

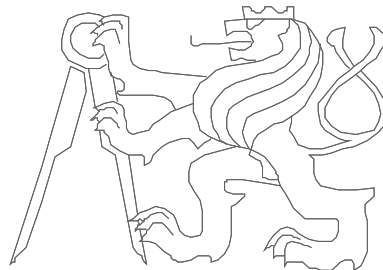
Computer Architectures

<https://cw.fel.cvut.cz/wiki/courses/b35apo/start>

Central Processing Unit (CPU)

Pavel Píša, Petr Štěpán, Richard Šusta,
Michal Štepanovský, Miroslav Šnorek

The lecture is based on A0B36APO lecture. Some parts are inspired by the book Paterson, D., Henessy, V.: Computer Organization and Design, The HW/SW Interface. Elsevier, ISBN: 978-0-12-370606-5 and it is used with authors' permission.



Czech Technical University in Prague, Faculty of Electrical Engineering

English version partially supported by:

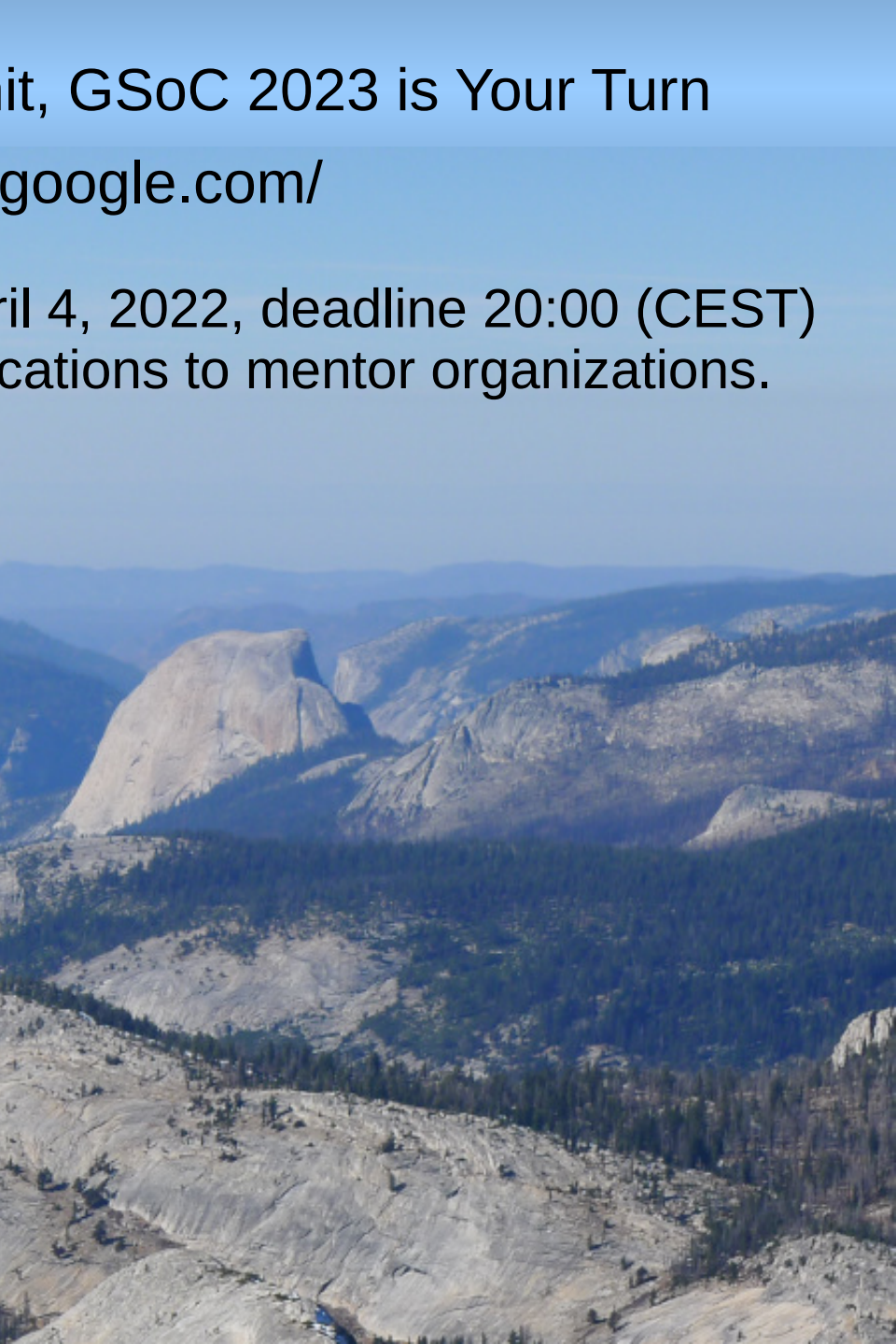
European Social Fund Prague & EU: We invests in your future.



GSOC 2016 Mentor Summit, GSoC 2023 is Your Turn

<https://summerofcode.withgoogle.com/>

Students Apply March 20 – April 4, 2022, deadline 20:00 (CEST)
Register and submit your applications to mentor organizations.



QtMips and QtRVSim – Origin and Development

- **MipsIt** used in past for Computer Architecture course at the Czech Technical University in Prague, Faculty of Electrical Engineering

- Diploma theses of Karel Kočí mentored by Pavel Píša

Graphical CPU Simulator with Cache Visualization

<https://dspace.cvut.cz/bitstream/handle/10467/76764/F3-DP-2018-Koci-Karel-diploma.pdf>

- Switch to QtMips in the 2019 summer semester
- Fixes, extension and partial internals redesign by Pavel Píša
- Switch to RISC-V architecture in 2022. Main work by Jakub Dupák 2021

Graphical RISC-V Architecture Simulator - Memory Model and Project Management

- <https://dspace.cvut.cz/bitstream/handle/10467/94446/F3-BP-2021-Dupak-Jakub-thesis.pdf>

- Alternatives:

- RARS: IDE with detailed help and hints

<https://github.com/TheThirdOne/rars>

- EduMIPS64: 1x fixed and 3x FP pipelines

<https://www.edumips.org/>

QtRVSim – Download

- Windows, Linux, Mac

<https://github.com/cvut/qtrvsim/releases>

- Ubuntu

<https://launchpad.net/~qtrvsimteam/+archive/ubuntu/ppa>

- Suse, Fedora and Debian

<https://software.opensuse.org/download.html?project=home%3Ajdupak&package=qtrvsim>

- Online version

<https://comparch.edu.cvut.cz/qtrvsim/app>

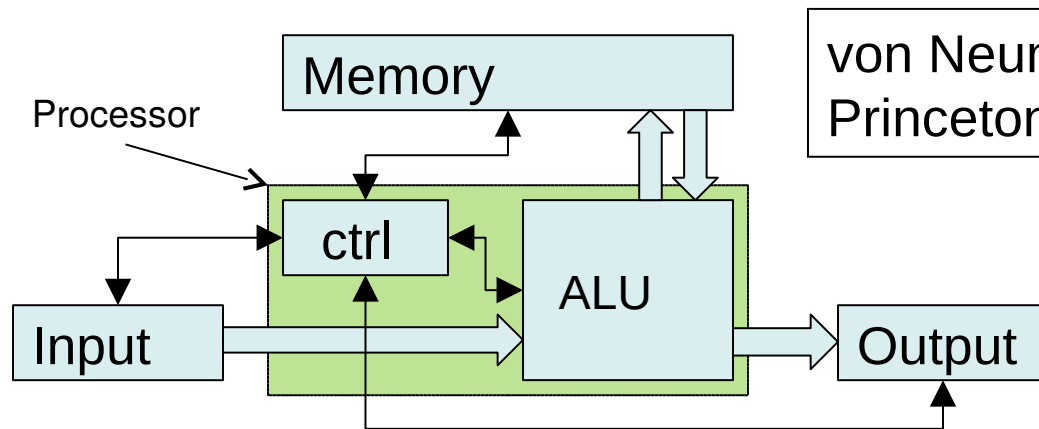
- RISC-V International talk for Academic and Training SIG

https://youtu.be/J6AcPZZ_ISg

- FOSDEM 2023 QtRvSim talk – Record of Interactive Session

https://fosdem.org/2023/schedule/event/rv_qtrvsim/

John von Neumann Computer Block Diagram



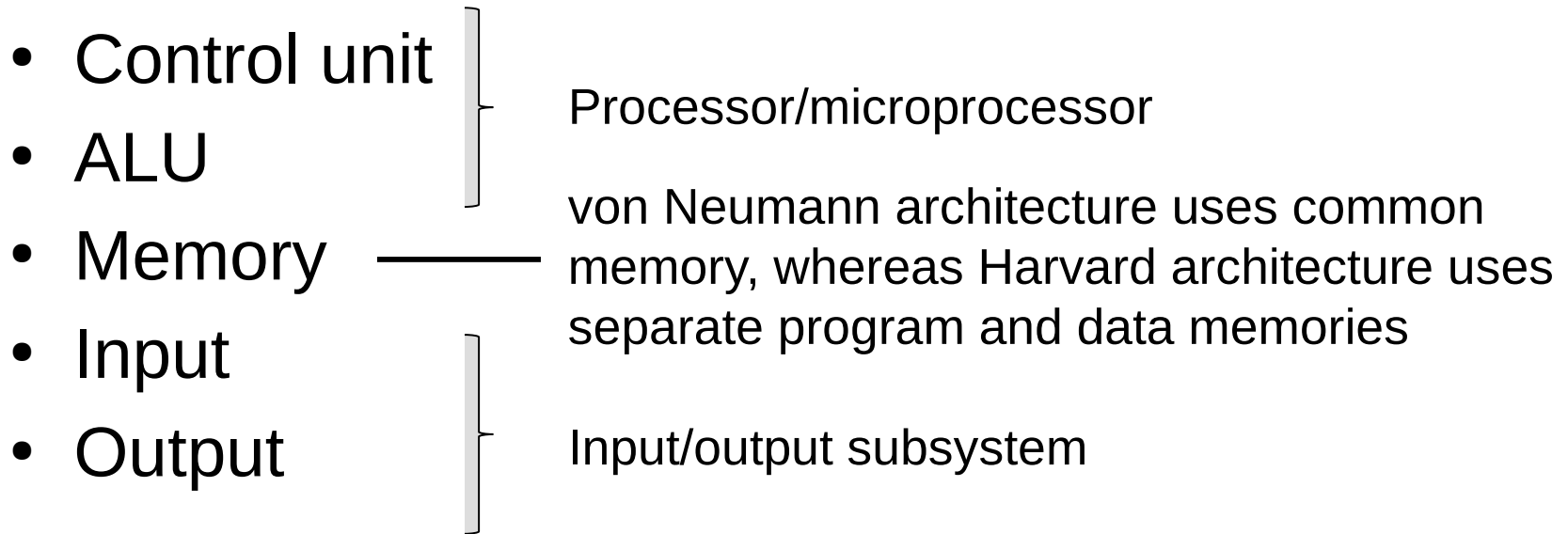
von Neumann's computer architecture
Princeton Institute for Advanced Studies



**28. 12. 1903 -
8. 2. 1957**

- 5 functional units – control unit, arithmetic logic unit, memory, input (devices), output (devices)
- An computer architecture should be independent of solved problems. It has to provide mechanism to load program into memory. The program controls what the computer does with data, which problem it solves.
- Programs and results/data are stored in the same memory. That memory consists of a cells of same size and these cells are sequentially numbered (address).
- The instruction which should be executed next, is stored in the cell exactly after the cell where preceding instruction is stored (exceptions branching etc.).
- The instruction set consists of arithmetics, logic, data movement, jump/branch and special/control instructions.

Computer based on von Neumann's concept



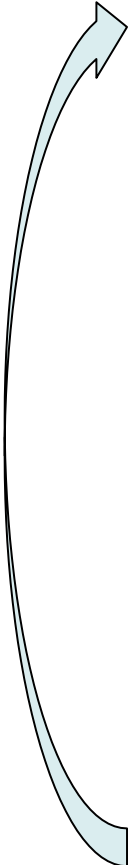
The control unit is responsible for control of the operation processing and sequencing. It consists of:

- registers – they hold intermediate and programmer visible state
- control logic circuits which represents core of the control unit (CU)

The most important registers of the control unit

- PC (Program Counter)
holds address of a recent or next instruction to be processed
- IR (Instruction Register)
holds the machine instruction read from memory
- Another usually present registers
 - General purpose registers (GPRs)
may be divided to address and data or (partially) specialized registers
 - SP (Stack Pointer) – points to the top of the stack; (The stack is usually used to store local variables and subroutine return addresses)
 - PSW (Program Status Word)
 - IM (Interrupt Mask)
 - Optional Floating point (FPRs) and vector/multimedia regs.

