Computer Architecture – An Introduction

CS2053 Computer Architecture

Computer Science & Engineering

University of Moratuwa

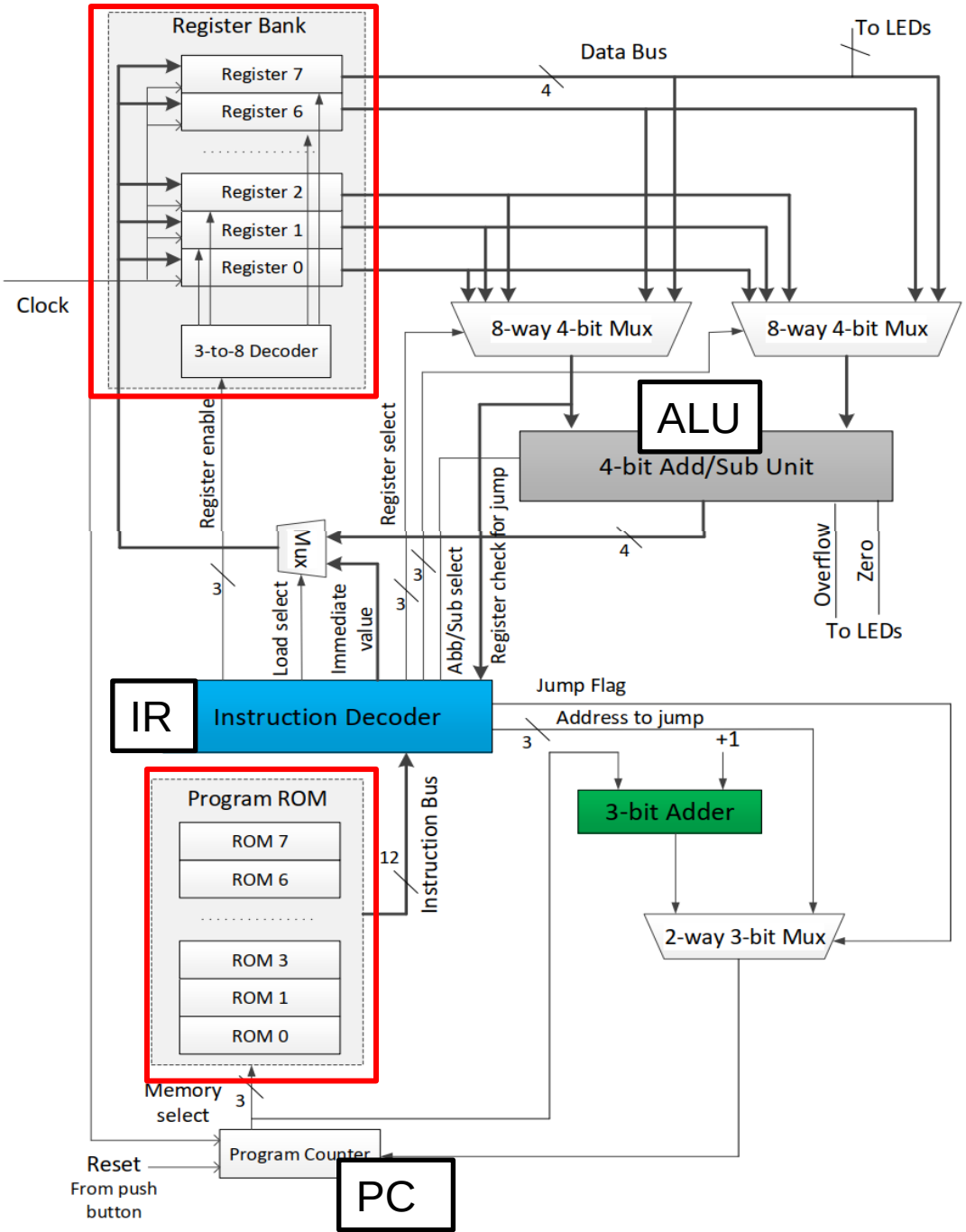
Part II

Computer Architecture

- How to
  - make computers efficient?
  - program computers?
  - make programs executable across different hardware?
  - make programs backward compatible?
  - Make computers connected
- Efficiency
  - Pipelining
  - Caching
  - …. Etc.
- Programmability
  - ISA (Instruction Set Architecture)

Perspective: Nano-processor

- Can you improve it further?
- Dissecting the process to break-down the different “stages”.



Where is ALU?

Where is IR?

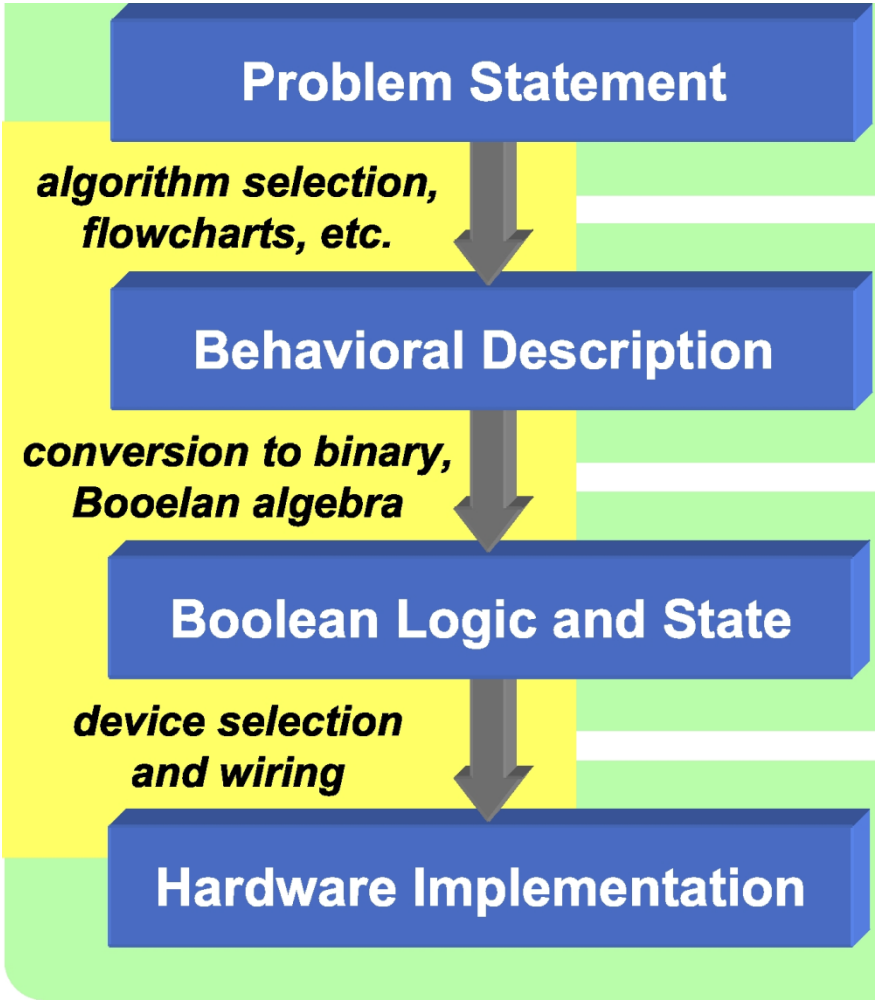
Where is PC?

Where are registers?

Where is the Memory?

*It can be a RAM

Building Digital Solutions to Computational Problems



- Algorithms, RTL, etc.
- Flowcharts
- State transition diagrams

- Logic equations
- Circuit schematics

- TTL Gates (AND, OR, XOR ...)
- Programmable Logic
- Custom ASICs
- FPGAs
- MCs, DSPs

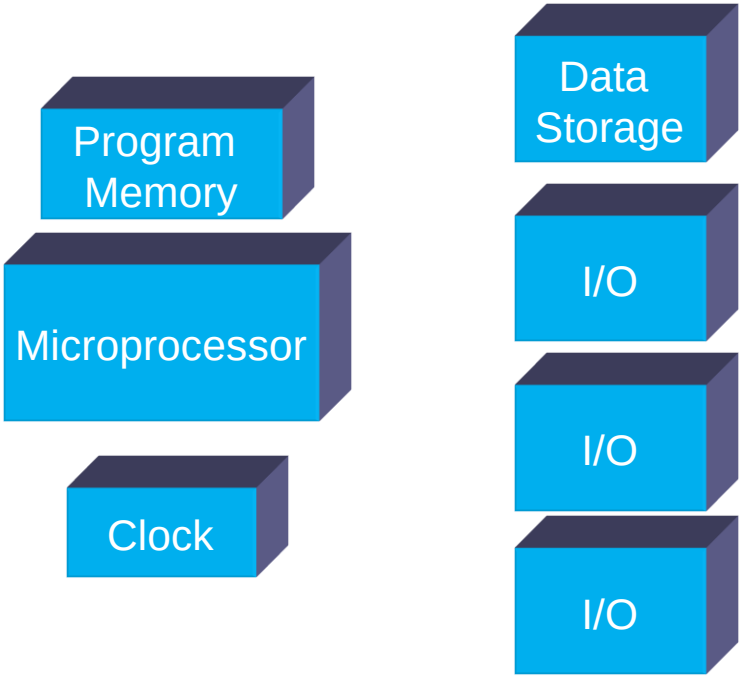
- Verilog or VHDL code
- Assembler
- C, C++

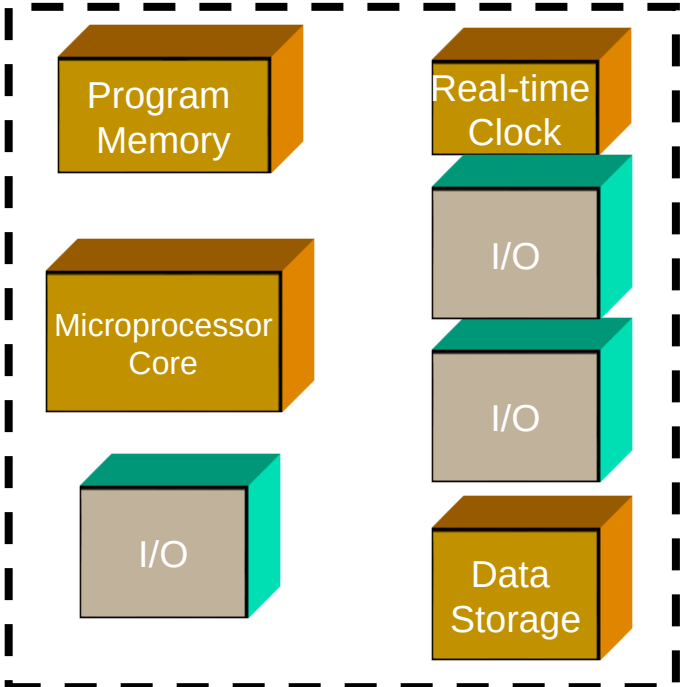
Architectural Differences

- Length of microprocessors' data word
  - 4, 8, 16, 32, 64, & 128 bit
- Speed of instruction execution
  - Clock rate → processor speed
- Instruction set – x86, ARM, SPARC, PIC, RISCV
- CPU architecture – RISC vs. CISC
- Size of direct addressable memory
- Number & types of registers
- Support circuits for performance
- Compatibility with existing software & hardware development systems – IBM System/370

Microprocessor vs. Microcontroller

- Microprocessor – CPU & various IO functions are packed as separate ICs
- Microcontroller – Most IO functions are integrated into same package with CPU
- System on Chip SoC – Processor(s) + GPU + FPGA

