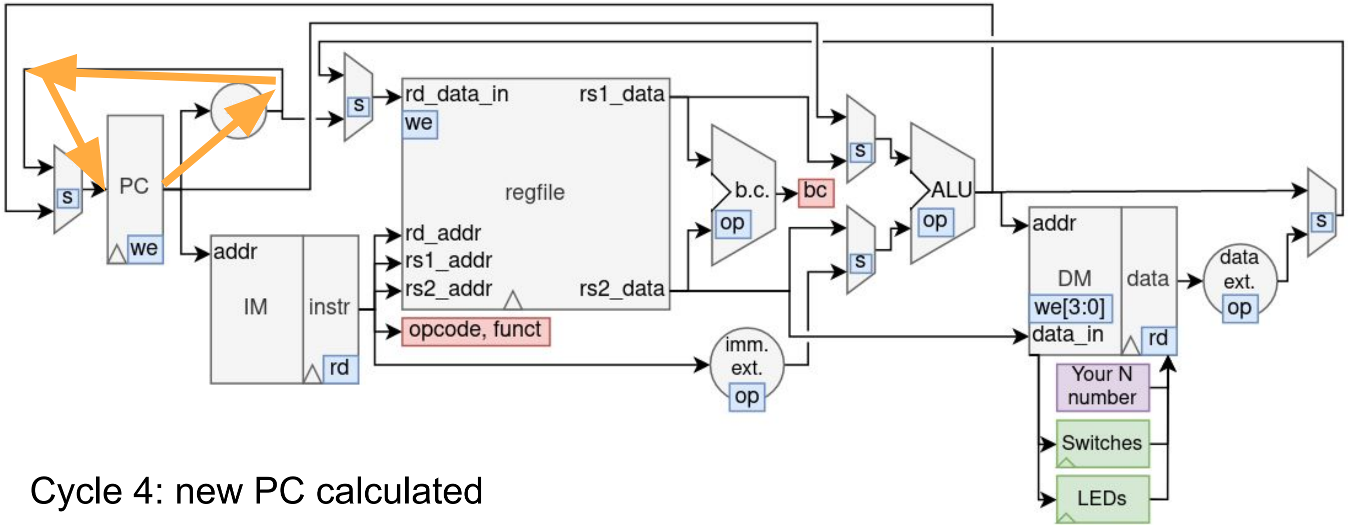
Complete datapath - executing in stages



Cycle 3: data saved to regfile

E.g.: **LW x3, 0(x0)**

Complete datapath - executing in stages

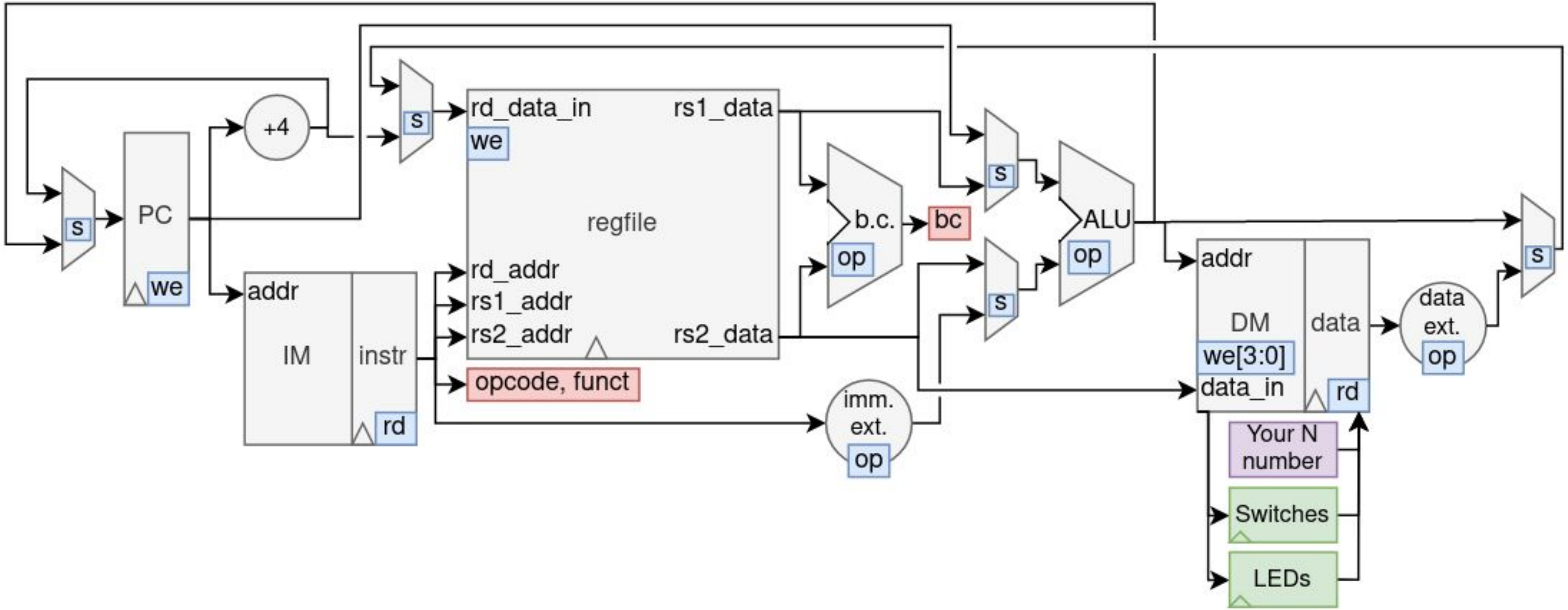