The main instruction cycle of the CPU

1. Initial setup/reset – set initial PC value, PSW, etc.
 2. Read the instruction from the memory
 - PC → to the address bus
 - Read the memory contents (machine instruction) and transfer it to the IR
 - $PC+I \rightarrow PC$, where I is length of the instruction
 3. Decode operation code (opcode)
 4. Execute the operation
 - compute effective address, select registers, read operands, pass them through ALU and store result
 5. Check for exceptions/interrupts (and service them)
 6. Repeat from the step 2
- 

Compilation: C → Assembler → Machine Code

```
/* ffs as log2(x)*/  
int x = 157;  
int y = 0;  
  
while(x != 0)  
{  
    x = x / 2;  
    y = y + 1;  
}
```

```
_start: addi a0, zero, 157 // int x = 157;  
        addi t1, zero, -1 // int y = 0;  
        beq a0, zero, done // while(x != 0) {  
loop:   srli a0, a0, 1 // x = x / 2;  
        addi t1, t1, 1 // y = y + 1;  
        bne a0, zero, loop //}  
done:   addi a0, t1, 0
```

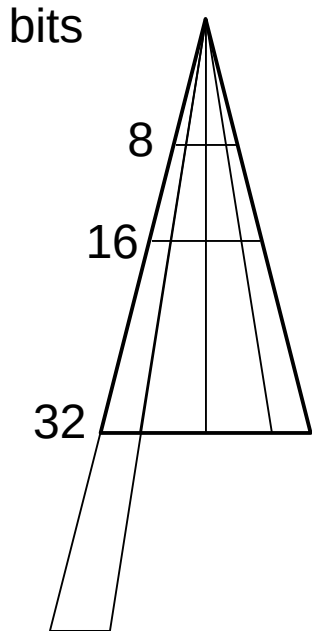
```
finis 0x00000200 09d00513 addi x10, x0, 157  
      0x00000204 fff00313 addi x6, x0, -1  
      0x00000208 00050863 beq x10, x0, 0x218  
      0x0000020c 00155513 srli x10, x10, 0x1  
      0x00000210 00130313 addi x6, x6, 1  
      0x00000214 fe051ce3 bne x10, x0, 0x20c  
      0x00000218 00030513 addi x10, x6, 0  
      0x0000021c 00100073 ebreak
```

<https://gitlab.fel.cvut.cz/b35apo/stud-support/>

[seminaries/qtrvsim/ffs-as-log2](#)

Instruction Encoding Constrains

Instruction



reserve
part of
code
space for
longer
operations

- Encode 256 combinations in 8-bits
- Opcode and address to directly operate on megabytes of ram does not fit
- Use some limited number of fast accessible registers or use stack concept
- 8 registers, 3 bits to encode, for two operand operations ($a += b$, $\text{mem}[a] = b$), 6 bits to select registers → only 4 operations in total
- 16 bit (65536), 16 registers, 256 two operands or 16 three operands ($4 + 3 * 4$ bits)
- 32 bit, 32 registers, three operands ($17 + 3 * 5$)
- But immediate for arithmetic and address usually required (CISC followup words, RISC uses limited ranges and some other mechanism)

RISC-V – Instruction Length Encoding

xxxxxxxxxxxxxxxxaa 16-bit (aa \neq 11)

xxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxbbb11 32-bit (bbb \neq 111)

· · ·xxxx xxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxx011111 48-bit

· · ·xxxx xxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxx01111111 64-bit

· · ·xxxx xxxxxxxxxxxxxxxxxxxxxx xnnnxxxx11111111 (80+16*nnn)-bit, nnn \neq 11:

· · ·xxxx xxxxxxxxxxxxxxxxxxxxxx x111xxxx11111111 Reserved for \geq 192-bits

Address:

base+4

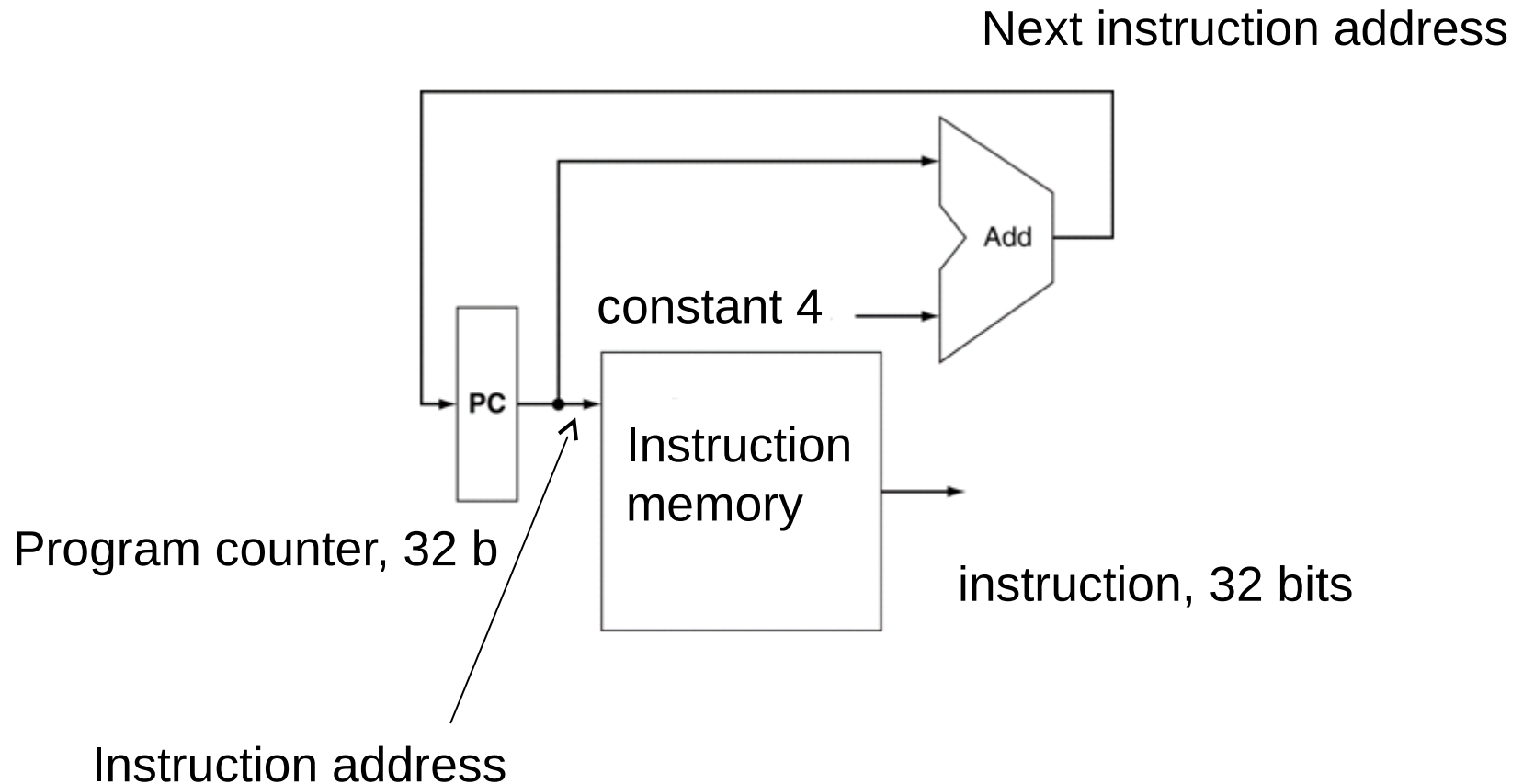
base+2

base

RISC-V Processor Registers

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	–
x4	tp	Thread pointer	–
x5	t0	Temporary/alternate link register	Caller
x6–7	t1– 2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
pc	pc	Program Counter	
f0–31		Floating point	
		Machine control and status	

Hardware realization of basic (main) CPU cycle



The goal of this lecture

- To understand the implementation of a simple computer consisting of CPU and separated instruction and data memory
- Our goal is to implement following instructions:
 - Read and write a value from/to the data memory
lw – load word, **sw** – store word
 - Arithmetic and logic instructions: **add**, **sub**, **and**, **or**, **slt**
Immediate variants: **addi**, **ori**, upper bits loads **lui**, **auipc**
 - Program flow change/jump instruction **beq**
 - Subroutine call **jal**, **jalr** (provides even return from subroutine **jr ra**)
- CPU will consist of control unit and ALU (data path).
- Notes:
 - The implementation will be minimal (single cycle CPU – all operations processed in the single step/clock period)
 - The lecture 5 focuses on more realistic pipelined CPU implementation

Subset of RISC-V Instructions to Implement

Instruction	Executed operation
lw rd, imm12(rs1)	[rd] \leftarrow Mem[[rs1] + imm12];
sw rs2, imm12(rs1)	Mem[[rs1] + imm12] \leftarrow [rs2];
addi rd, rs1, imm12	[rd] \leftarrow [rs1] + imm12;
add rd, rs1, rs2	[rd] \leftarrow [rs1] + [rs2];
sub rd, rs1, rs2	[rd] \leftarrow [rs1] - [rs2];
and rd, rs1, rs2	[rd] \leftarrow [rs1] & [rs2];
or rd, rs1, rs2	[rd] \leftarrow [rs1] [rs2];
slt rd, rs1, rs2	[rd] \leftarrow [rs1] < [rs2];
beq rs1, rs2, imm12	if [rs1] == [rs2] go to [PC]+(imm12<<1); else go to [PC]+4;
jal rd, imm20	[rd] \leftarrow [PC]+4; go to [PC]+(imm20<<1);
jalr rd, rs1, imm12	[rd] \leftarrow [PC]+4; go to [rs1]+imm12;
lui rd, imm20	[rd] \leftarrow imm20<<12
auipc rd, imm20	[rd] \leftarrow PC+(imm20<<12)

Remark, all immediate operands are signed

The MIPS instruction format and instruction types

- The three types of the instructions are considered:

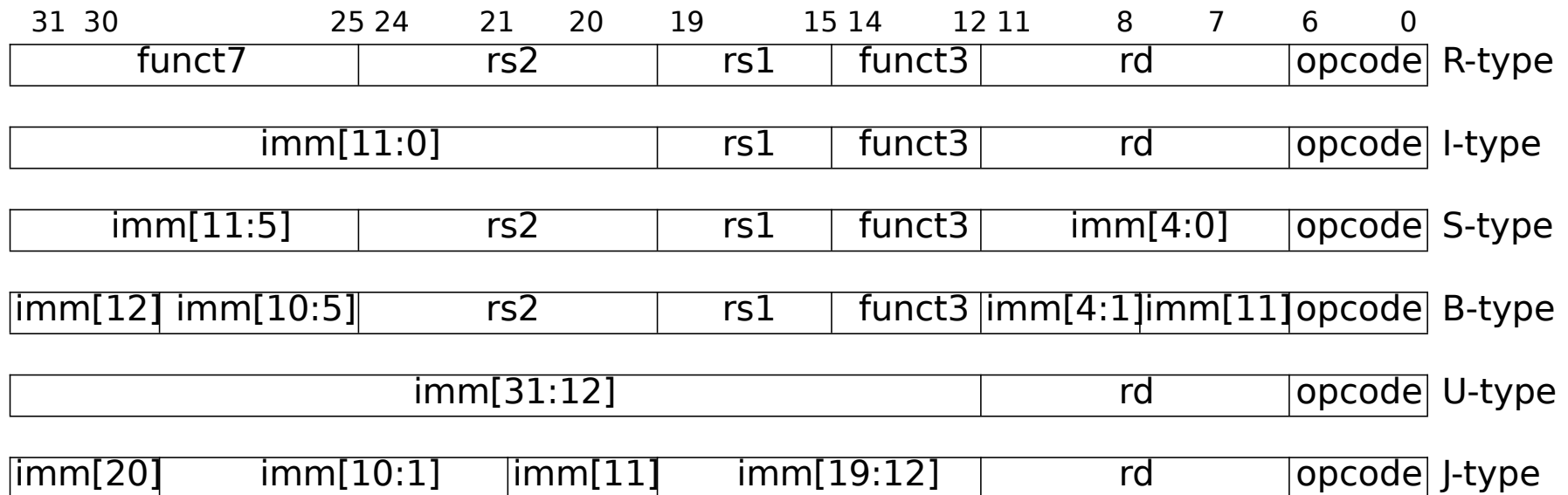
Type	31... 0					
R	opcode(6), 31:26	rs(5), 25:21	rt(5), 20:16	rd(5), 15:11	shamt(5)	funct(6), 5:0
I	opcode(6), 31:26	rs(5), 25:21	rt(5), 20:16	immediate (16), 15:0		
J	opcode(6), 31:26	address(26), 25:0				

- the R type instructions → opcode=000000, funct – operation
- rs – source, rd – destination, rt – source/destination
- shamt – for shift operations, immediate – direct operand

- 5 bits allows to encode 32 GPRs (\$0 is hardwired to 0/discard)