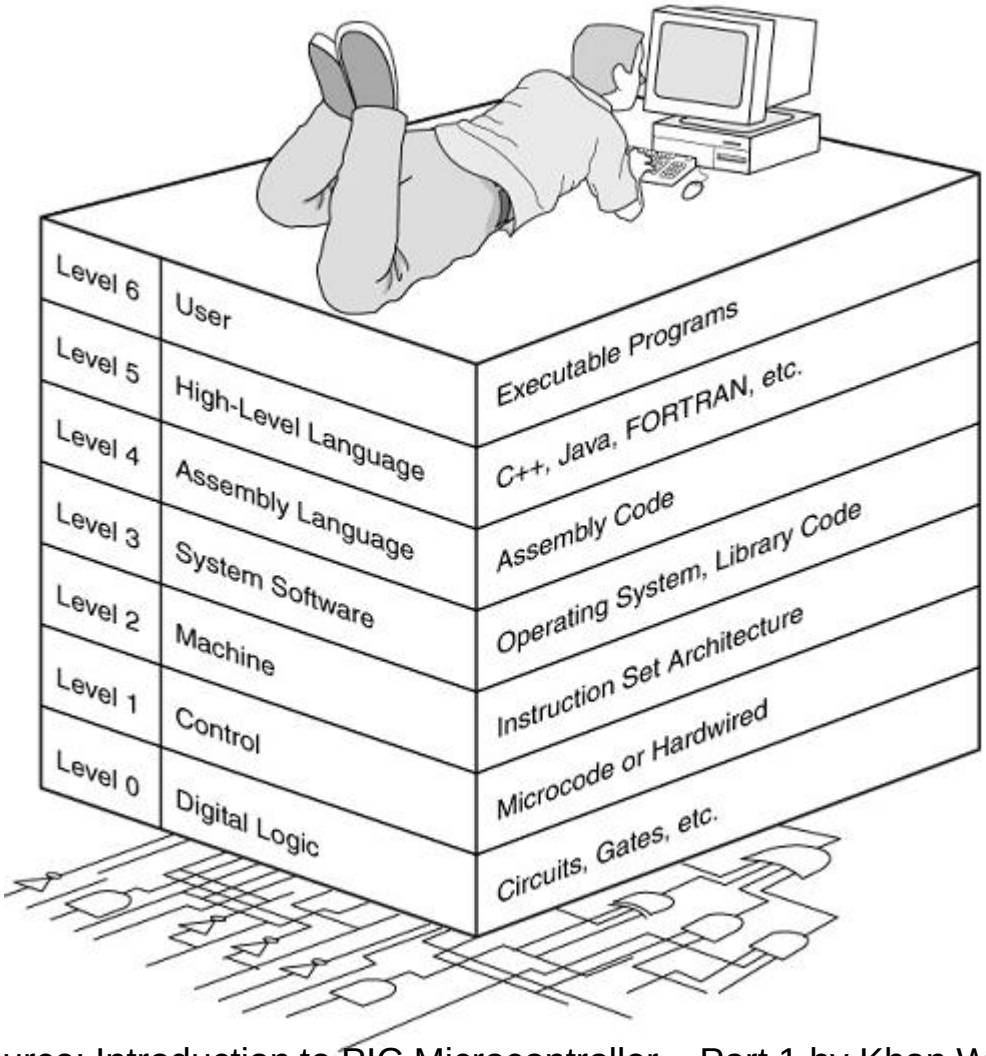




Processor design to programming

- How to program the computer?
  - Make the computer accessible for programmers.
  - What should a programmer know about the computer, in order to execute a program?
- Can the same program run in different processors?
- Different levels of programming

Programming Hierarchies



Source: Introduction to PIC Microcontroller – Part 1 by Khan Wahid

What is missing in nano-processor?

- Stages of executing an instruction
  - Nano-processor was single cycle
    - Single cycle vs multicycle CPU
  - Example
    - Fetch instruction
    - Increment Program Counter
    - Decode instruction
    - Fetch operands from memory
    - Execute instruction
    - Store results back to memory
    - Go to first stage and repeat
- **Pipelining**

What is missing in nano-processor?

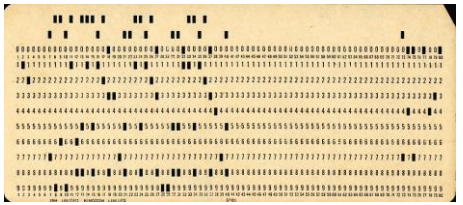
- Peripherals
  - Memory : RAM (internal, external)
    - **Memory Hierarchy / Cache hierarchy**
  - Inputs Outputs
    - Memory mapped inputs/outputs
- Way to program it
  - Abstraction of implementation details from usage
  - What do you need to know about the processor in order to program it?
    - Instruction set / Registers / Memory map /Interrupts

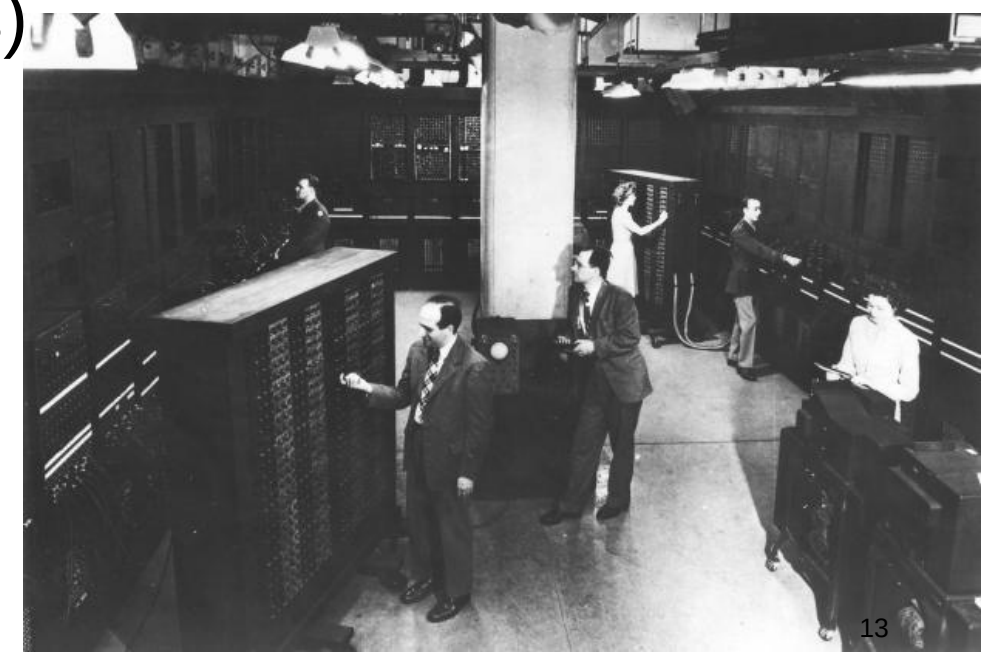
What is missing in nano-processor?

- Efficiency/ Power
  - Features of the processor
    - Floating point/ special co-processors
  - How to speedup?
    - Clock speed
    - Integer operations per second (IOPS) (FLOPS)
  - How to reduce energy consumption ?
  - How to manage heat?

Programming Language Levels

- Machine code (40s-50s)
  - 0001000000111000 0001001000110100
  - 0101110000000000
  - 0001110000000000 0001001000110101
- Hex notation (50s-60s)
  - 1038 1234
  - 5C00
  - 1E00 1235





Source: http://mentalfloss.com/article/53160/meet-refrigerator-ladies-who-programmed-eniac

Programming Language Levels (Cont.)

- Assembler
  - Machine code (60s-70s)

```
.define const = 6
num1: .byte [1]
num2: .byte [2]
move.b num1,d0
addq.b #const,d0
move.b d0,num2
```

- High-level languages
  - C code fragment (70s-80s)

```
#define const  6
int  num1, num2;
num2 = num1 + const;
```

Why Assembly is still useful?

- Can produce code that runs fast
- Better use of CPU resources
- Only way to use some advanced features

Example RISC-V Program

```
.globl main

main:                           # Register t1 is also called register 6 (x6)

        li t1, 0x0                      # t1 = 0
REPEAT:
        addi t1, t1, 6                  # t1 = t1 + 6
        addi t1, t1, -1                 # t1 = t1 - 1
        andi t1, t1, 3                  # t1 = t1 AND 3
        beq  zero, zero, REPEAT   # Repeat the loop
        nop
.end
```

Instruction Execution Sequence

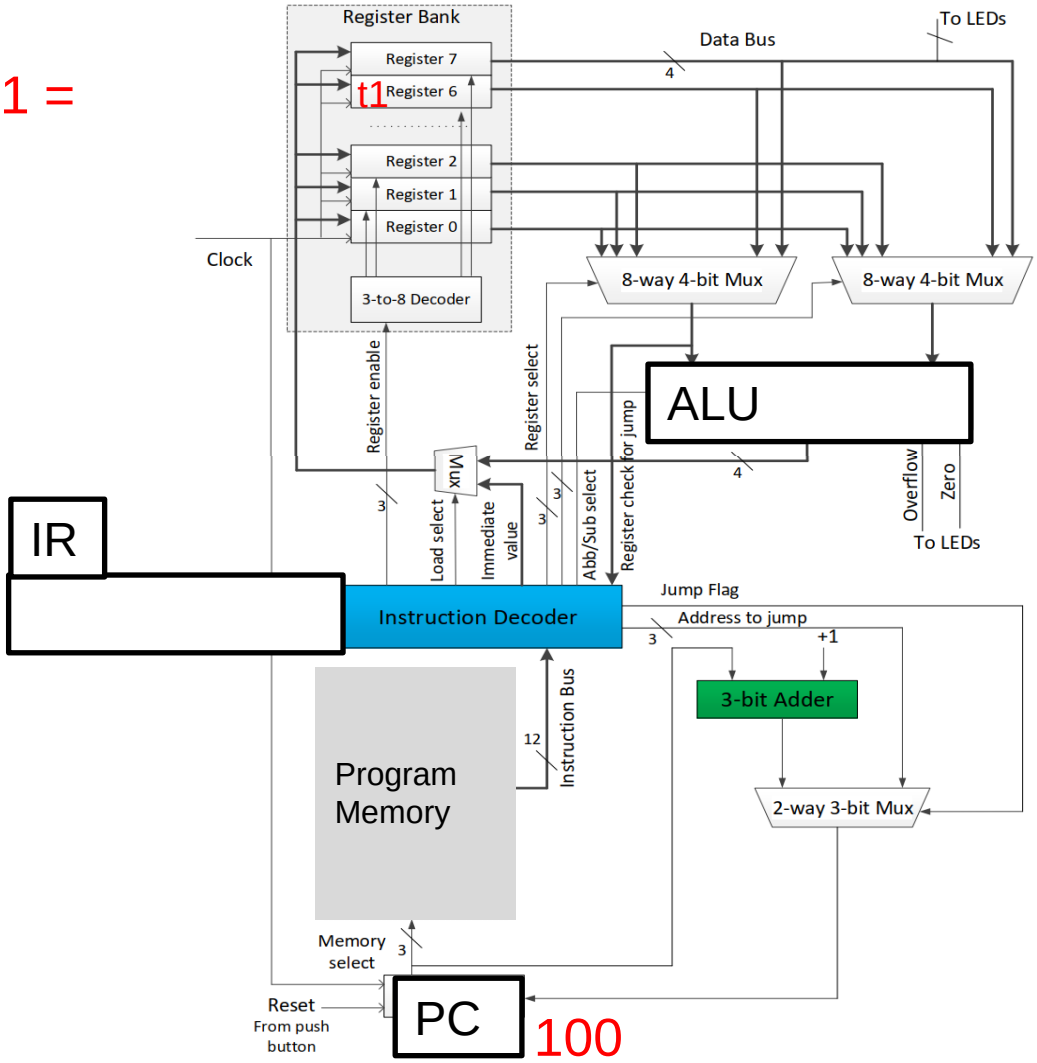
1. Fetch next instruction from memory to IR
2. Change PC to point to next instruction
3. Determine type of instruction just fetched
4. If instruction needs data from memory, determine where it is
5. Fetch data if needed into register
6. Execute instruction
7. Store results back to the memory if needed
8. Go to step 1 & continue with next instruction