Cycle 4: new PC calculated

E.g.: **LW x3, 0(x0)**

Actual construction of a CPU

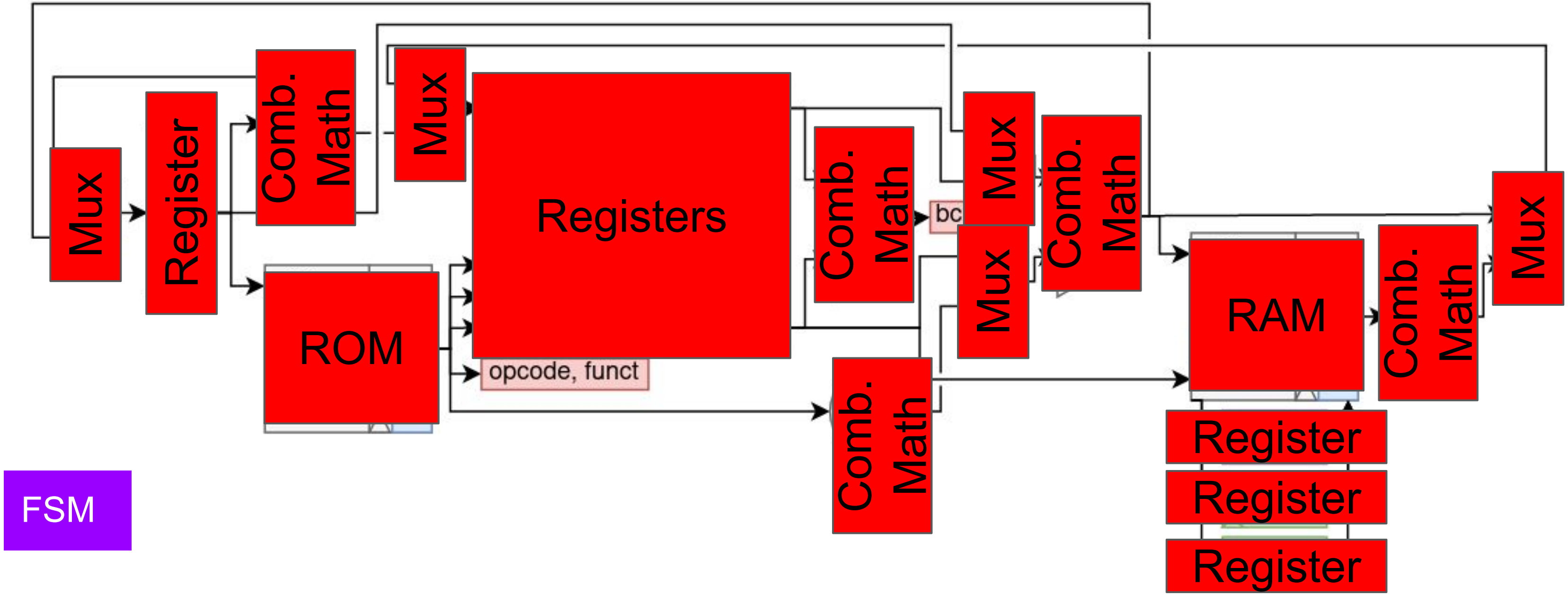
- Consider the simplified architecture diagram:



- Every component here is describable using only material from this course.