The RISC-V instruction format and instruction types

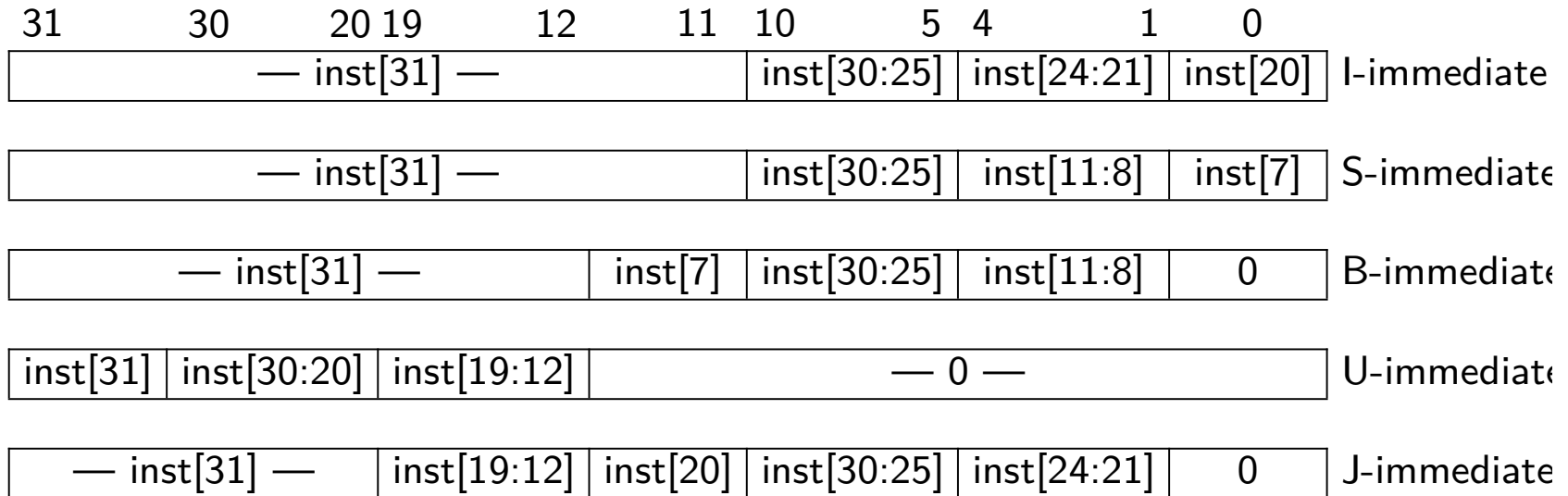
- The six basic formats of the instructions are considered:



- opcode – specify operation, func 3 and func 7 variant
- rs1 – source 1, rs2 – source 2, rd – destination
- 5 bits allows to encode 32 GPRs (\$0 is hardwired to 0/discard)

RISC-V Immediate Operand Encoding

- There are five options how to form immediate operand



- goal: keep same position in the instruction for corresponding bit (compare U and J, B and S, S and I)
- sign in instruction bit 31 (needs fan-out to drive multiple bits)

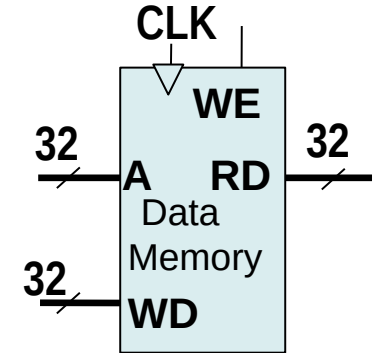
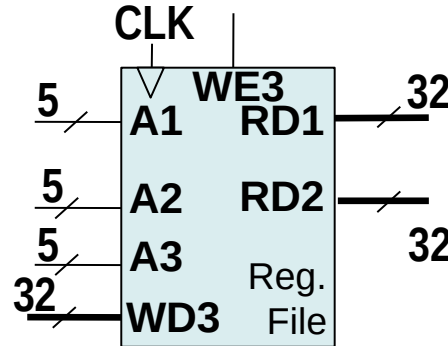
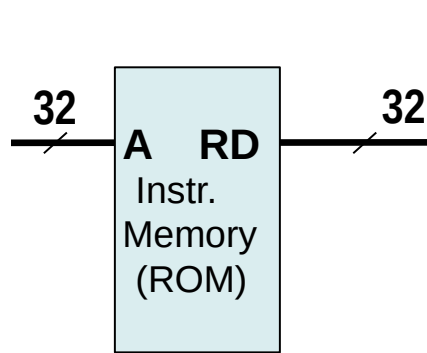
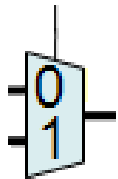
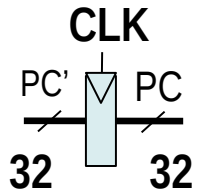
Opcode encoding

The primary selection of operation is **opcode**

Opcode	The group, operation	Actual operations for our subset
0110011	R-type (see funct7 and funct3)	add, sub, slt, or, and
0010011	ALU-imm (see funct3)	addi
0000011	Memory load (funct3 = 010)	lw
0100011	Memory store (funct3)	sw
1100011	Branch (see funct3)	beq
1101111	jal	jal
1100111	jalr	jalr
0000111	lui	lui

func7	fun3	instruction/ALU operation
0000000	000	add
0100000	000	sub
0000000	010	slt
0000000	110	or
0000000	111	and

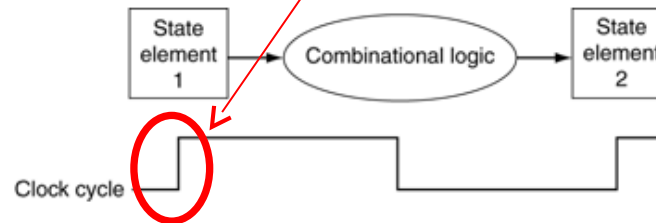
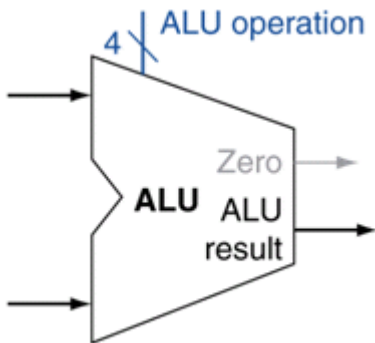
CPU building blocks



Write at the rising edge of CLK when WE = 1

Multiplexer

Read after "enough time" for data propagation

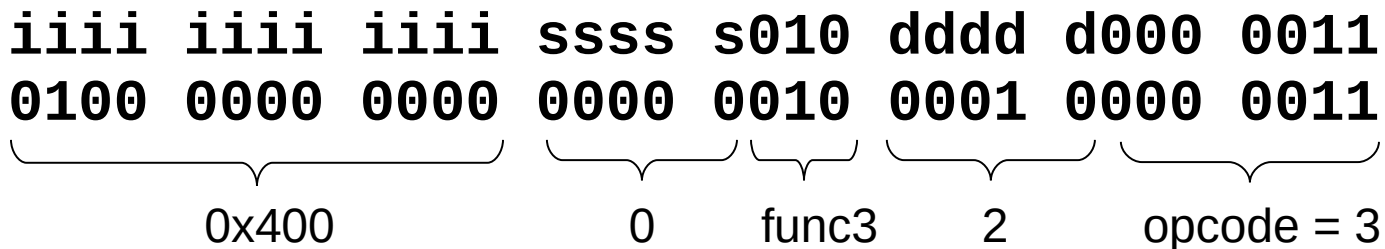


The load word instruction

lw – load word – load word from data memory into a register

Description	A word is loaded into a register from the specified address
Operation:	$[rd] \leftarrow \text{Mem}[[rs1] + \text{imm12}];$
Syntax:	lw rd, imm12(rs1)
Encoding:	iiii iiiiiiii ssss s010 dddd d000 0011

Example: Read word from memory address 0x400 into register number 2:
lw x2, 0x400(x0)



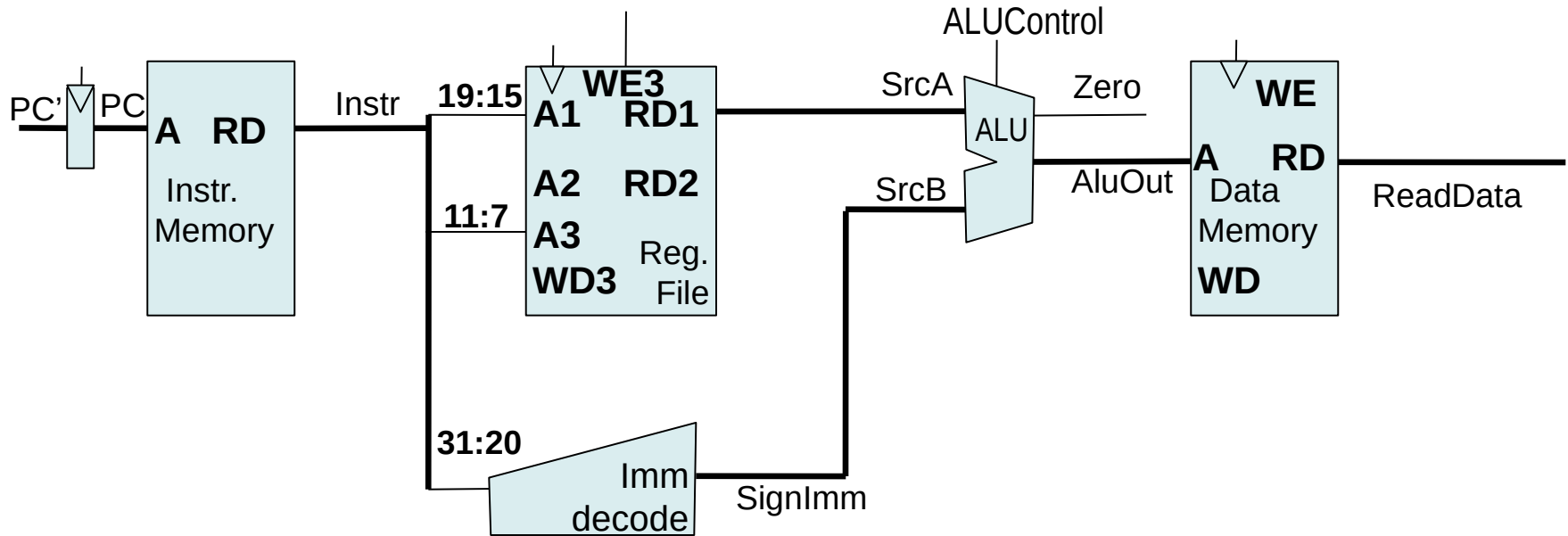
0x 40 00 21 03 – machine code for instruction lw x2, 0x400(x0)

Note: Register x0 is hardwired to the zero

Single cycle CPU – implementation of the load instruction

lw: rs1 – base address, imm12 – offset, rd – register where to store fetched data

I	imm(12), 31:20	rs1(5), 19:15	func3, 14:12	rd(5), 11:7	opcode(7), 6:0
---	----------------	---------------	--------------	-------------	----------------

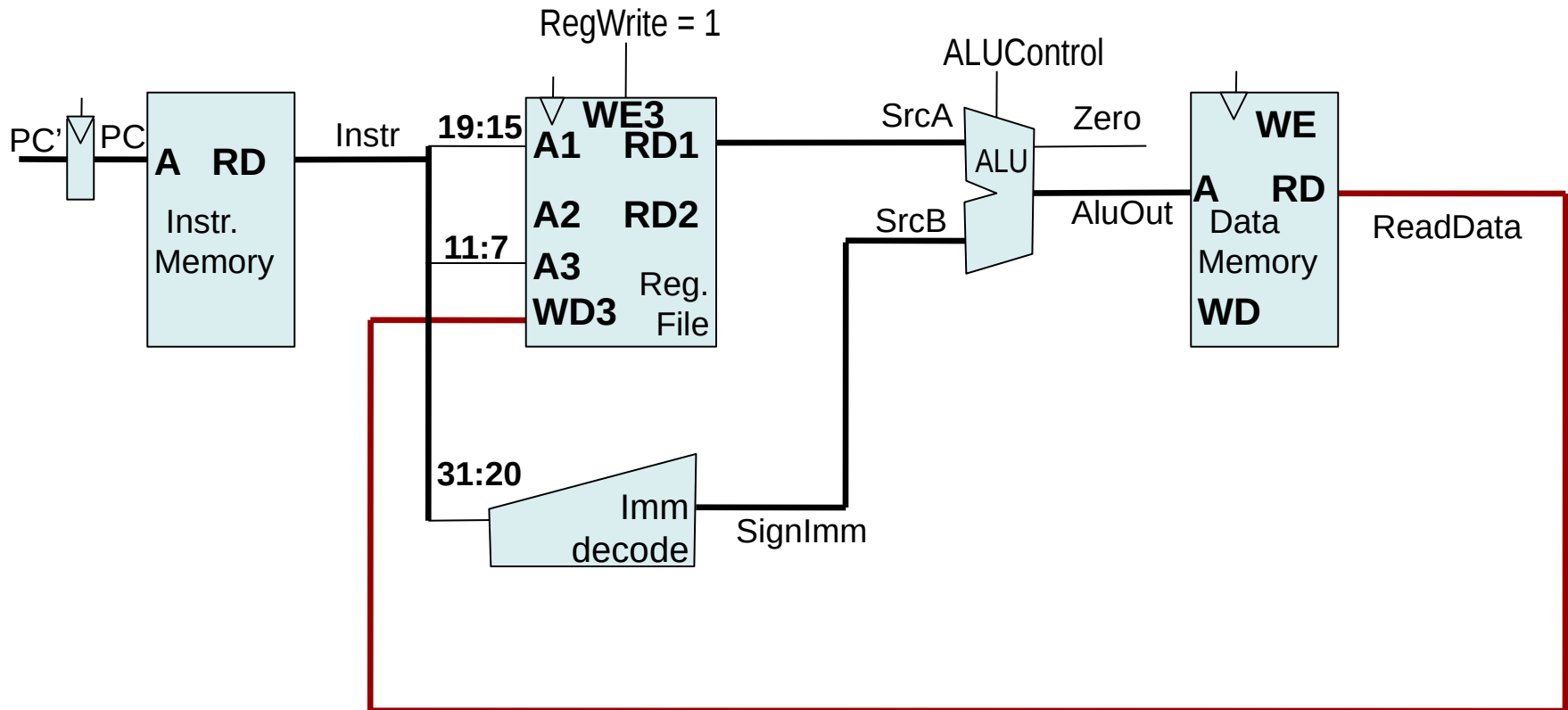


Single cycle CPU – implementation of the load instruction

Iw: rs1 – base address, imm12 – offset, rd – register where to store fetched data