Execution of a program

t1 =

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory



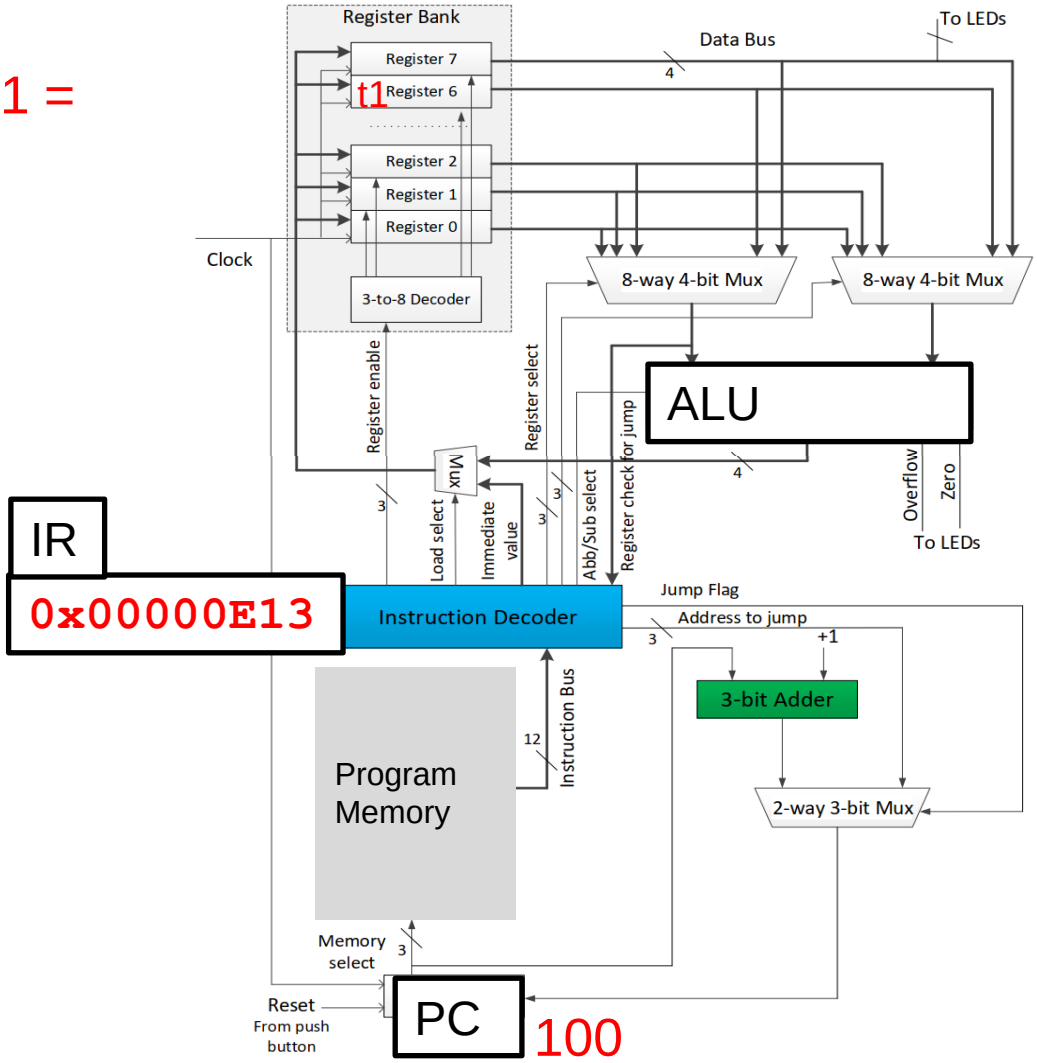
Execution of a program

Fetch Instruction

t1 =

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory



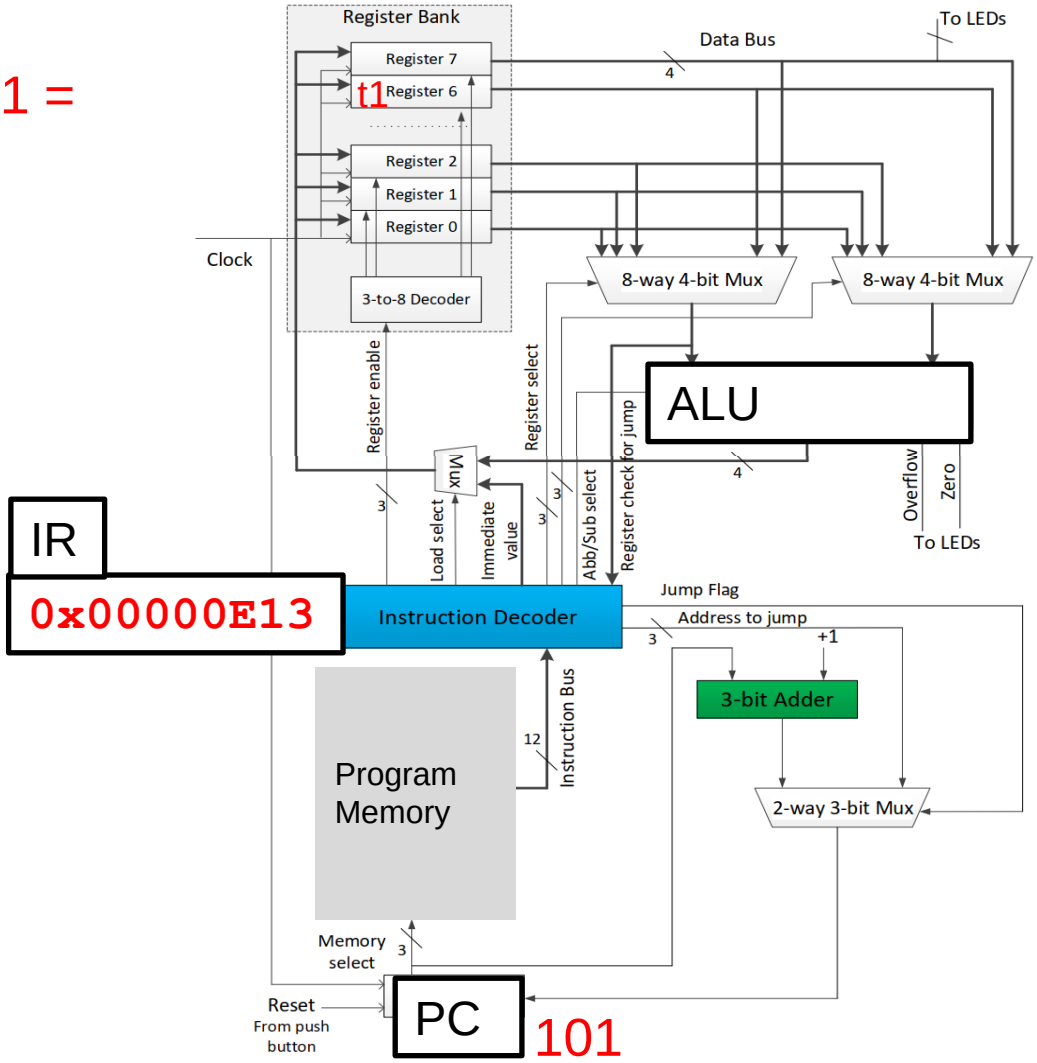
Execution of a program

Fetch Instruction
Increment PC

t1 =

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory



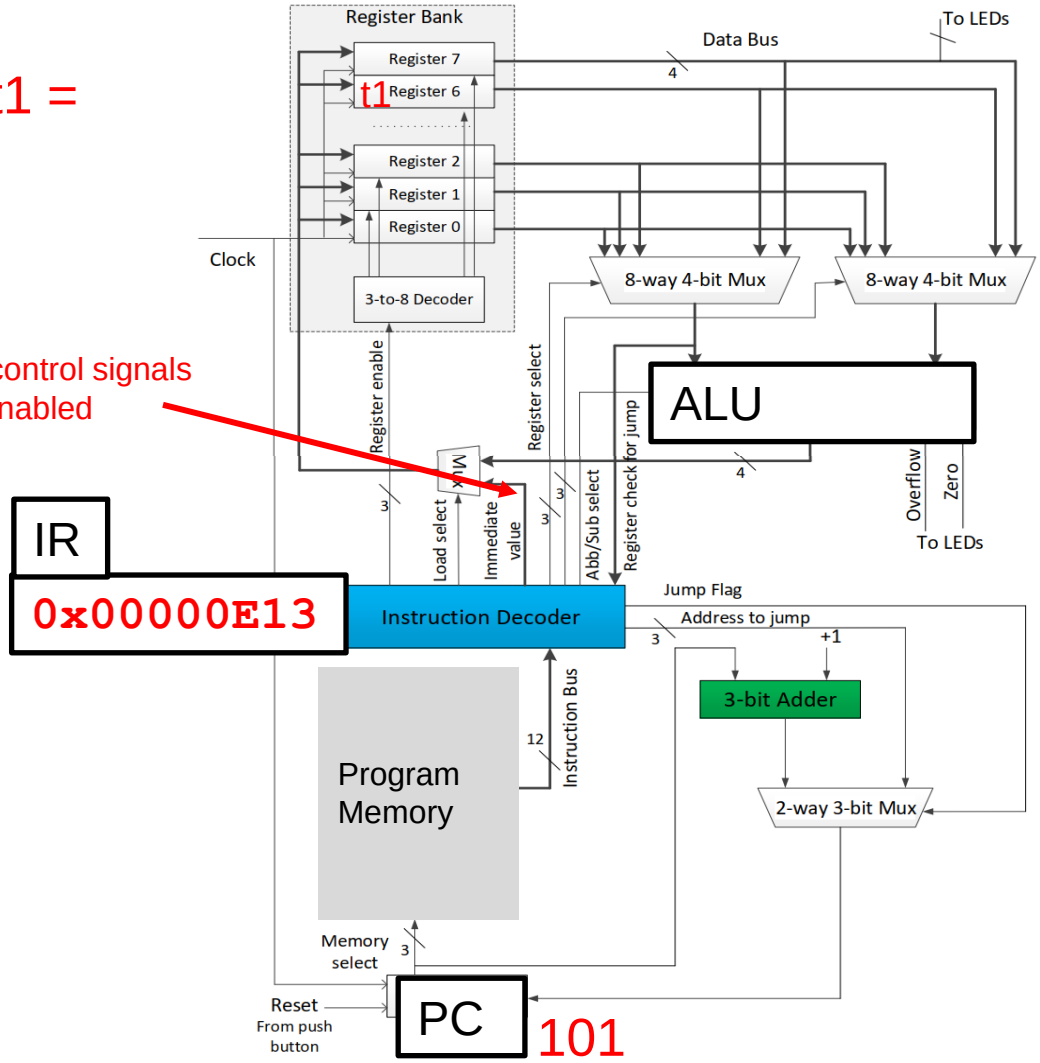
Execution of a program

Fetch Instruction
Increment PC
Decode

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory

t1 =



Execution of a program

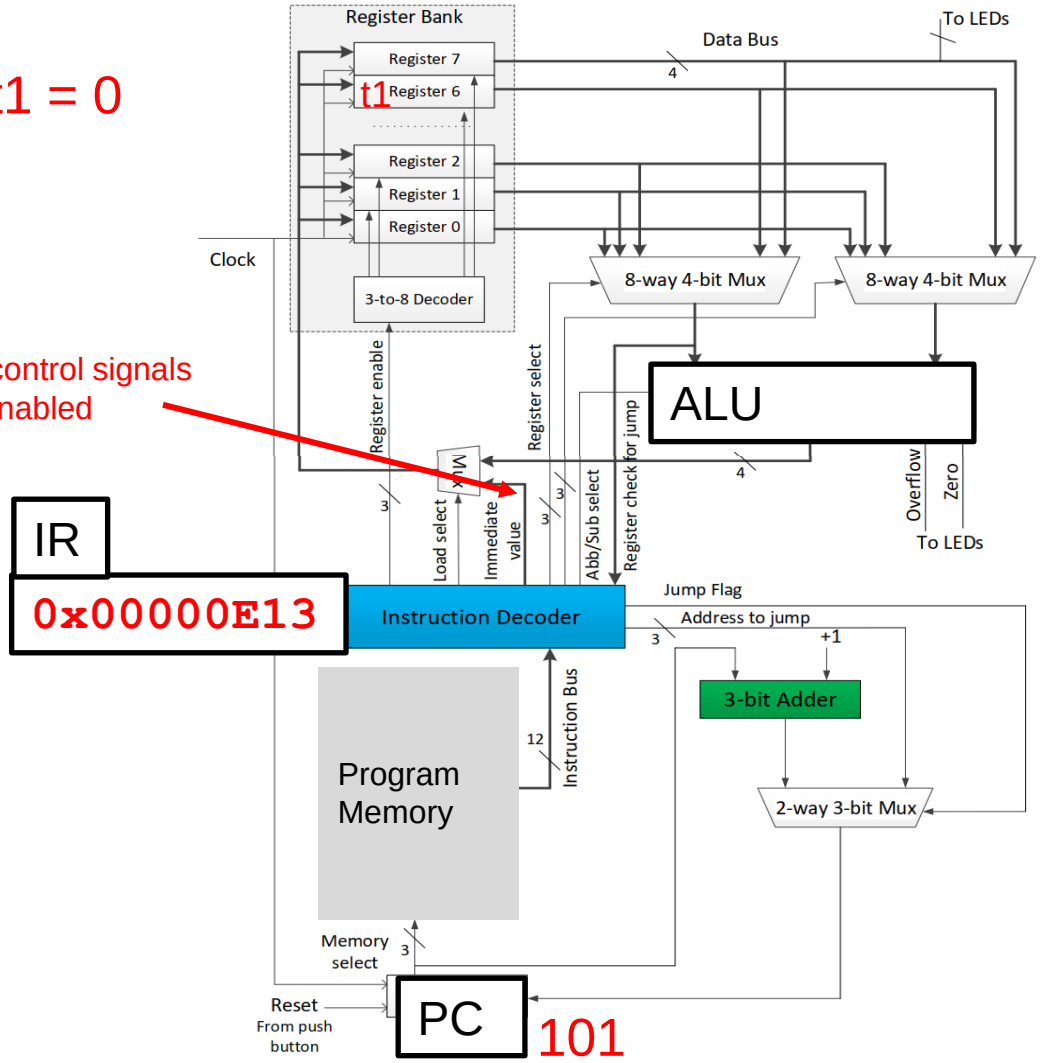
Fetch Instruction
Increment PC
Decode
Execute