Actual construction of a CPU

- Consider the simplified architecture diagram:



- Every component here is describable using only material from this course.

Other possible CPU architectures

- In addition to
  - Layout: Stack/Accumulator/Register-Memory/Register-Register, and
  - ISA: RISC/CISC
- There is also the clocking characteristics:
  - **Multi-cycle - what you will make**
  - **Single-cycle**
  - **Pipelined**

Single-cycle architectures

- Single-cycle architectures execute one instruction per clock cycle
- I.e. each posedge of a clock fetches, decodes, executes, and stores results
- Long critical paths - slower clock speeds
- But, potentially simpler designs

	Cycle 0	Cycle 1	Cycle 2	Cycle 3	Cycle 4
Instruction 0	Executed				
Instruction 1		Executed			
Instruction 2			Executed		
Instruction 3				Executed	
Instruction 4					Executed

Multi-cycle architectures

- Multi-cycle architectures execute one instruction per many clock cycles
- Each posedge of a clock runs one stage of execution
- Shorter critical paths - faster clock speeds
- But, “wasted” HW resources (in a given posedge, HW may not be functioning)

	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
Ins. 0	🟥	🟥	🟥	🟥																
Ins. 1					🟧	🟧	🟧	🟧												
Ins. 2									🟩	🟩	🟩	🟩								
Ins. 3													🟦	🟦	🟦	🟦				
Ins. 4																	🟪	🟪	🟪	🟪

Pipelined architectures

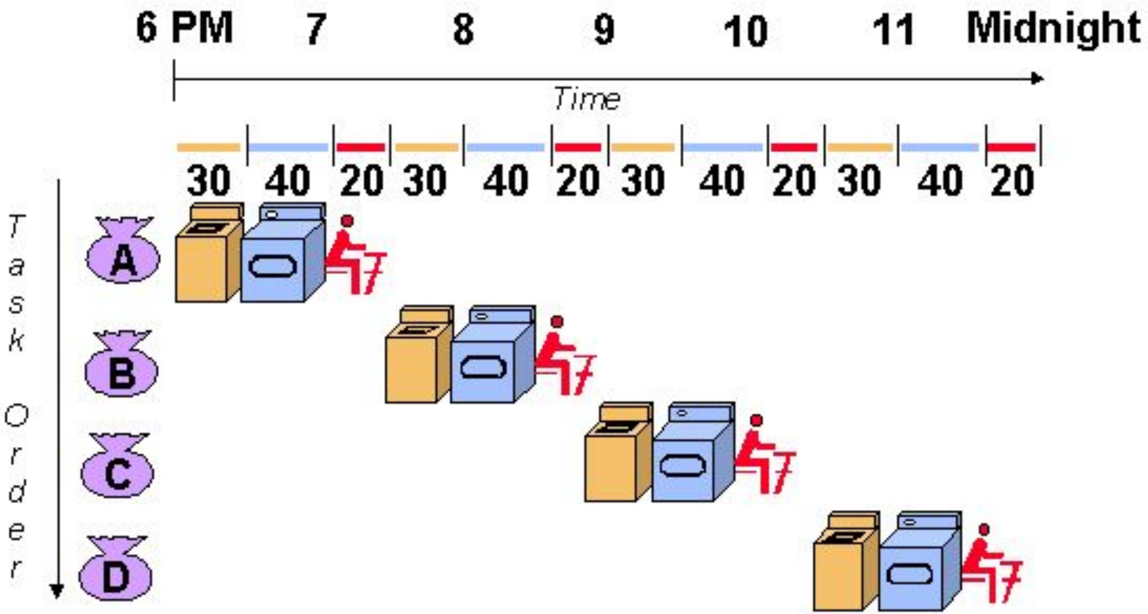
- Pipelined architectures share the resources of a HW system in parallel
- Each posedge of clock runs a stage of execution w/ previous stages also in use
- Shorter critical paths - faster clock speeds + “faster” execution!
- But, ***more complex designs - for “hazards” - “data forwarding” / “stalls”***

	0	1	2	3	4	5	6	7	8	9	..	..	..	..	..	..	..	..	..	..
Ins. 0	🟥	🟥	🟥	🟥																
Ins. 1		🟧	🟧	🟧	🟧															
Ins. 2			🟩	🟩	🟩	🟩														
Ins. 3				🟦	🟦	🟦	🟦													
Ins. 4					🟪	🟪	🟪	🟪												

Understanding pipelining

Single / multi-cycle laundry:

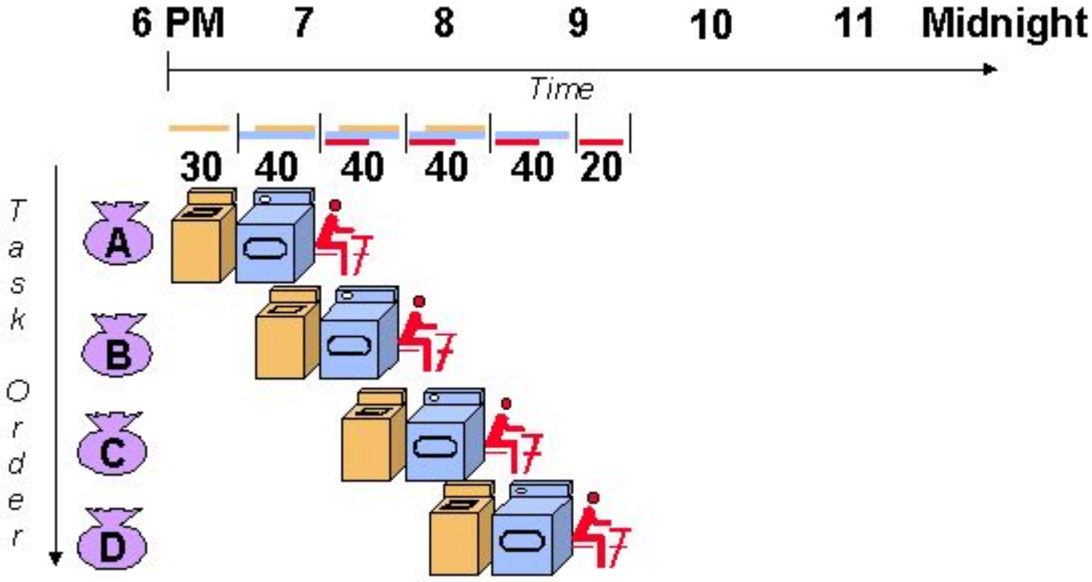
Each person must wait until the previous person has entirely finished



Understanding pipelining

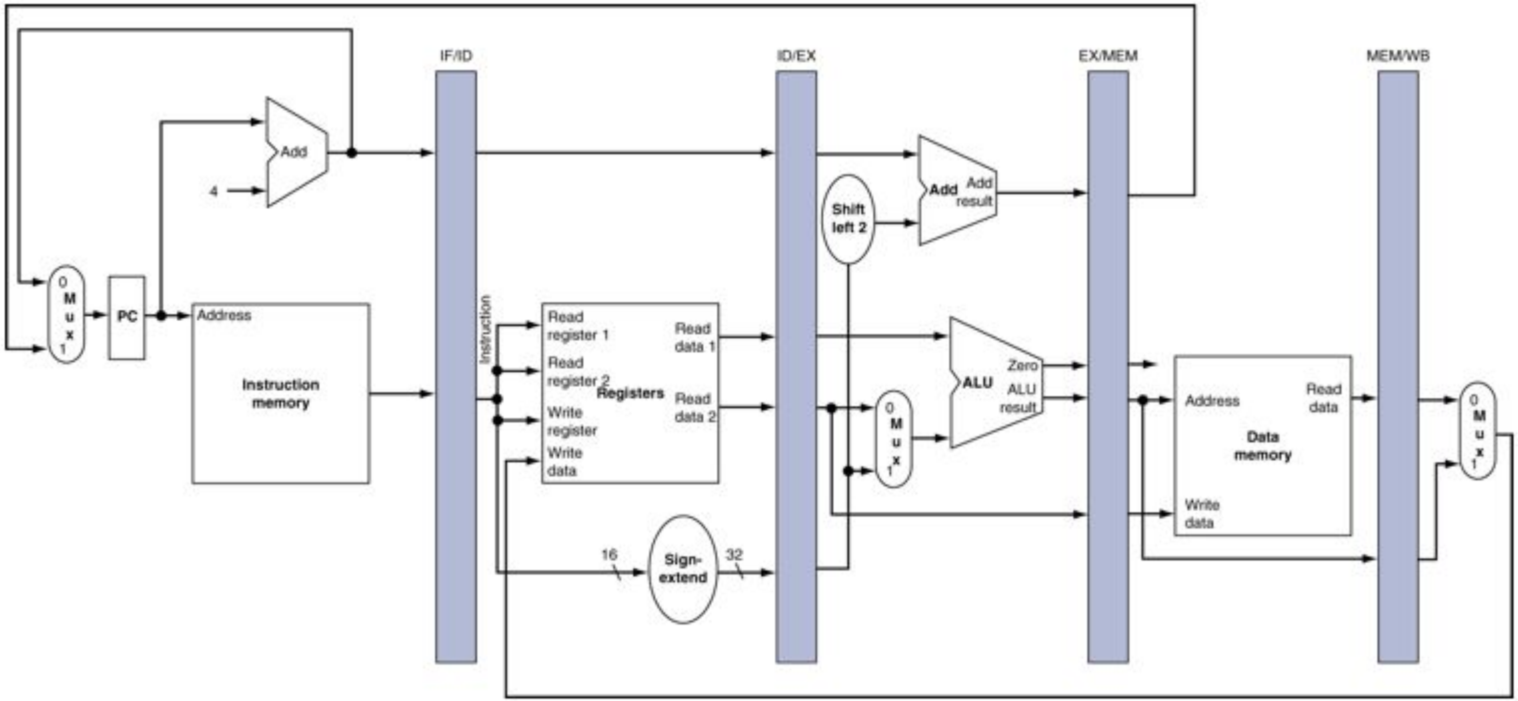
Pipelined laundry:

Now each person can start on their laundry when the appropriate station becomes free



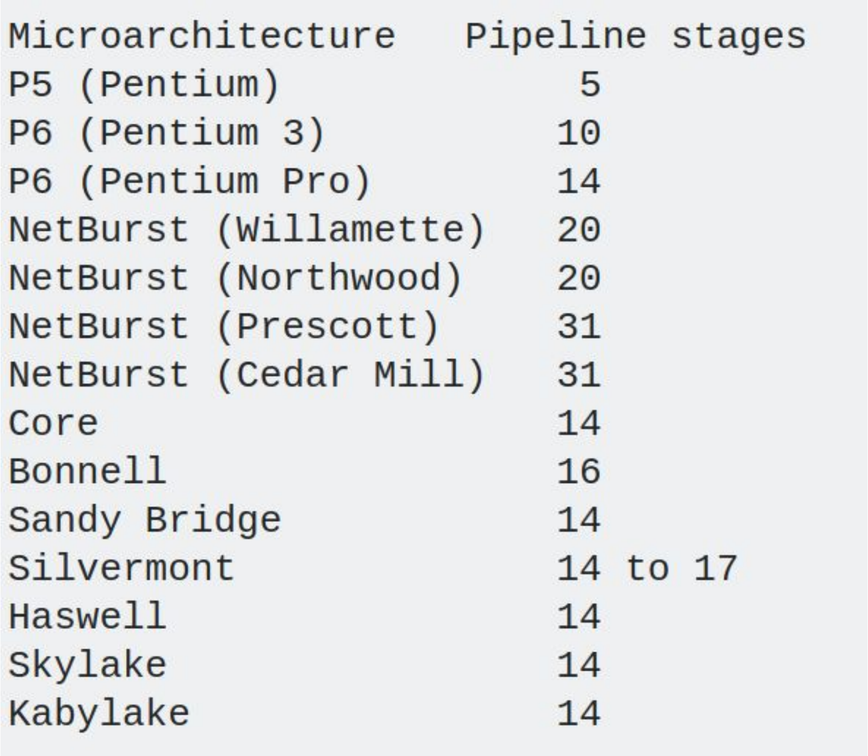
How long are pipelines?

- Typical RISC designs have 4 or 5 (IF, ID, EX, MEM, WB)



How long are pipelines?

For CISC, e.g. Intel processors, they can be *very* long!



Microarchitecture	Pipeline stages
P5 (Pentium)	5
P6 (Pentium 3)	10
P6 (Pentium Pro)	14
NetBurst (Willamette)	20
NetBurst (Northwood)	20
NetBurst (Prescott)	31
NetBurst (Cedar Mill)	31
Core	14
Bonnell	16
Sandy Bridge	14
Silvermont	14 to 17
Haswell	14
Skylake	14
Kabylake	14

Pipelines are one of the marvels of CPU design!

Summary

- NYU-6463-RV32I is complete microprocessor architecture
- No individual component is complex
- Instructions are stored in binary which you can decode
- Overall system works together to execute code
    - Datapath AND Control unit / FSM
- Multi-cycle - do a small thing in each clock cycle