I	imm(12), 31:20	rs1(5), 19:15	func3, 14:12	rd(5), 11:7	opcode(7), 6:0
---	----------------	---------------	--------------	-------------	----------------

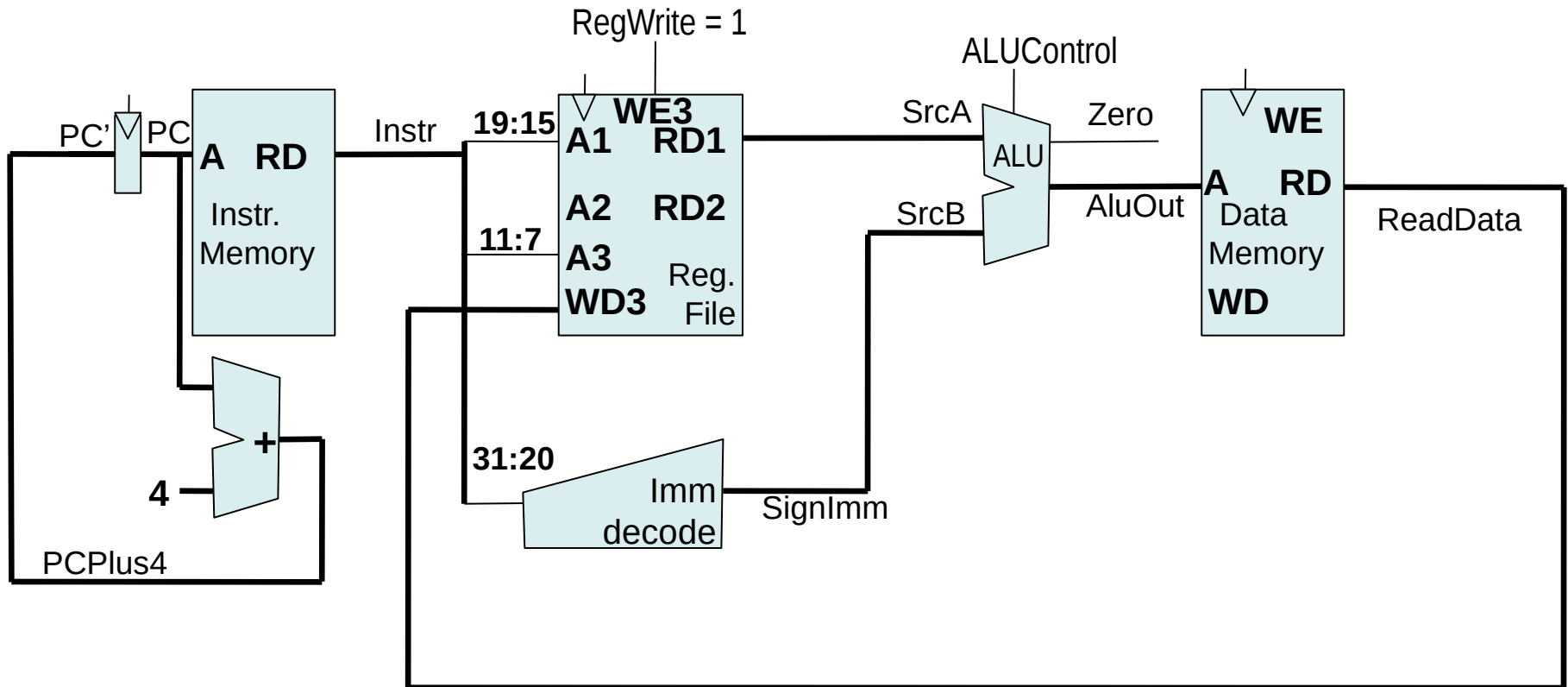
Write at the rising edge of the clock



Single cycle CPU – implementation of the load instruction

lw: rs1 – base address, imm12 – offset, rd – register where to store fetched data

I	imm(12), 31:20	rs1(5), 19:15	func3, 14:12	rd(5), 11:7	opcode(7), 6:0
---	----------------	---------------	--------------	-------------	----------------



QtMips – MIPS Architecture Emulator

The image shows the QtMips MIPS Architecture Emulator interface. Red arrows point from labels to specific parts of the software. The interface includes a menu bar (File, Mac OS, Windows, Help), a toolbar with icons for Load, Run, Single Step, and Make. A Registers window shows a table of MIPS registers (x0 to x31) with their current values. A Program window shows assembly code with instructions like 'addi x8, x11, 0' and 'bne x11, x0, 0x10390'. A CPU core view shows a detailed block diagram of the MIPS processor, including the Cache, Program Memory, Registers, ALU, and Data Cache. A Peripherals window shows LED RGB 1 and LED RGB 2, and a terminal window displays 'Hello world. Hello world. Hello world. Hello world.'.

Load

Run

Single Step

Make

Registers

x0/zero	0x0	x1/ra	0x100f8	x2/sp	0x13c98	x3/gp	0x13c18	x4/tp	0x0	x5/t0	0x0	x6/t1	0x11061	x7/t2	0x0
x8/s0	0x1100c	x9/s1	0x4	x10/a0	0x11000	x11/a1	0x0	x12/a2	0x29	x13/a3	0xa	x14/a4	0x20	x15/a5	0xffff
x16/s2	0x0	x17/s3	0x4	x18/s4	0xffff	x19/s5	0x11000	x20/s6	0x20	x21/s7	0x0	x22/s8	0x0	x23/s9	0x0
x24/s10	0x0	x25/s11	0x4	x26/s12	0x0	x27/s13	0x4	x28/t3	0x0	x29/t4	0xbffff00	x30/t5	0x0	x31/t6	0x0

Assembler

Editor

Code

Exceptions control

Terminal

Peripherals

Cache

Data memory

CPU core view

- single cycle
- pipelined

The store word instruction

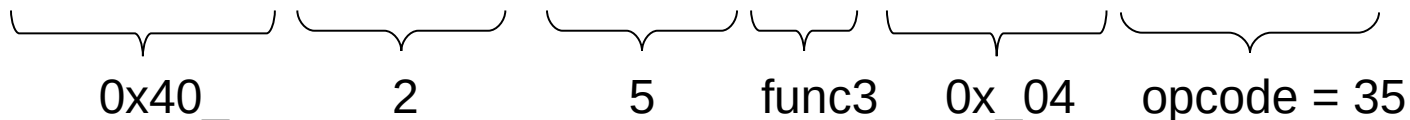
sw – **store word** – store word in a register to data memory

Description	Stores a value in register rs2 to given address in memory.
Operation:	Mem[[rs1] + imm12] = [rs2]
Syntax:	sw rs2,imm12(rs1)
Encoding:	iiii iiit tttt ssss s010 iiii i000 0011

Example: Store word in register 2 to memory address computed as addition of value in register 5 and constant 0x404, bit symbol **t** used for rs2:

sw x2, 0x404(x5)

iiii iiit tttt ssss s010 iiii i010 0011
0100 0000 0010 0010 1010 0010 0010 0011

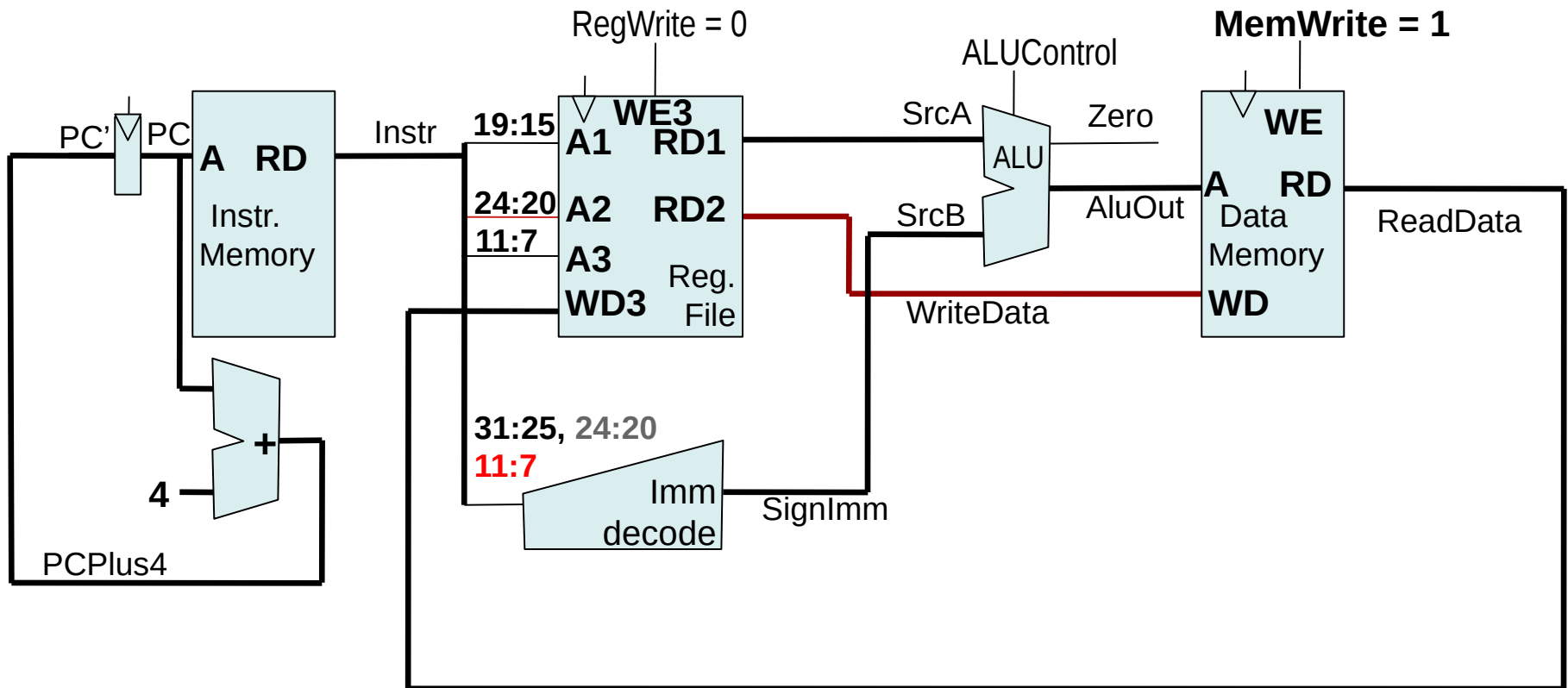


0x 40 22 a2 23 – machine code for instruction sw x2, 0x404(x5)

Single cycle CPU – implementation of the store instruction

sw: rs1 – base address, imm12 – offset, rs2 – select register to store into memory

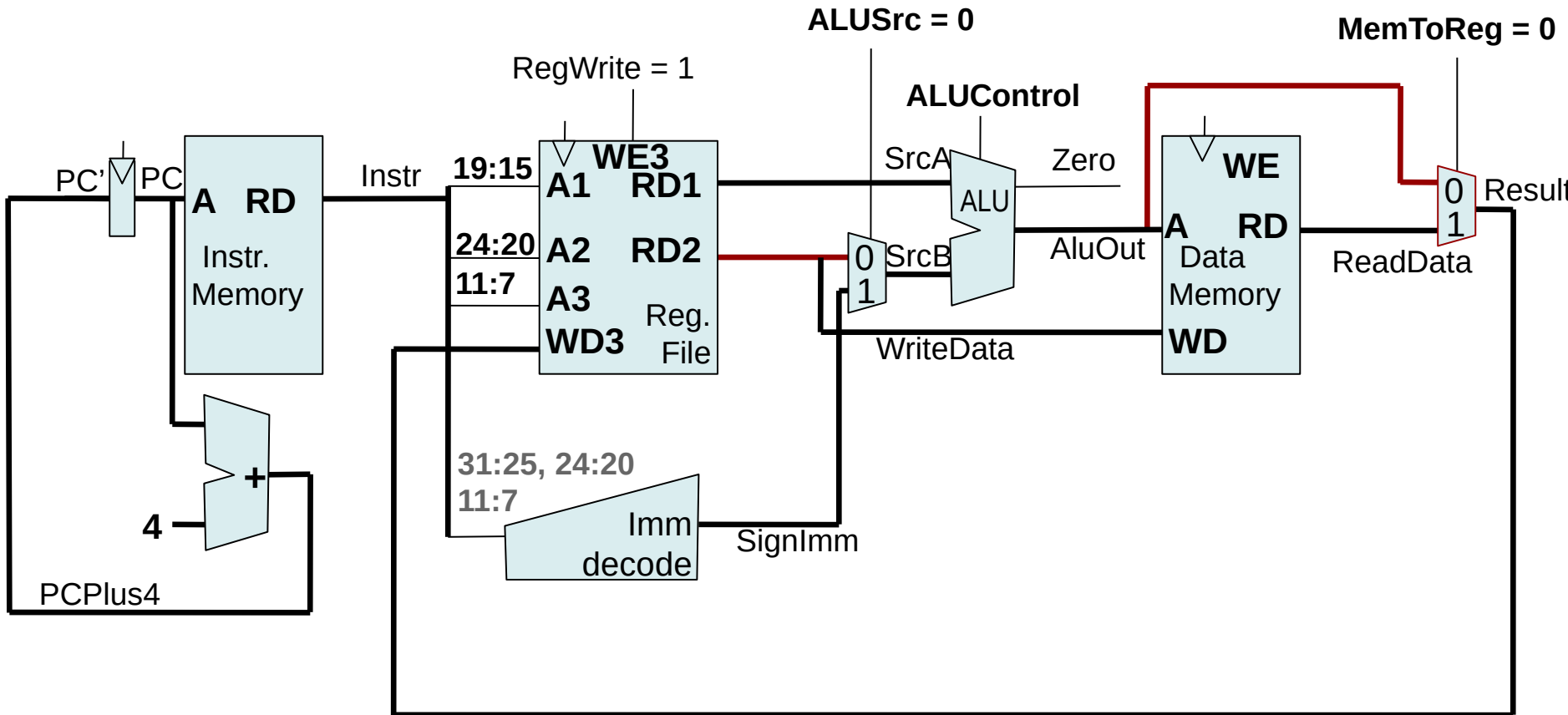
S	imm(12), 31:25, 11:7	rs2(5), 24:20	rs1(5), 19:15	func3, 14:12	opcode(7), 6:0
---	----------------------	---------------	---------------	--------------	----------------