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory

t1 = 0

The control signals are enabled



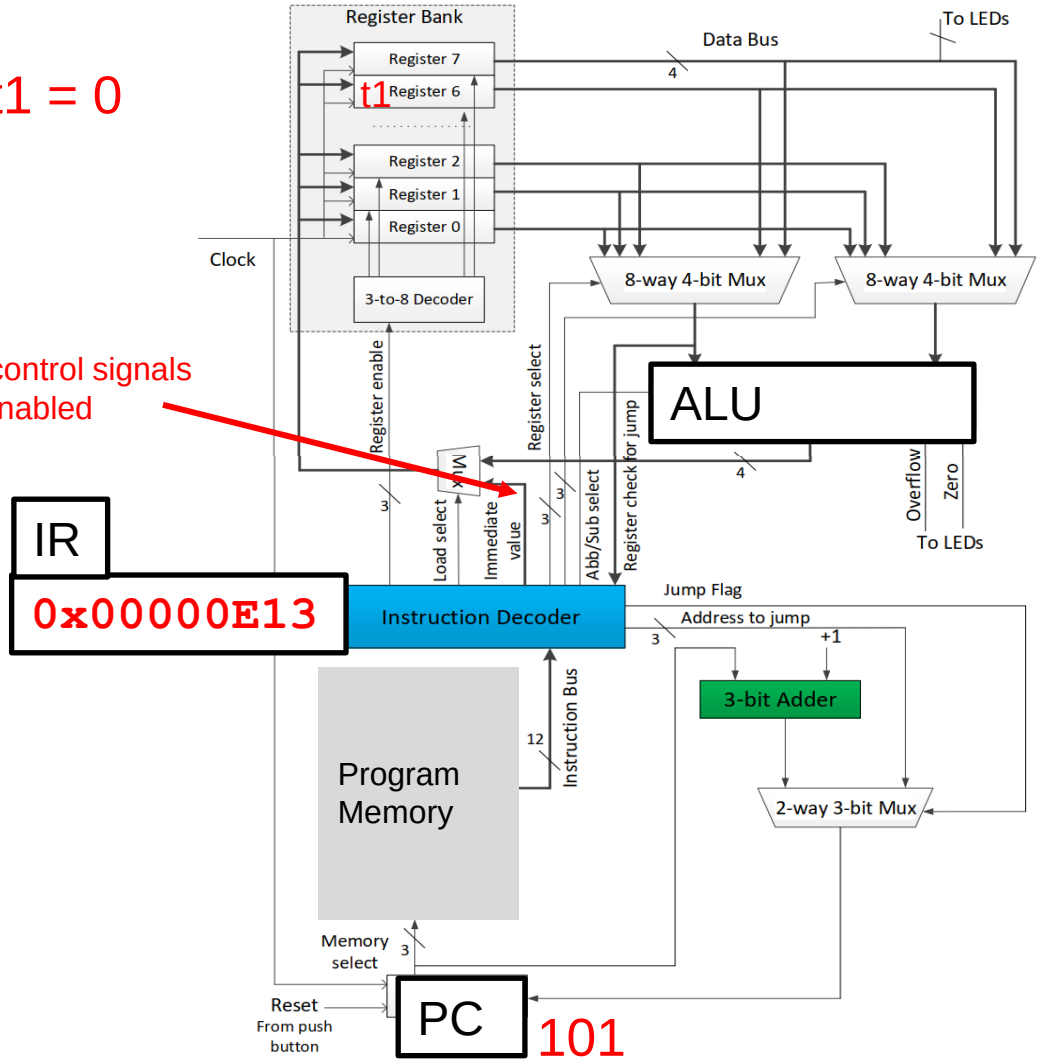
Execution of a program

Fetch Instruction
Increment PC
Decode
Execute
Writeback?

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory

t1 = 0



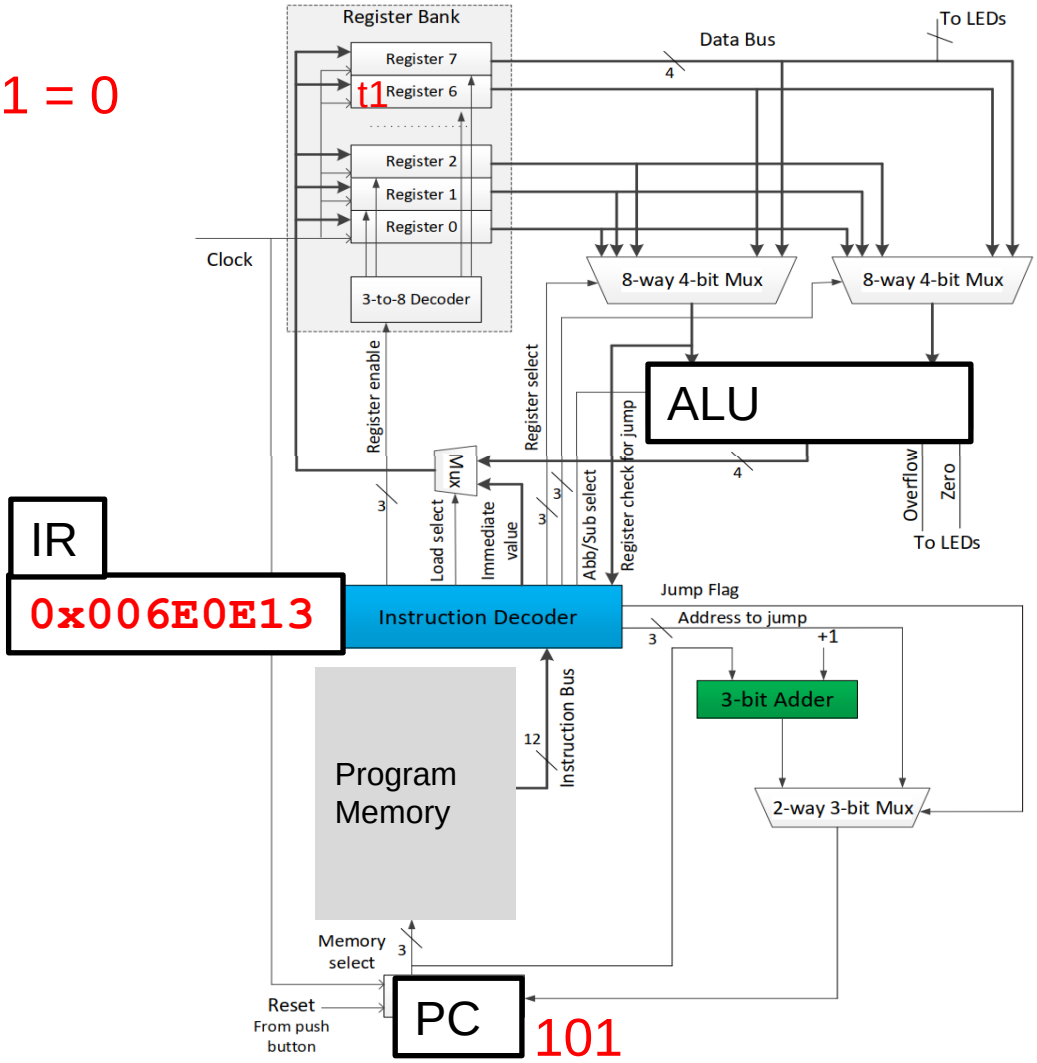
Execution of a program

Fetch Instruction

t1 = 0

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory



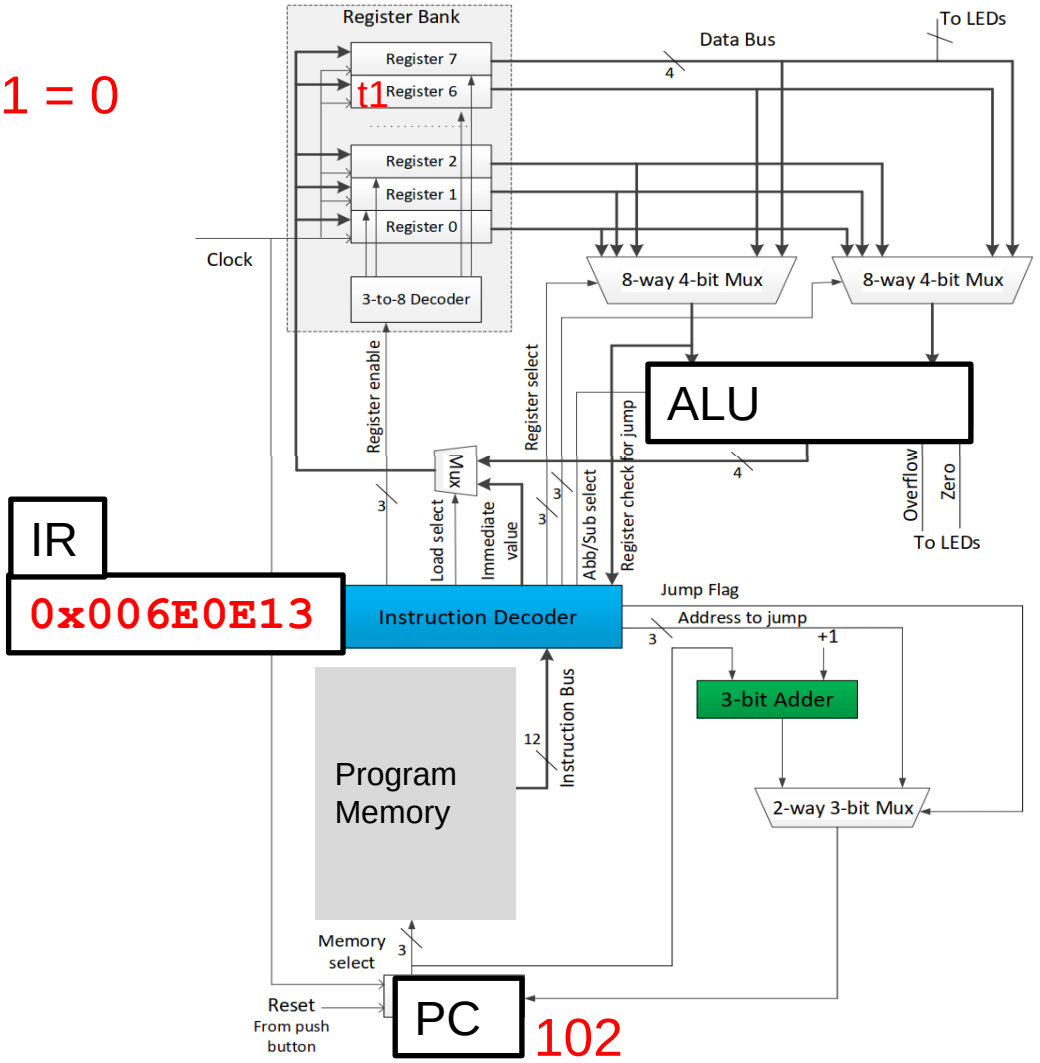
Execution of a program

Fetch Instruction
Increment PC

t1 = 0

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory

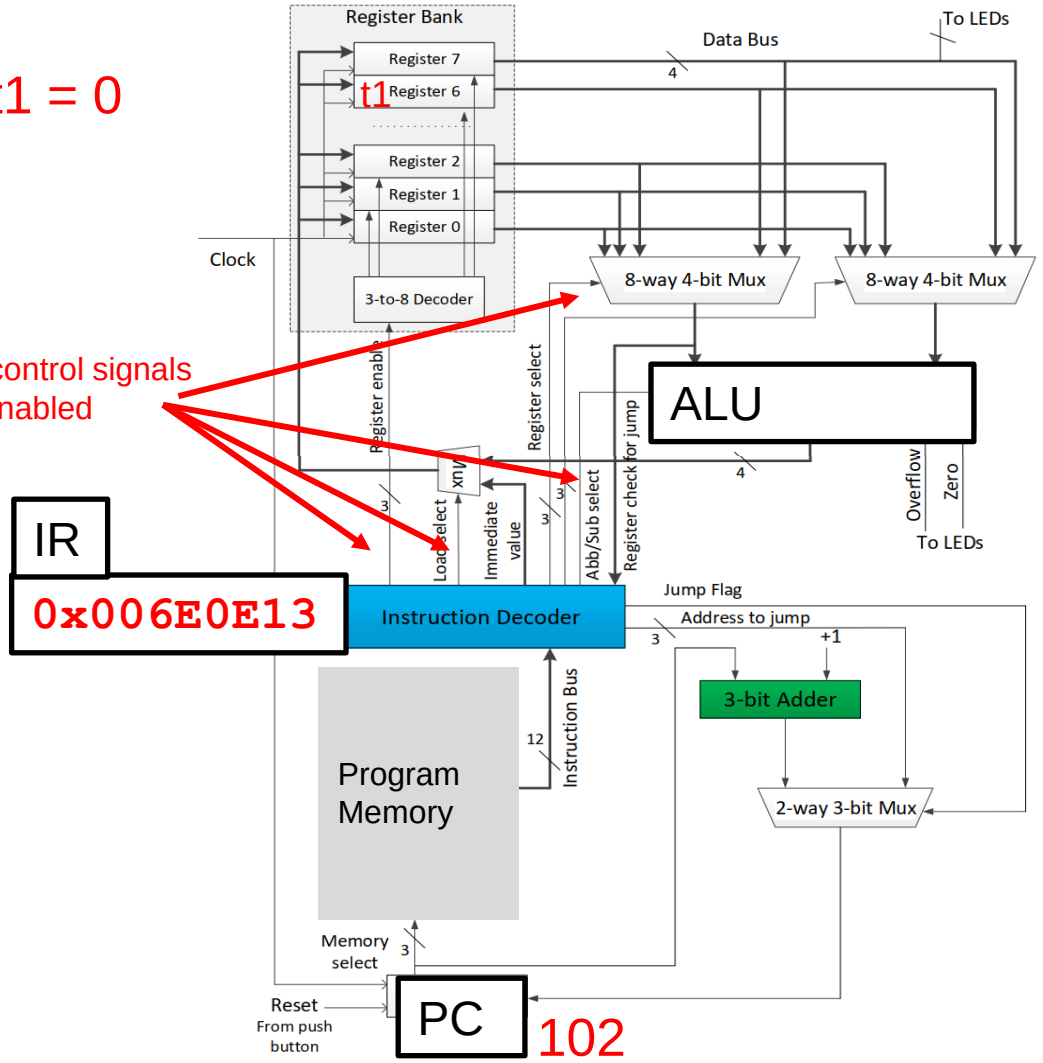


Execution of a program

Fetch Instruction
Increment PC
Decode

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

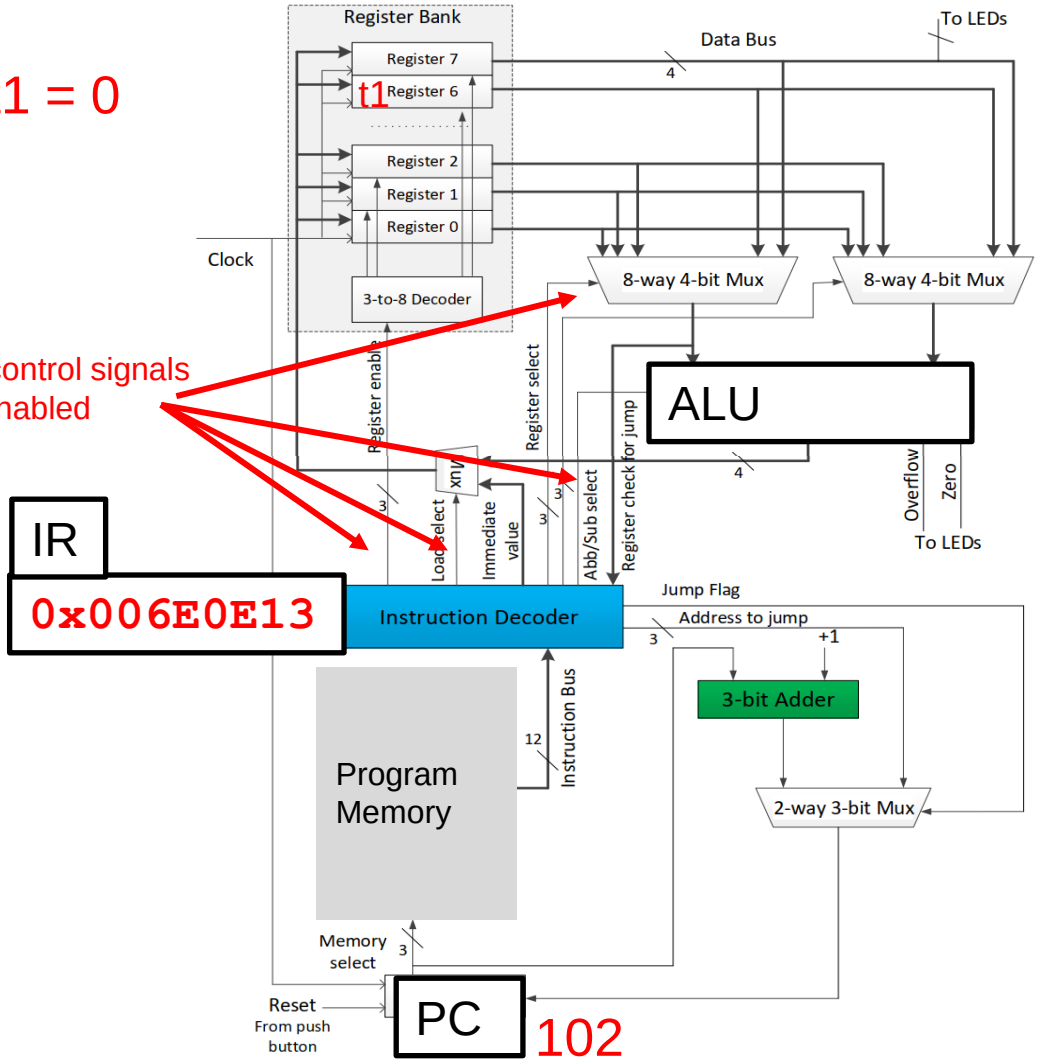
Program memory



Execution of a program

Fetch Instruction
Increment PC
Decode

t1 = 0



100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory

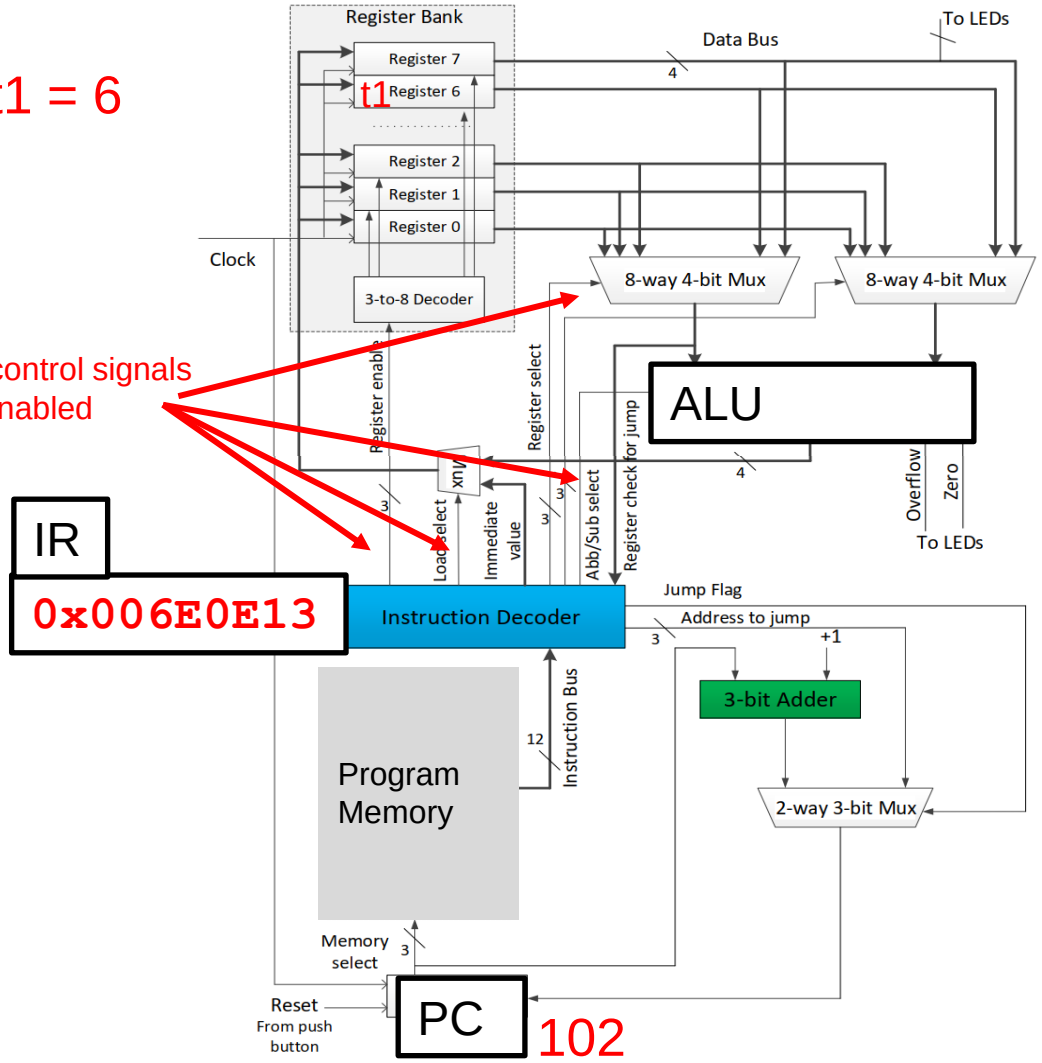
Execution of a program

Fetch Instruction
Increment PC
Decode
Execute

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory

t1 = 6



The control signals are enabled

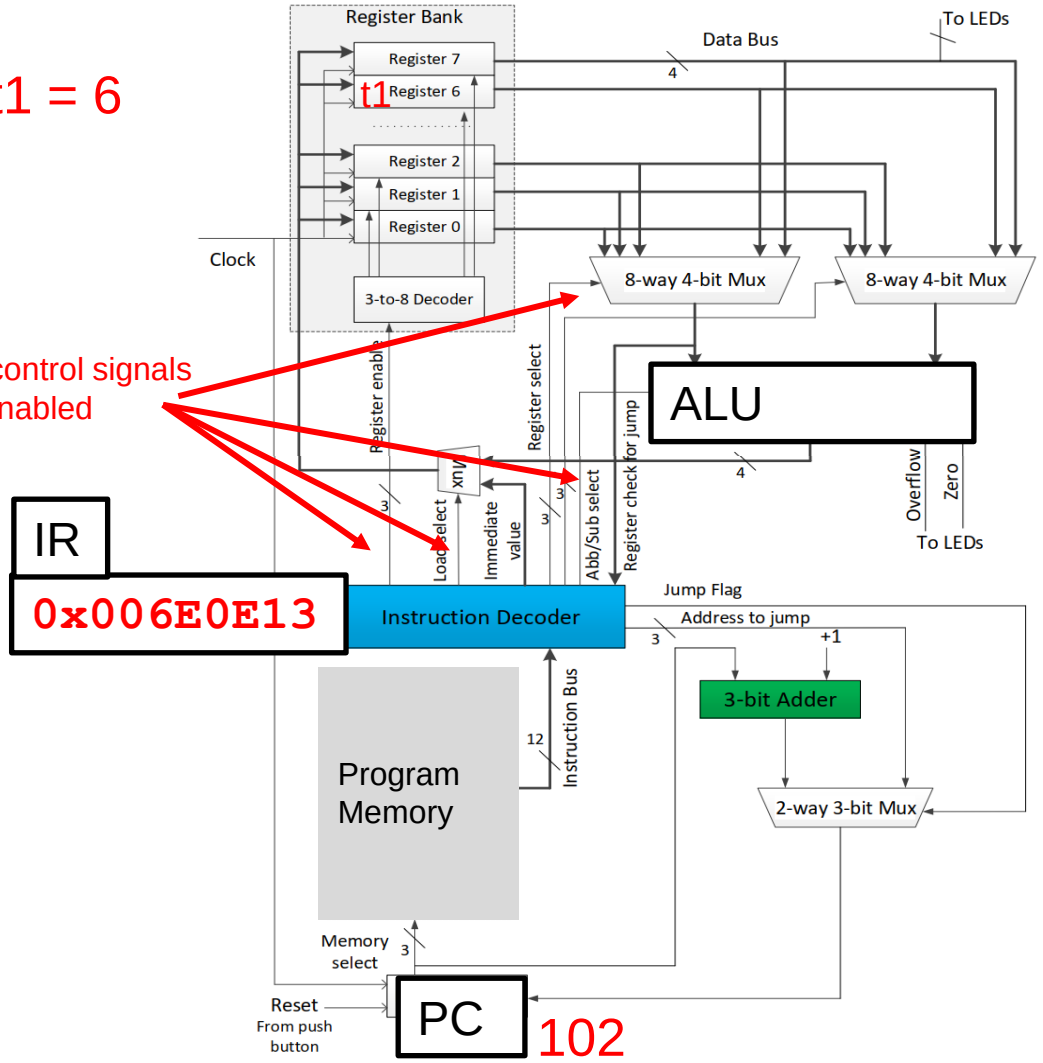
Execution of a program

Fetch Instruction
Increment PC
Decode
Execute
Writeback?