Single cycle CPU – implementation of the **add** instruction

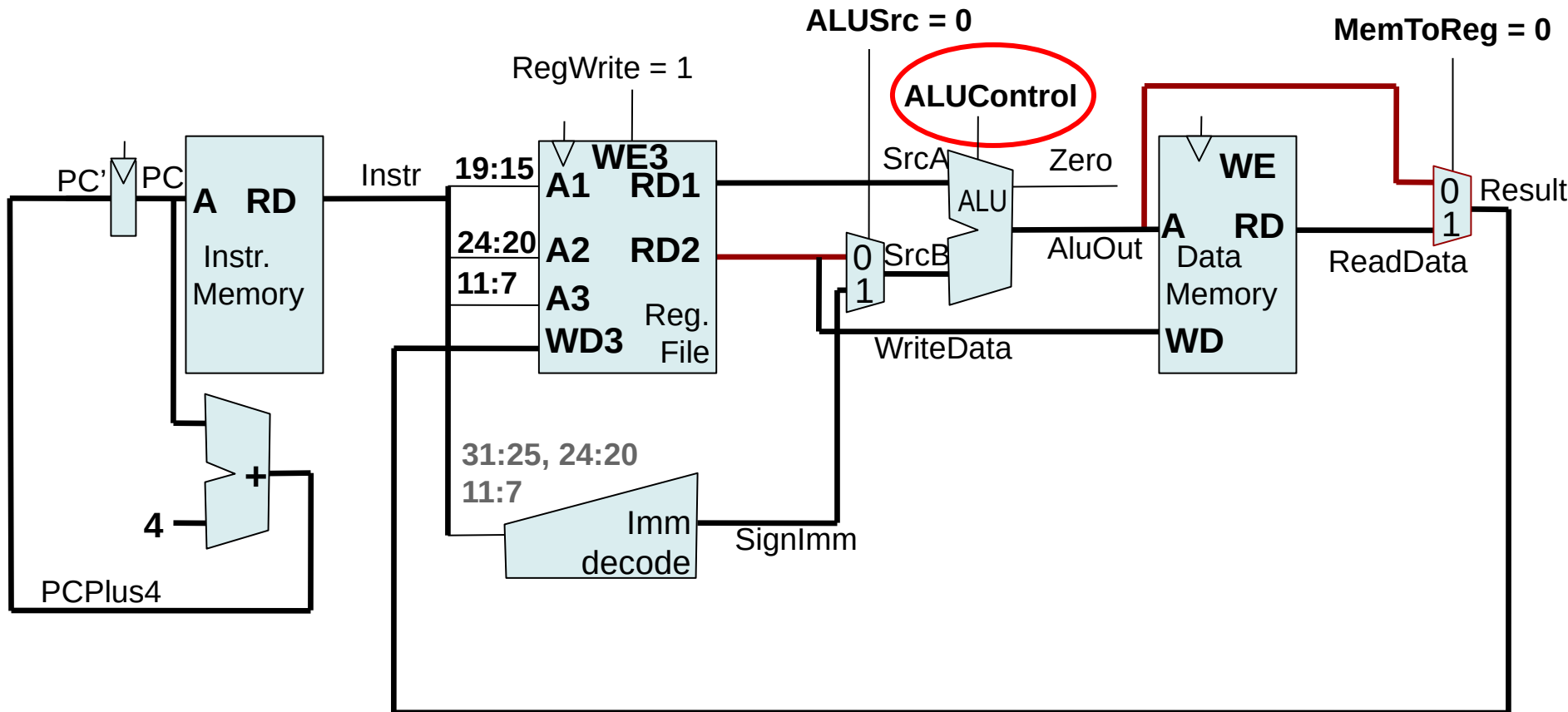
add: type R, rs, rt – source, rd – destination, funct – select ALU operation = add

R	func(7),31:25	rs2(5),24:20	rs1(5),19:15	func3, 14:12	rd(5),11:7	opcode(7), 6:0
---	---------------	--------------	--------------	--------------	------------	----------------



Single cycle CPU – **sub, and, or, slt**

Only difference is another ALU operation selection (ALUControl). The data path is the same as for **add** instruction

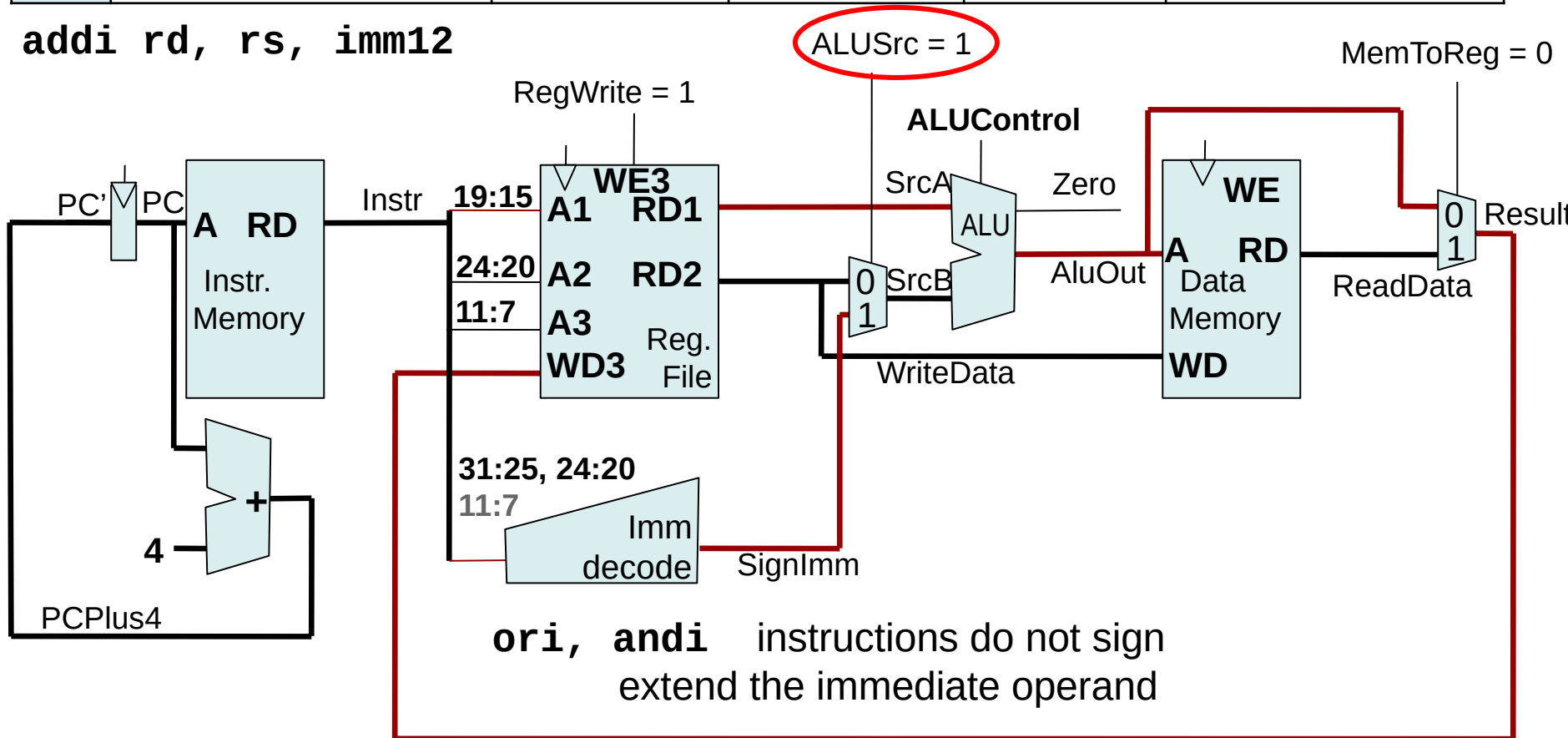


Single cycle CPU – **addi, ori, andi**

addi – add immediate; $[rd] = [rs1] + imm12$

I	imm(12), 31:20	rs1(5), 15:19	func3, 14:12	rd(5), 7:11	opcode(7), 6:0
---	----------------	---------------	--------------	-------------	----------------