100	li t1, 0x0
101	addi t1, t1, 6
102	addi t1, t1, -1
103	andi t1, t1, 3
104	
105	

Program memory

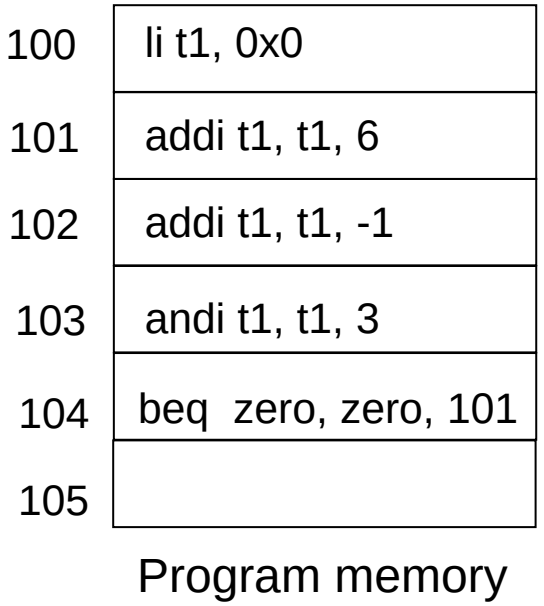
t1 = 6



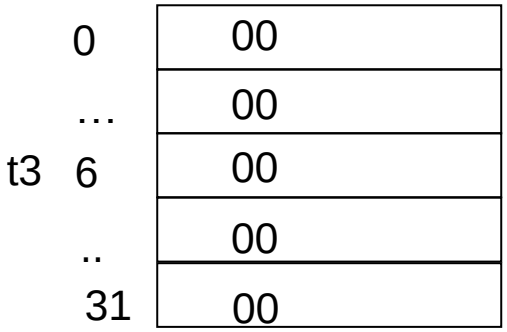
The control signals are enabled

Simple RISC-V Assembly program

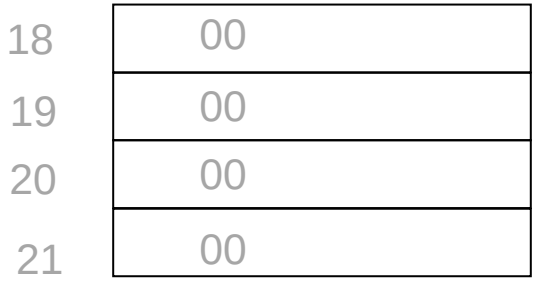
```
.globl main

main:
     li t1, 0x0                    # t1 = 0
REPEAT:
     addi t1, t1, 6     # t1 = t1 + 6
     addi t1, t1, -1    # t1 = t1 - 1
     andi t1, t1, 3     # t1 = t1 AND 3

     beq  zero, zero, REPEAT
                        # Repeat the loop
     nop
.end
```



Program memory



Register Bank



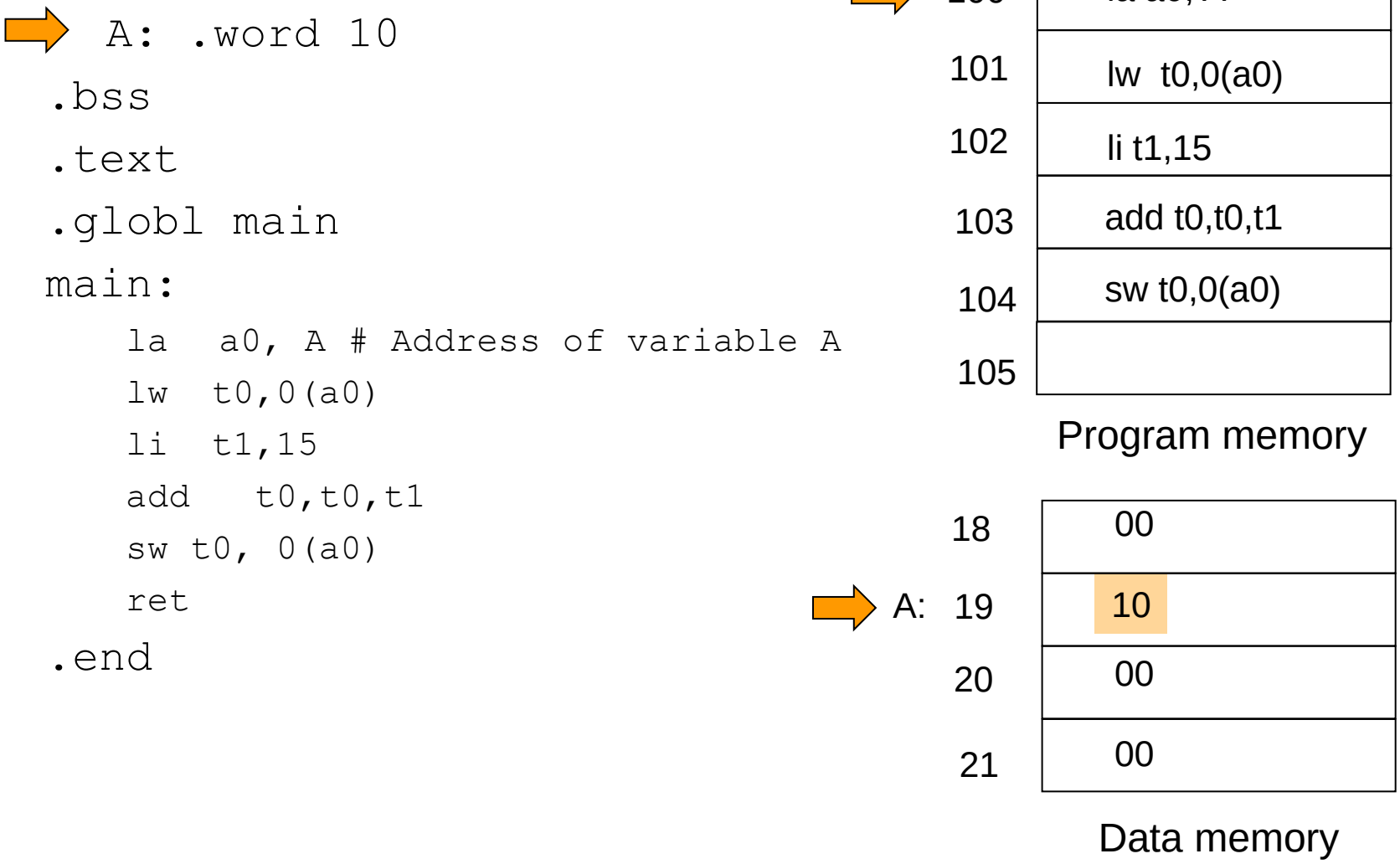
Data memory

Instruction Execution Sequence

1. Fetch next instruction from memory to IR
2. Change PC to point to next instruction
3. Determine type of instruction just fetched
4. If instruction needs data from memory, determine where it is
5. Fetch data if needed into register
6. Execute instruction
7. Store results back to the memory if needed
8. Go to step 1 & continue with next instruction

Sample Program

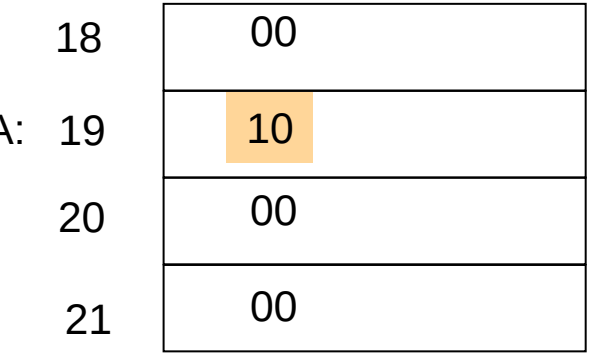
```
.data
 A: .word 10
.bss
.text
.globl main
main:
    la   a0, A # Address of variable A
    lw   t0,0(a0)
    li   t1,15
    add    t0,t0,t1
    sw t0, 0(a0)
    ret
.end
```



Sample Program

```
.data
   A:  .word 10
.bss
.text
.globl main
main:
    la   a0, A # Address of variable A
    lw   t0,0(a0)
    li   t1,15
    add    t0,t0,t1
    sw t0, 0(a0)
    ret
.end
```

Program memory



Data memory

Execution of a program

	0	00
	…	00
t3	6	00
	..	00
	31	00

Register Bank

a0 =

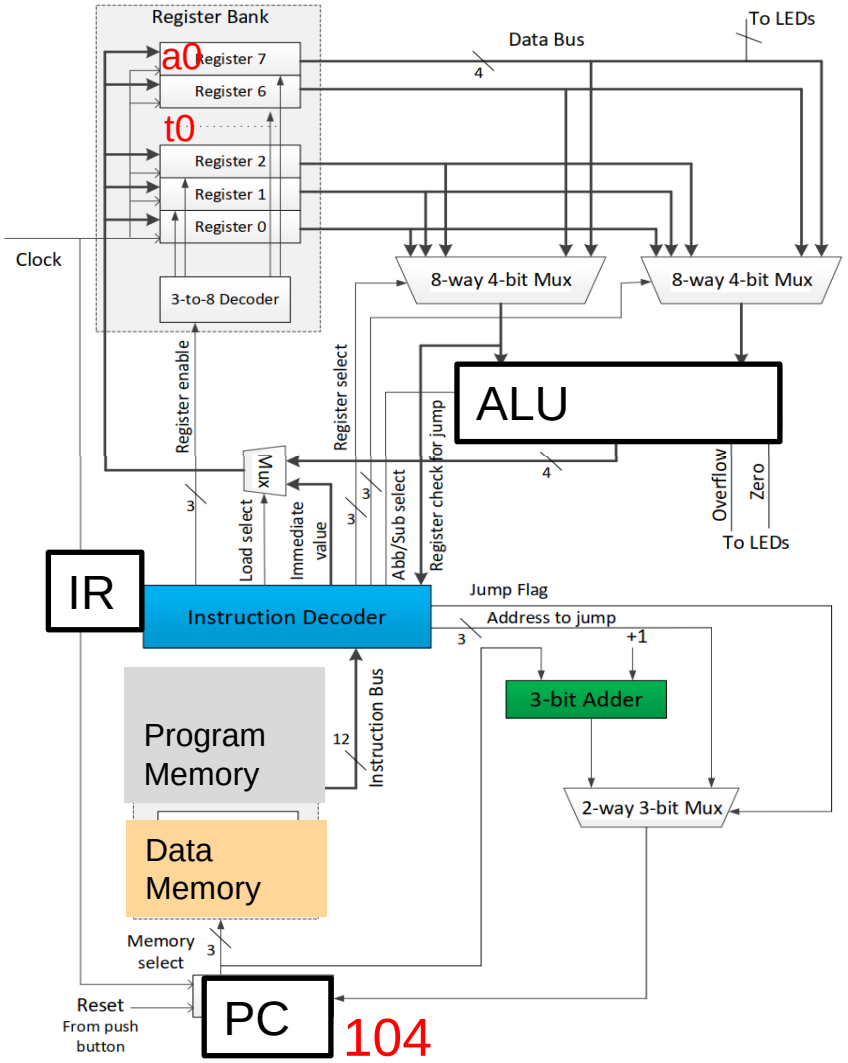
t0 =

	100	la a0, A
	101	lw t0,0(a0)
	102	li t1,15
	103	add t0,t0,t1
⇨	104	sw t0,0(a0)
	105	

Program memory

	18	00
⇨ A:	19	10
	20	00
	21	00

Data memory



Assembling and linking

- Convert written symbols to binary (**Assembling**)

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011
Instruction	**Format**	**immediate**		**rs1**	**funct3**	**rd**	**opcode**
addi (add immediate)	I	constant		reg	000	reg	0010011
ld (load doubleword)	I	address		reg	011	reg	0000011
Instruction	**Format**	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**
sd (store doubleword)	S	address	reg	reg	011	address	0100011

FIGURE 2.5 RISC-V instruction encoding. In the table above, "reg" means a register number between 0 and 31 and "address" means a 12-bit address or constant. The funct3 and funct7 fields act as additional opcode fields.

R type (Register)
I type (Immediate)
S type (Store)

- Mapping memory to variables and inputs/outputs
- Bringing multiple functions/programs to work together (**linking**)

ISA: Instruction Set Architecture

- What programmer should know to use a processor?
  - Set of registers ?
  - Instruction set ?
  - Memory map ? (RAM/ Inputs / Outputs)

  - Compilers make it easy to program at even higher levels

ISA: Instruction Set Architecture
Layer of abstraction between hardware and software

How to design a good ISA?

RISC –V Registers

32 32-bit registers

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporary variables
s0/fp	x8	Saved variable / Frame pointer
s1	x9	Saved variable
a0-1	x10-11	Function arguments / Return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved variables
t3-6	x28-31	Temporary variables

RISC V Instruction Set

Common RISC-V Assembly Instructions & Pseudoinstructions

RISC-V Assembly	Description	Operation
`add s0, s1, s2`	Add	s0 = s1 + s2
`sub s0, s1, s2`	Subtract	s0 = s1 - s2
`addi t3, t1, -10`	Add immediate	t3 = t1 – 10
`mul t0, t2, t3`	32-bit multiply	t0 = t2 * t3
`div s9, t5, t6`	Division	t9 = t5 / t6
`rem s4, s1, s2`	Remainder	s4 = s1 % s2
`and t0, t1, t2`	Bit-wise AND	t0 = t1 & t2
`or t0, t1, t5`	Bit-wise OR	t0 = t1 | t5
`xor s3, s4, s5`	Bit-wise XOR	s3 = s4 ^ s5
`andi t1, t2, 0xFFB`	Bit-wise AND immediate	t1 = t2 & 0xFFFFFFFB
`ori t0, t1, 0x2C`	Bit-wise OR immediate	t0 = t1 | 0x2C
`xori s3, s4, 0xABC`	Bit-wise XOR immediate	s3 = s4 ^ 0xFFFFFABC
`sll t0, t1, t2`	Shift left logical	t0 = t1 << t2
`srl t0, t1, t5`	Shift right logical	t0 = t1 >> t5
`sra s3, s4, s5`	Shift right arithmetic	s3 = s4 >>> s5
`slli t1, t2, 30`	Shift left logical immediate	t1 = t2 << 30
`srli t0, t1, 5`	Shift right logical immediate	t0 = t1 >> 5
`srai s3, s4, 31`	Shift right arithmetic immediate	s3 = s4 >>> 31

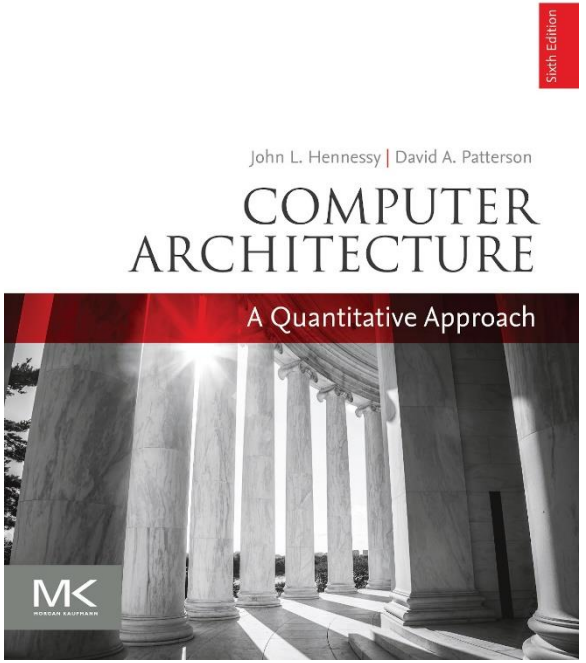
RISC V Instruction Set

Common RISC-V Assembly Instructions & Pseudoinstructions (continued)

RISC-V Assembly	Description	Operation
`lw s7, 0x2C(t1)`	Load word	s7 = memory[t1+0x2C]
`lh s5, 0x5A(s3)`	Load half-word	s5 = SignExt(memory[s3+0x5A]$_{15:0}$)
`lb s1, -3(t4)`	Load byte	s1 = SignExt(memory[t4-3]$_{7:0}$)
`sw t2, 0x7C(t1)`	Store word	memory[t1+0x7C] = t2
`sh t3, 22(s3)`	Store half-word	memory[s3+22]$_{15:0}$ = t3$_{15:0}$
`sb t4, 5(s4)`	Store byte	memory[s4+5]$_{7:0}$ = t4$_{7:0}$
`beq s1, s2, L1`	Branch if equal	if (s1==s2), PC = L1
`bne t3, t4, Loop`	Branch if not equal	if (s1!=s2), PC = Loop
`blt t4, t5, L3`	Branch if less than	if (t4 < t5), PC = L3
`bge s8, s9, Done`	Branch if not equal	if (s8>=s9), PC = Done
`li s1, 0xABCDEF12`	Load immediate	s1 = 0xABCDEF12
`la s1, A`	Load address	s1 = Variable A's memory address (location)
`nop`	Nop	no operation
`mv s3, s7`	Move	s3 = s7
`not t1, t2`	Not (Invert)	t1 = ~t2
`neg s1, s3`	Negate	s1 = -s3
`j Label`	Jump	PC = Label
`jal L7`	Jump and link	PC = L7; ra = PC + 4
`jr s1`	Jump register	PC = s1

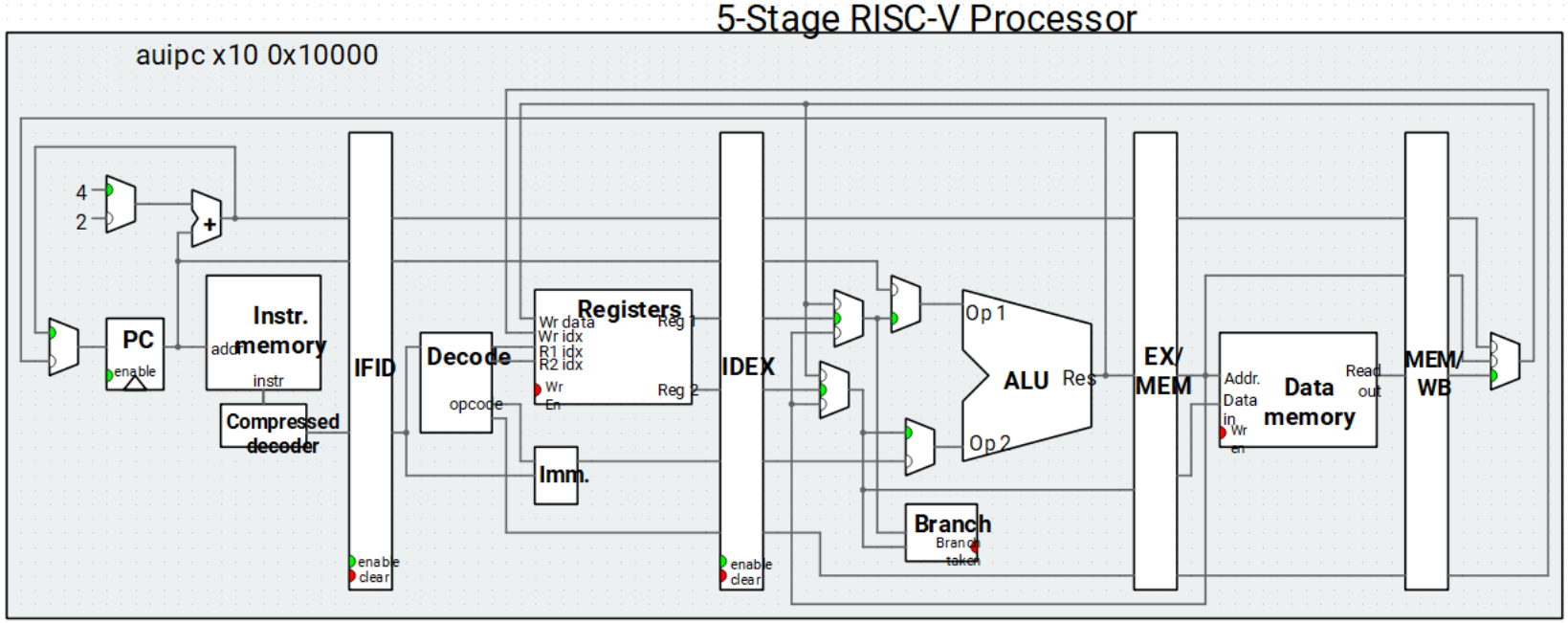
Textbook

Instruction Set Architecture (ISA)
Quantitative design and analysis
Memory Hierarchy
Instruction level parallelism
Data level parallelism
Thread-Level Parallelism
Domain specific architectures



Tools: Simulation

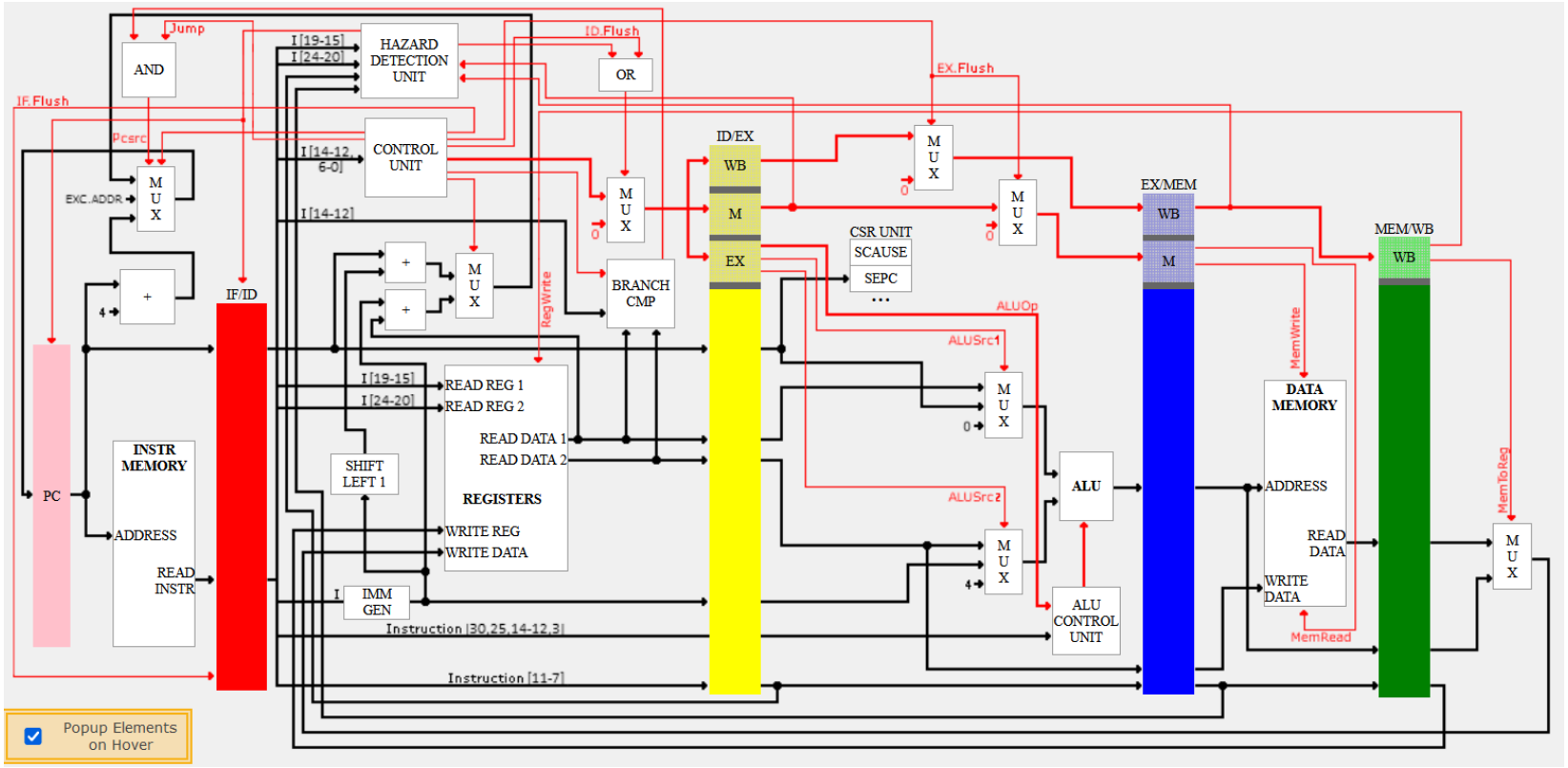
- Learn the concepts on simulation (no hardware)
- Assess various strategies and try-out instructions.
- Example: Ripes simulator (Standalone software)



https://github.com/mortbopet/Ripes/blob/master/docs/introduction.md

Tools : Simulation

- Example: WebRiscV Simulator (Web Based)



https://webriscv.dii.unisi.it/index.php

Tools: Optimize Processor Hardware

- Build, and test using Verilog, Try it in FPGAs
- https://dms.uom.lk/s/PXs7rncgRL4GgXF
- Edx Course is available https://www.edx.org/course/computer-architecture-with-an-industrial-risc-v-core