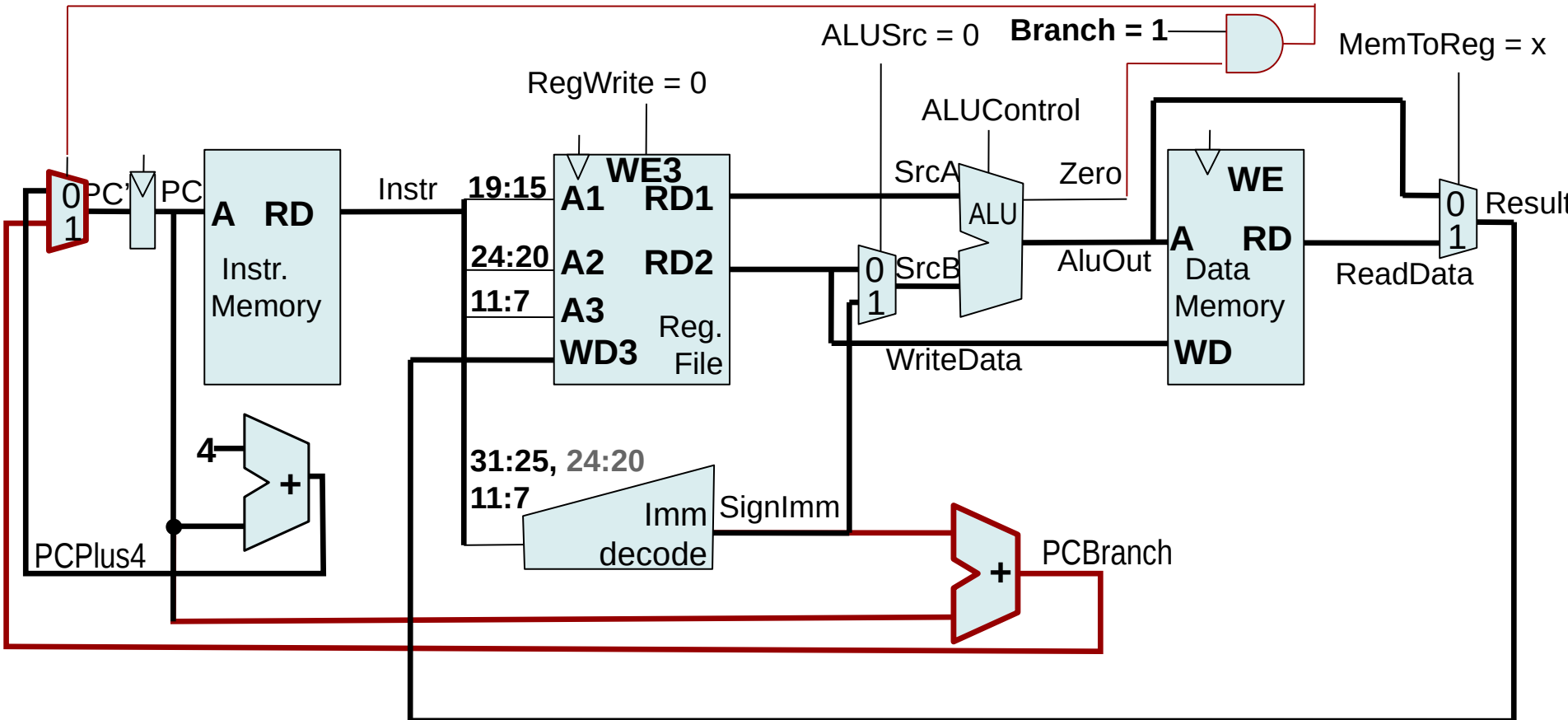
addi rd, rs, imm12



Single cycle CPU – implementation of **beq**

beq – branch if equal; imm–offset; $PC' = PC + \text{SignImm}$

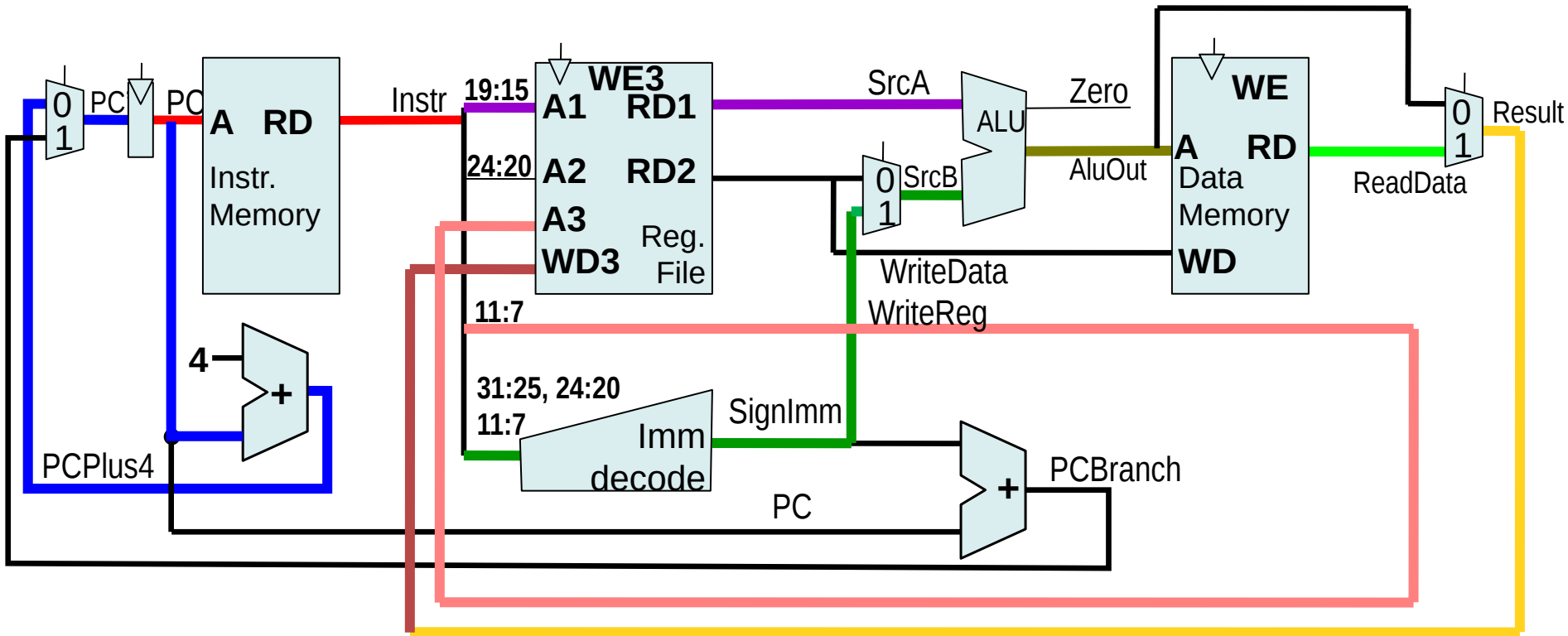
S	imm(12), 31:25, 11:7	rs2(5), 24:20	rs1(5), 19:15	func3, 14:12	opcode(7), 6:0
---	----------------------	---------------	---------------	--------------	----------------



Single cycle CPU – Throughput: $IPS = IC / T = IPC_{str} \cdot f_{CLK}$

- What is the maximal possible frequency of the CPU?
- It is given by latency on the critical path – it is $1w$ instruction in our case:

$$T_c = t_{PC} + t_{Mem} + t_{RFread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{RFsetup}$$



Single cycle CPU – Throughput: $IPS = IC / T = IPC_{str} \cdot f_{CLK}$

- What is the maximal possible frequency of the CPU?
- It is given by latency on the critical path – it is T_c instruction in our case:

$$T_c = t_{PC} + t_{Mem} + t_{RFread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{RFsetup}$$

Consider following parameters:

- $t_{PC} = 30 \text{ ns}$
- $t_{Mem} = 300 \text{ ns}$
- $t_{RFread} = 150 \text{ ns}$
- $t_{ALU} = 200 \text{ ns}$
- $t_{Mux} = 20 \text{ ns}$
- $t_{RFsetup} = 20 \text{ ns}$

Then $T_c = 1020 \text{ ns} \rightarrow f_{CLK} \text{ max} = 980 \text{ kHz}$,

$IPS = 980e3 = 980\,000$ instructions per second

Notes

- Remember the result, so you can compare it with result for pipelined CPU during lecture 4
- You should compare this with actual 30×10^9 IPS per core, i.e. total 128 300 MIPS for today high-end CPUs
- How many clever enhancements in hardware and programming/compiler are required for such advance!!!
- After this course you should see behind the first two hills on that road.

- We will continue with control unit implementation and its function

Control Unit Signals Generation

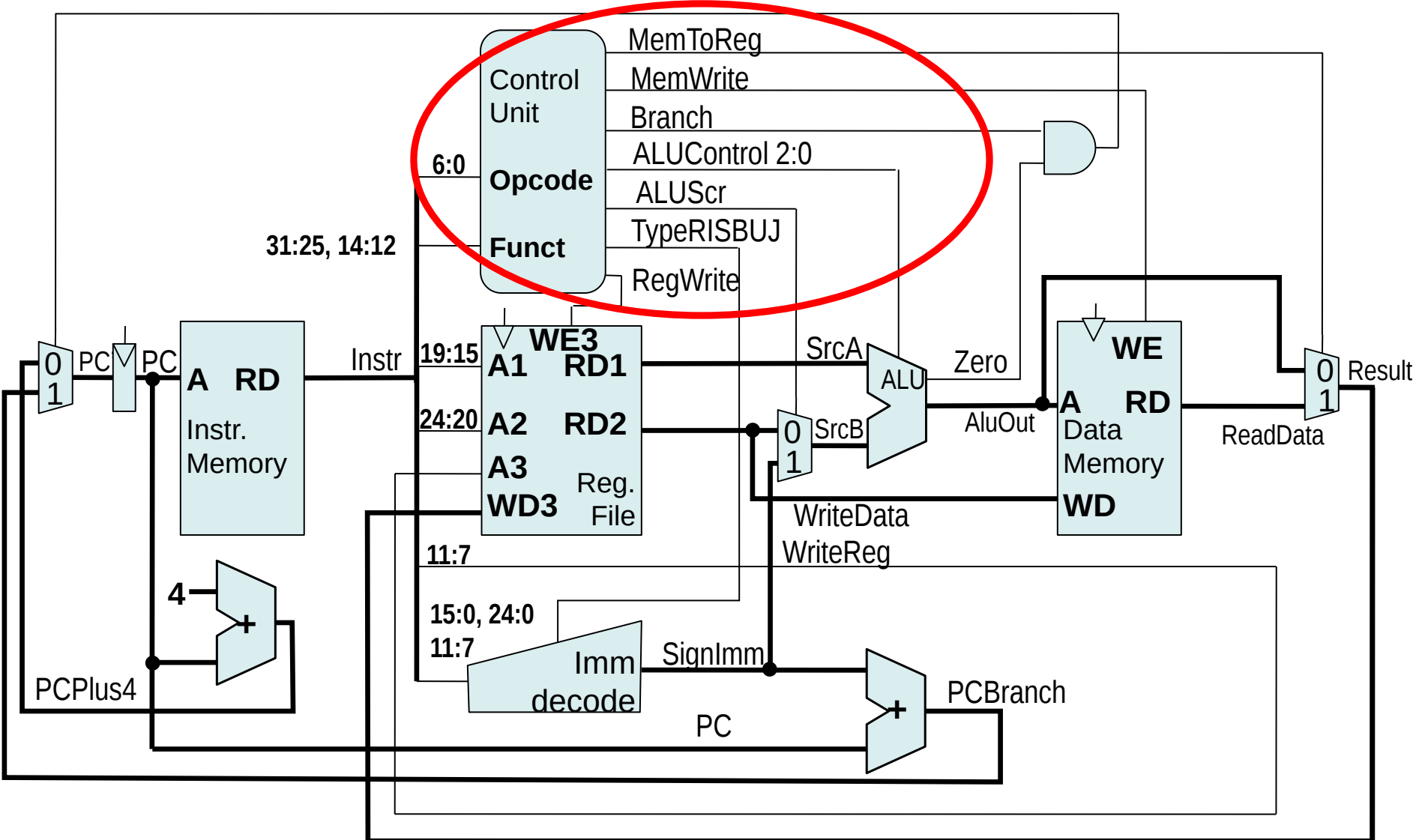
The control unit for single cycle RISC-V architecture can be simple combinatorial table.

For our simple case, only sequential blocks are PC, registers and data memory

Instrukce	Opcode	Funct3	Funct7	ALUSrc	ALUControl	MemWrite	MemToReg	RegWrite	BranchBeq	BranchJal	BranchJalr
lw	0000011	010	don't care								
sw	0100011	010	don't care								
add	0110011	000	0000000								
sub	0110011	000	0100000								
slt	0110011	010	0000000								
or	0110011	110	0000000								
and	0110011	111	0000000								
andi	0010011	000	don't care								
beq	1100011	000	don't care								
jal	1101111	don't care	don't care								
jalr	1100111	000	don't care								

The values of control signals according to previous slides

The Control Unit of the Single Cycle CPU



The control unit (CU)

- The control unit is typically a sequential circuit
 - It generates the control signals at appropriate time (CU outputs)
 - storage select, write enable (WE) and clock gating
 - data route – multiplexers control
 - function select – ALU operation/activation
 - It reacts to the status signals (CU inputs)
 - it only selects how to react on Zero in our case
 - many more things,
 - many more conditions can influence instruction cycle in case of real CPU – interrupts, exceptions etc.

Control unit – more detailed/generic

The task of CU is to control other units. It coordinates their activities and data exchanges between them. It controls fetching of the instructions from the (main/instruction) memory. It ensures their decoding and it sets gates, control and data paths to such state that instruction (can be) is executed.

Generally, the task of CU is to generate sequences of control signals for computer subsystems in such order that prescribed operations (arithmetic, program flow change, data exchange, control etc.) are executed.

Each step of this sequence can be considered or implemented as micro-operation. The micro-operation is elementary operation which reads and can change single or multiple registers (programmer visible or hidden in micro-architecture of CPU).

Usual effect of the micro-operation is change of the content of some register (in our case R0 to R31 or PC) or memory or both.

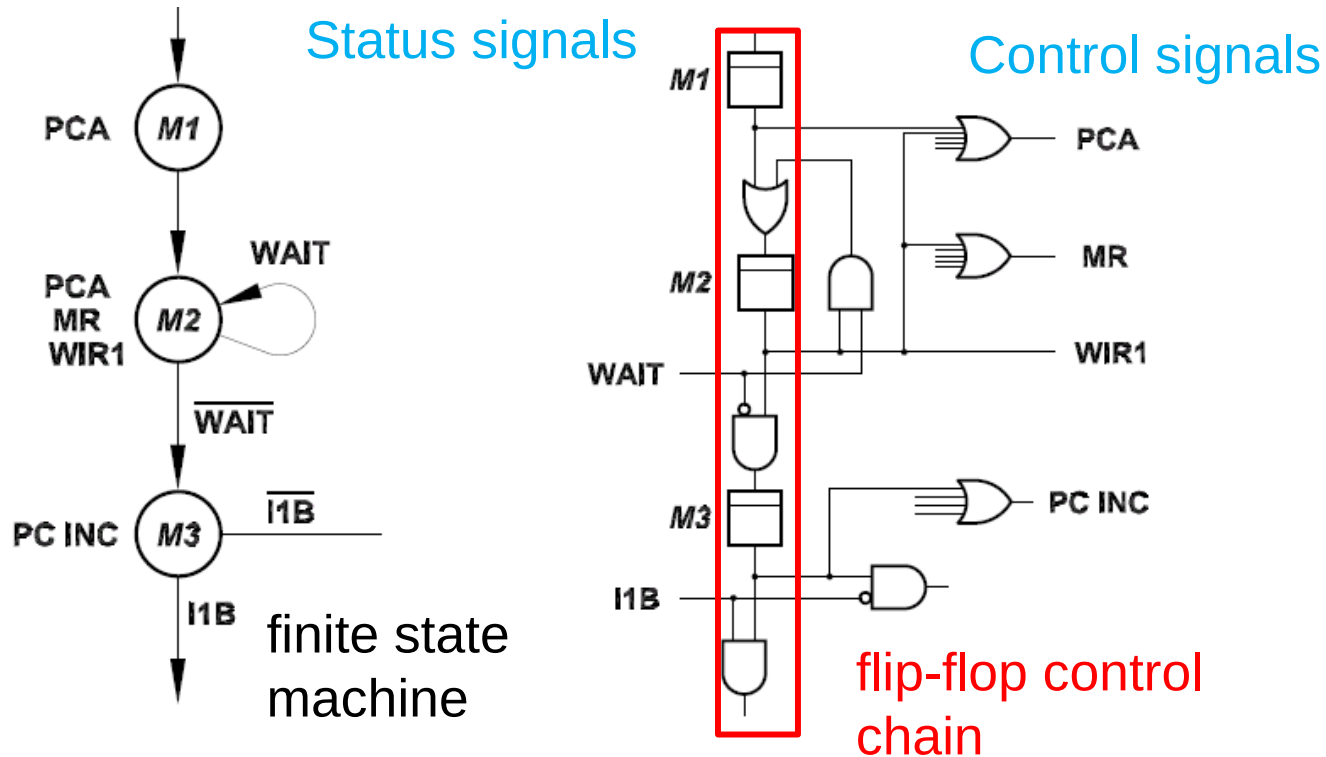
Some illustrative examples of micro-operation sequence