Tools: Build and Optimize

- Hardware
  - Verilog
- Test software

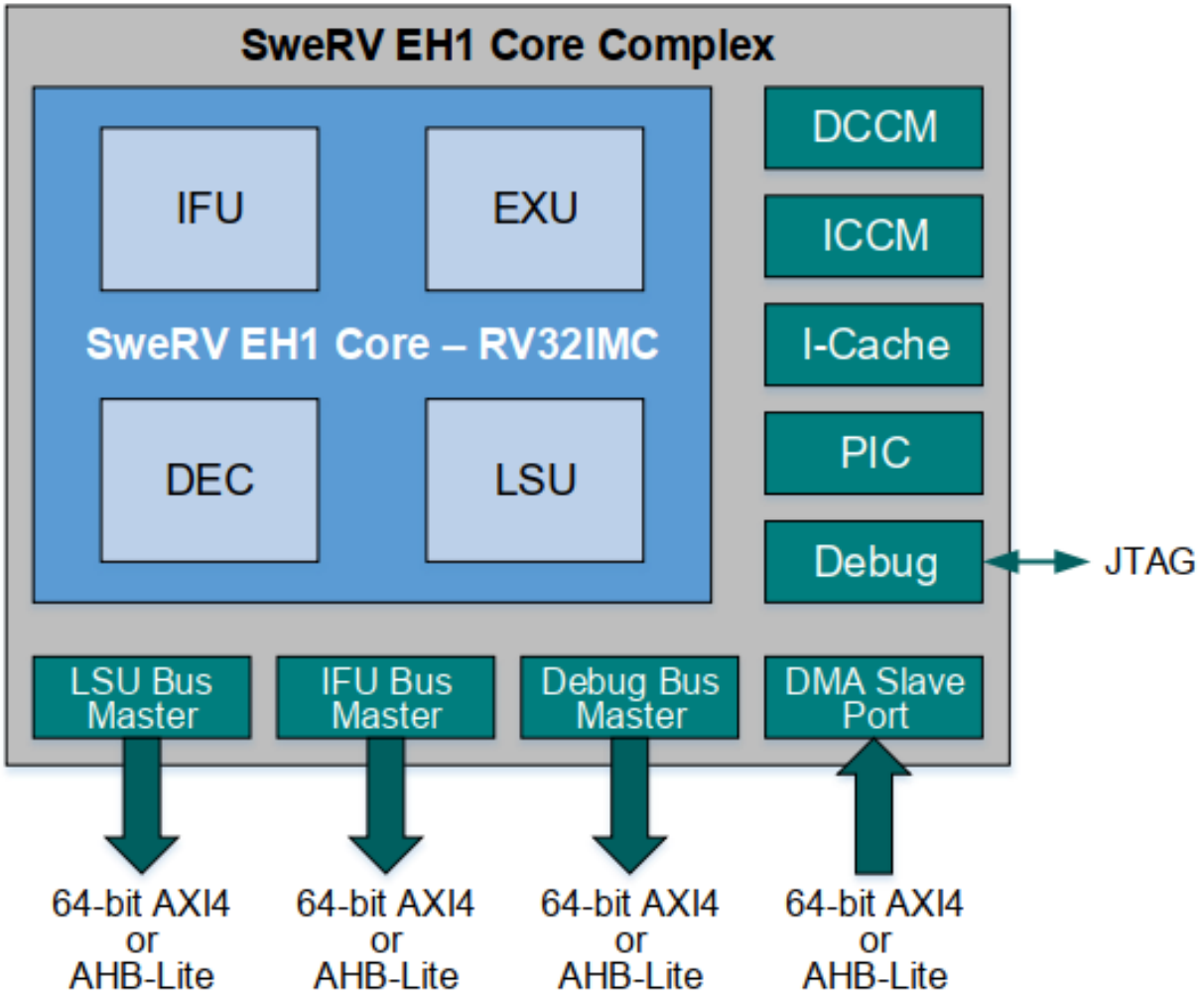


Figure 1-1 SweRV EH1 Core Complex

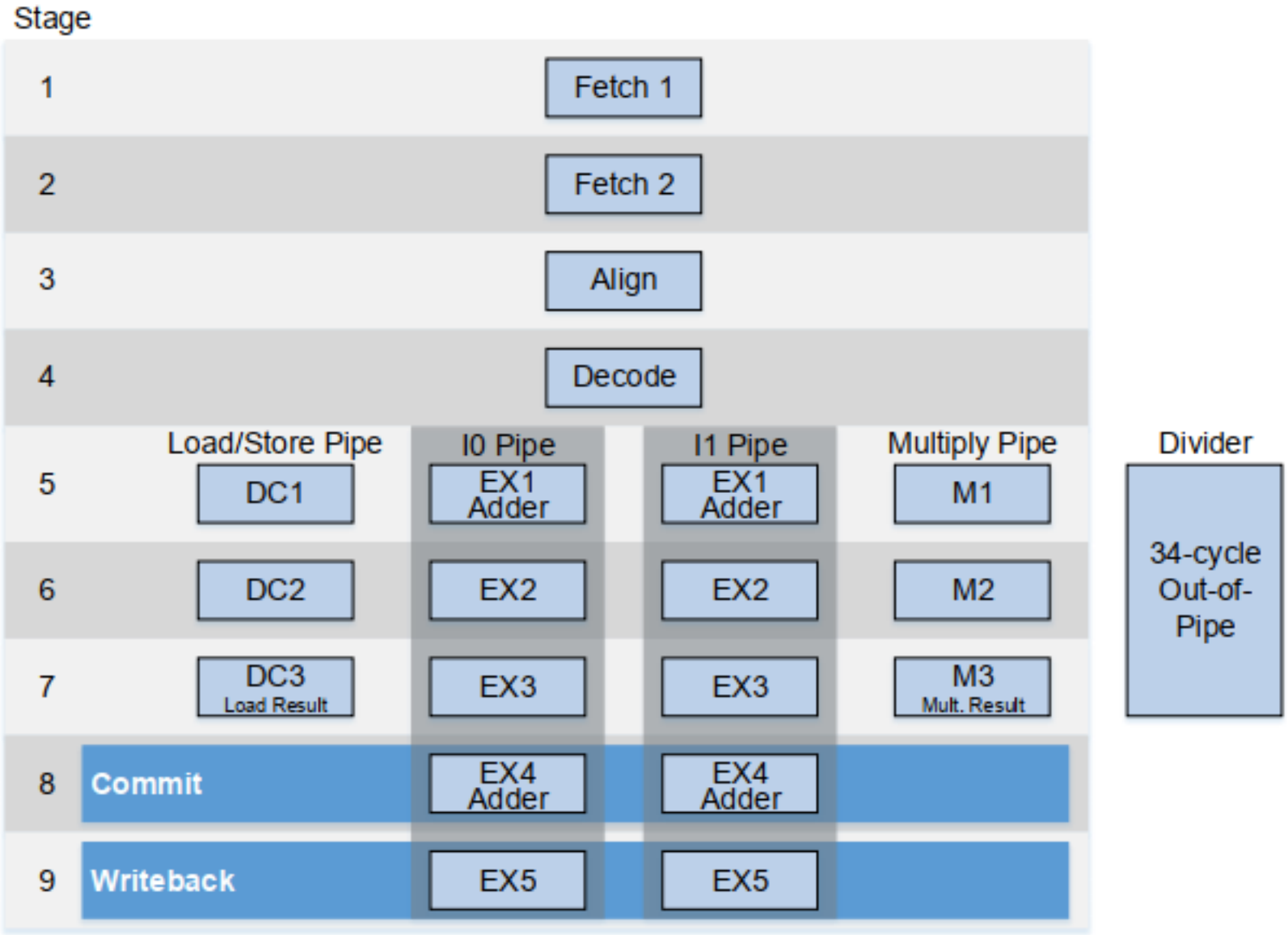


Figure 1-2 SweRV EH1 Core Pipeline

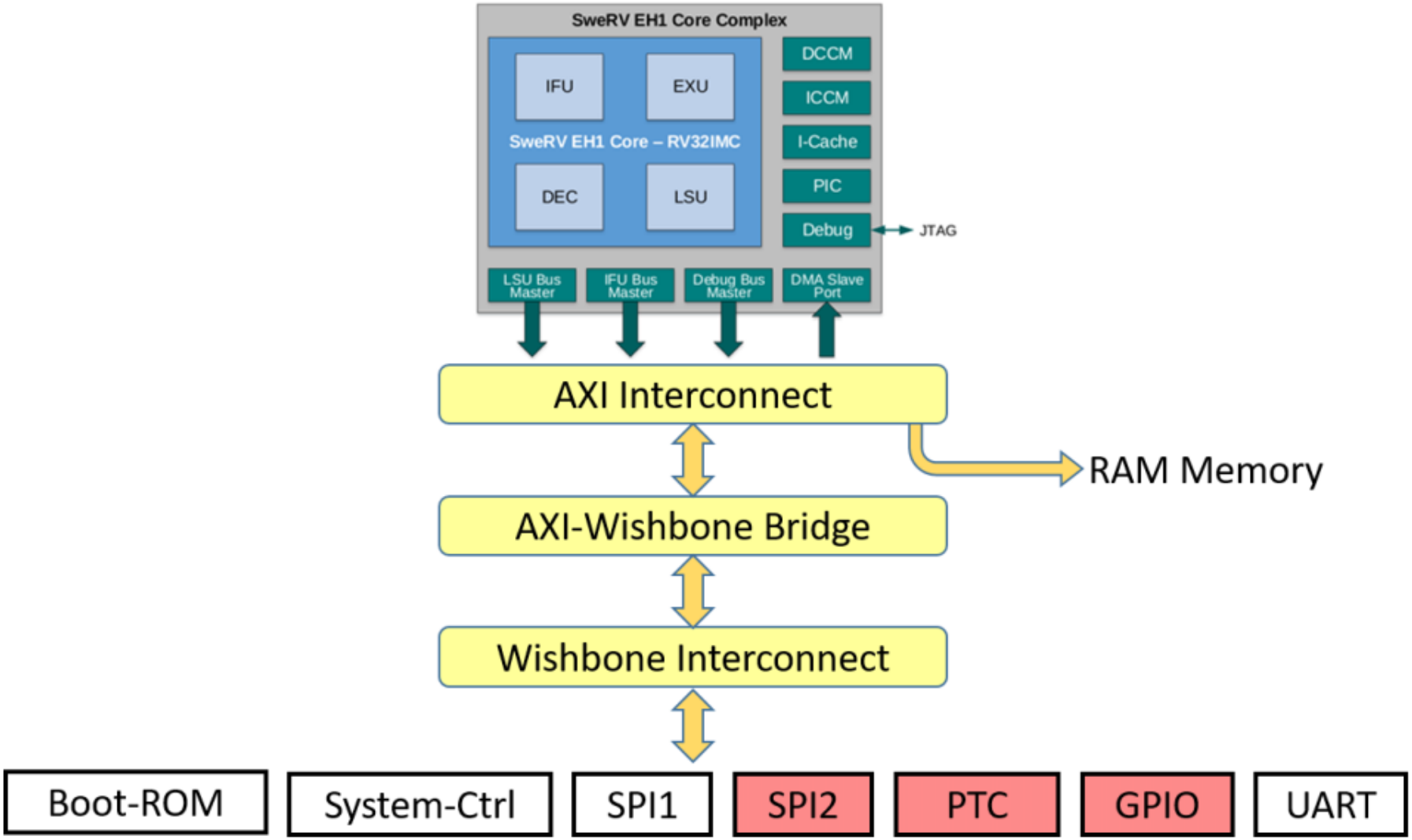


Figure 21. SweRVolfX (SweRVolf eXtended with 4 new peripherals) System on Chip

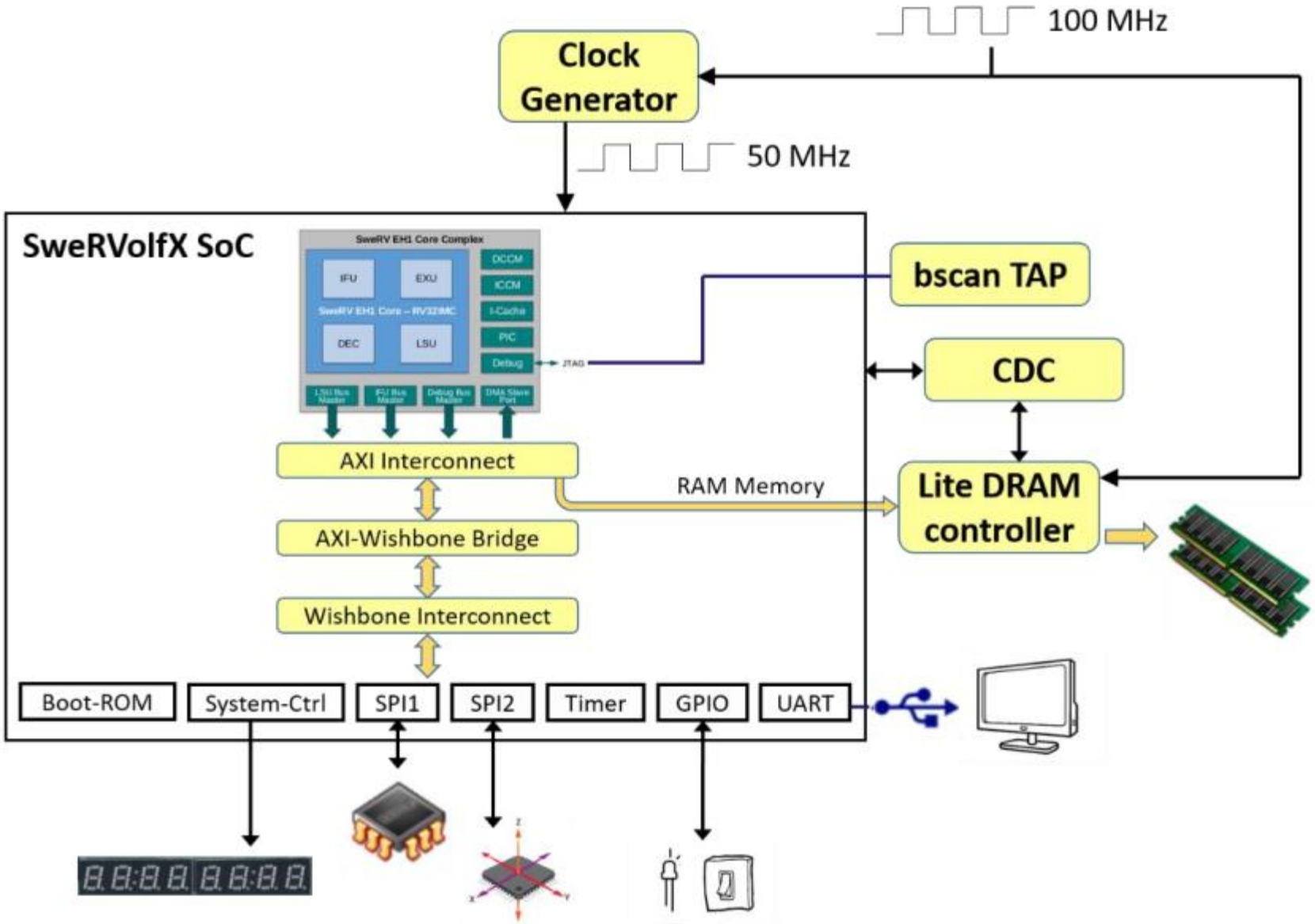


Figure 25. RVfpgaNexys