- $R(\text{MAR}) \leftarrow R(\text{CIAC})$
Move the content of Current Instruction Address Counter to the Memory Address Register
- $R(\text{CIAC}) \leftarrow R(\text{CIAC})+1$
Increment CIAC register
- $M(\text{MBR}) \leftarrow M(\text{MAR})$
read the value from the memory
- IF F(S) THEN $R(\text{A}) \leftarrow R(\text{MDR})$

Possible hardware realizations of the control unit

- Hardwired control unit – implemented by sequential circuit – next-state function/sequencer
 - one flip-flop per state chain (like ring counter)
 - with explicit counter
 - finite state machine (FSM – Mealy, Moore)
 - other implementation
- Microprogram control unit
 - horizontal microcode
 - vertical microcode
 - diagonal

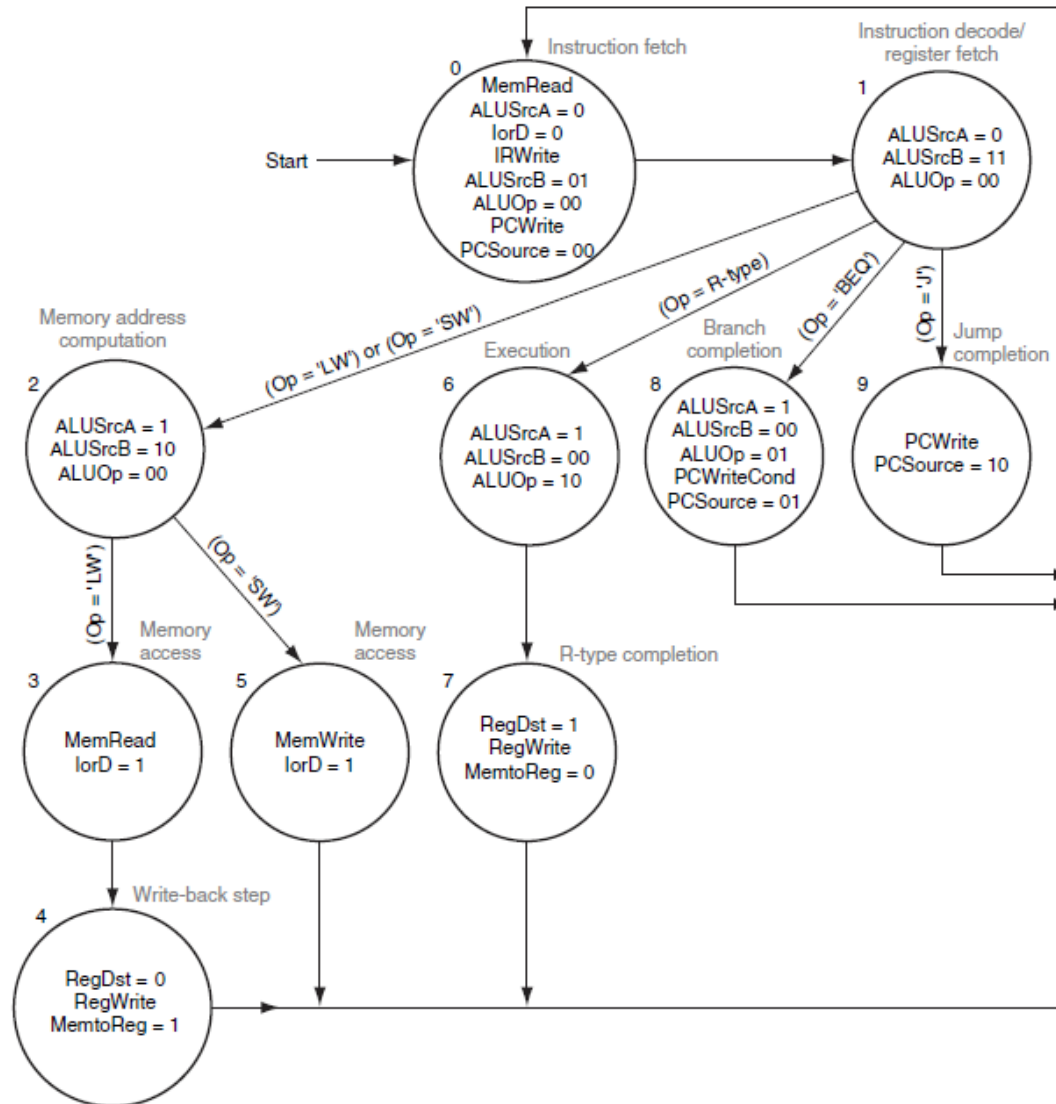
One flip-flop per state control unit



The function of CU can be prescribed by FSM. It can be straightforwardly implemented in VERILOG/VHDL (for the case that one instruction is executed in more cycles and there is no/minimal activities overlap).

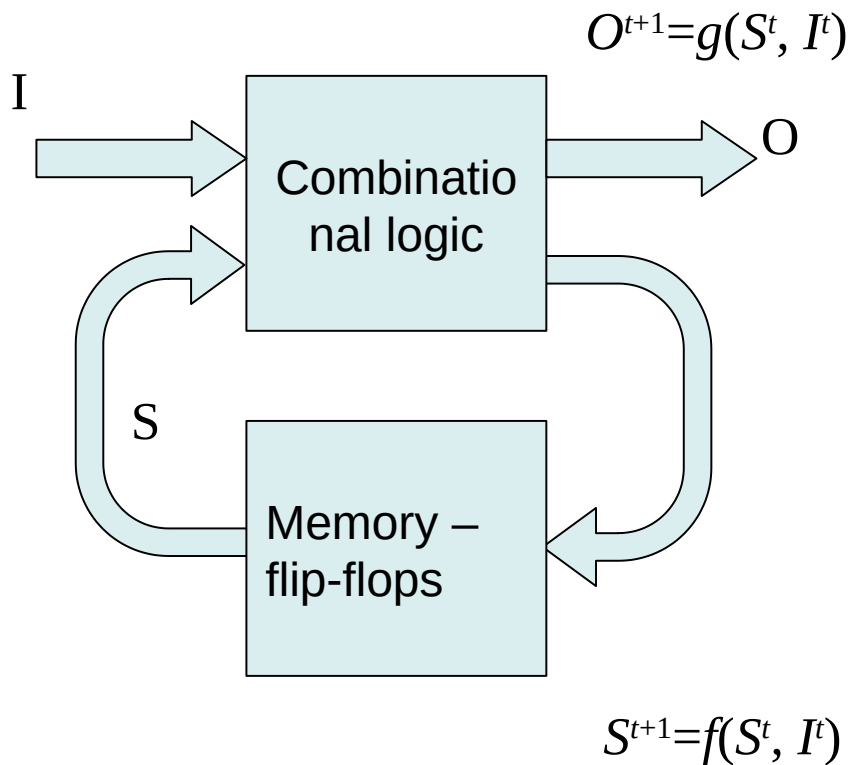
Note: The names of signals and states shown in this example do not correspond to the previously discussed MIPS CPU model!

Finite state machine – Example

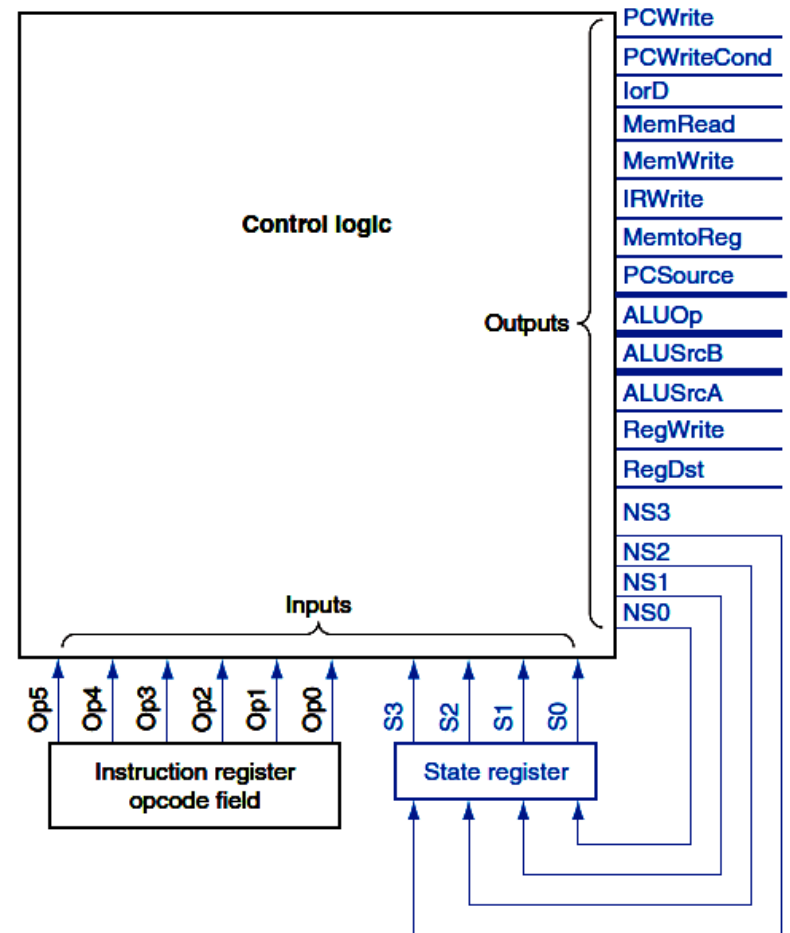


Finite state machine of CU – example

General model of logic sequential circuit (Huffman)



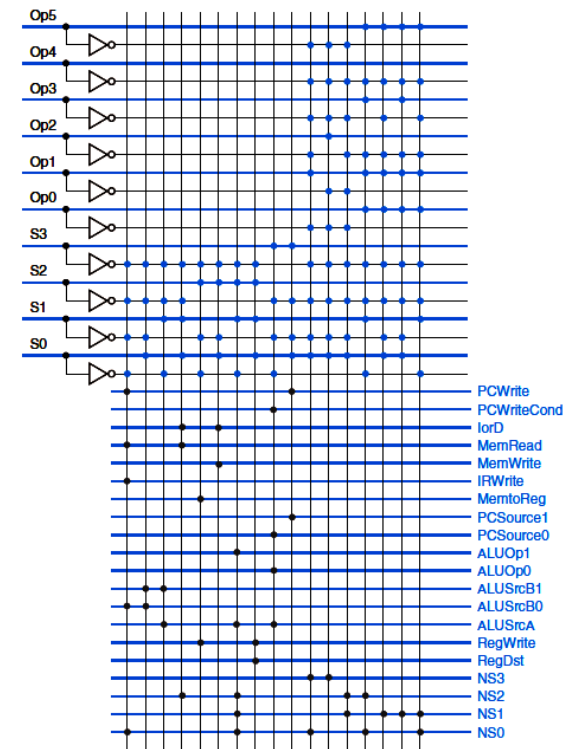
Control unit



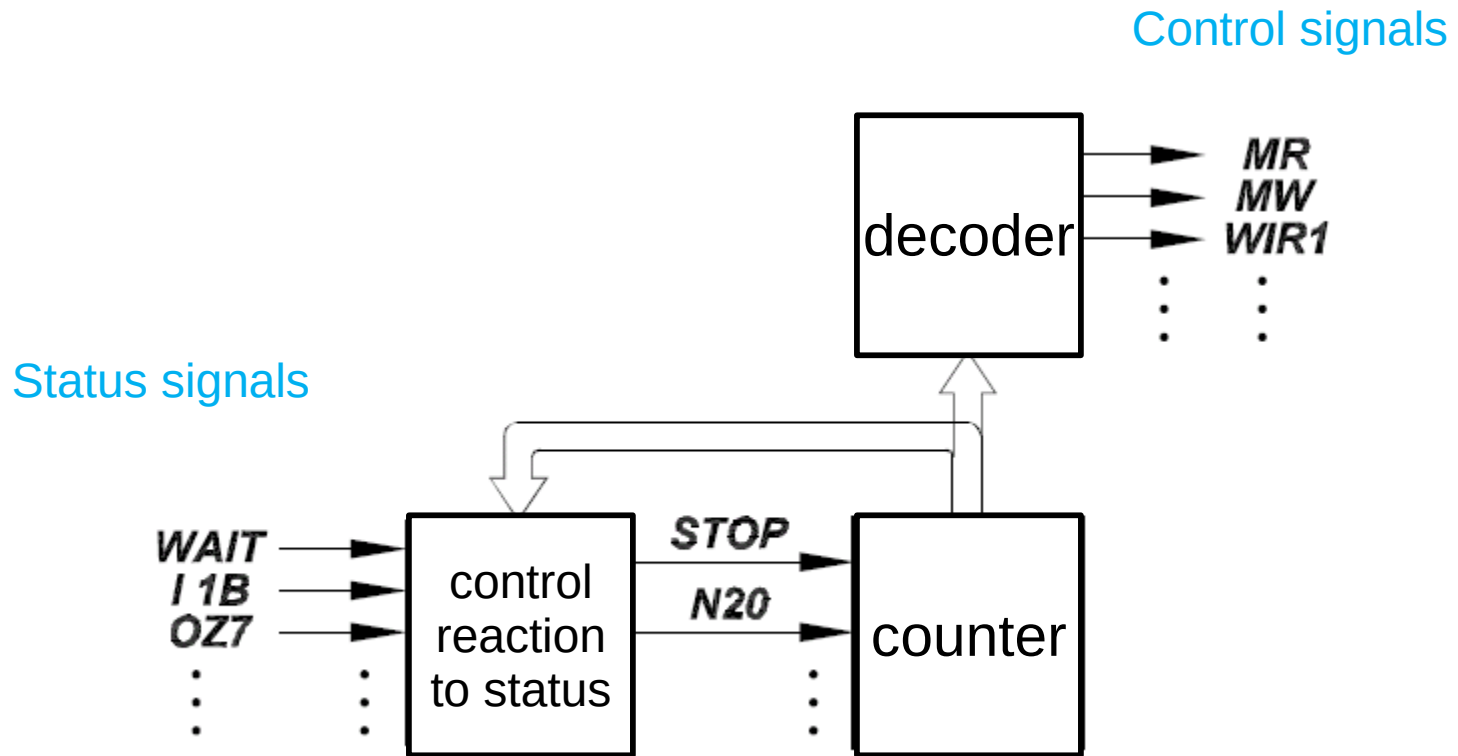
Finite state machine – Example

Combinational logic block show on previous slide can be implemented by:

- Directly as combinational logic build from separate logic gates
- With use of ROM memory (inputs are connected to ROM address inputs)
- With use of FPGA/PLA (programmable logic array)

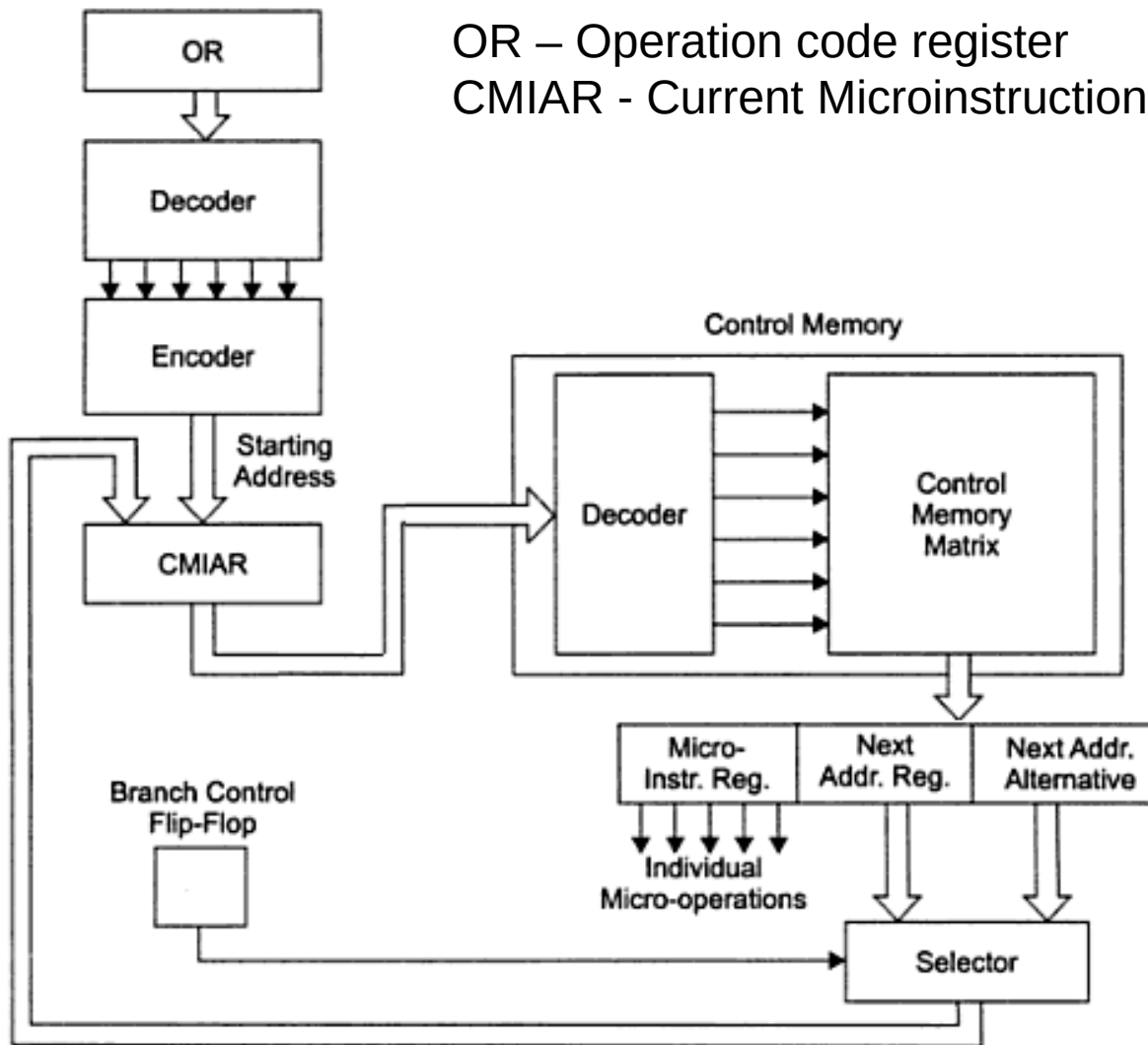


Explicit counter based control unit



Note: again for concept illustration only

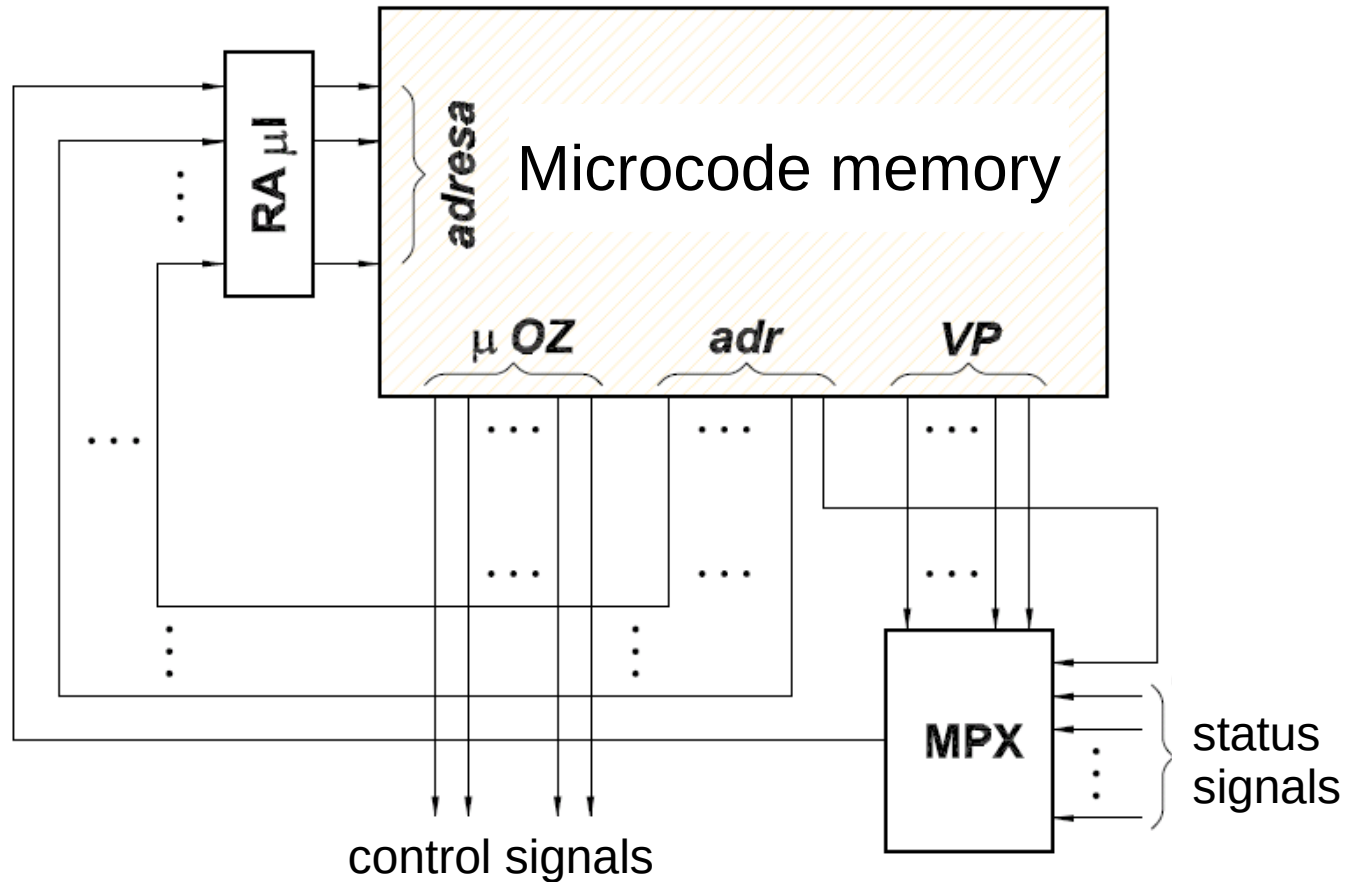
Microprogrammed control unit



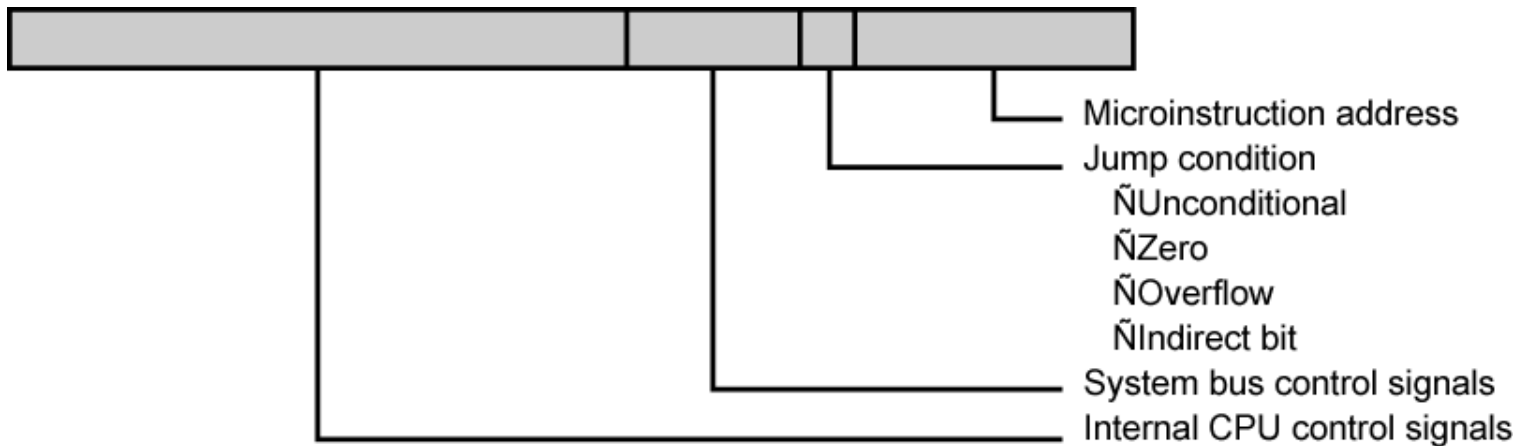
OR – Operation code register

CMIAR - Current Microinstruction Address Register

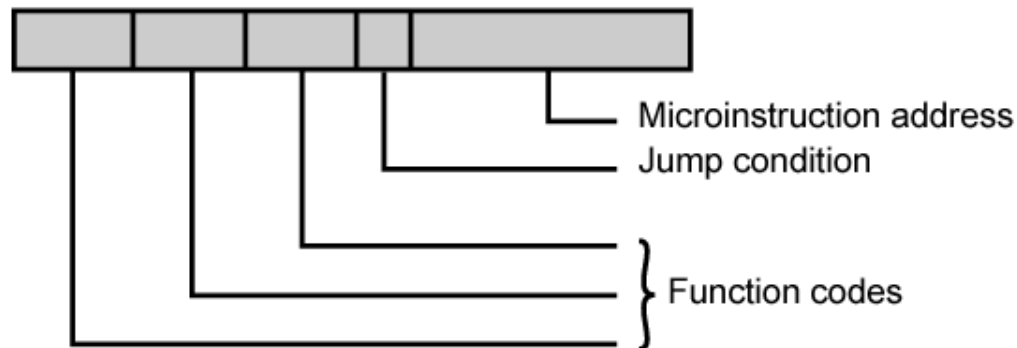
Horizontal microprogrammed control unit



Signals and next state encoding in microinstruction



(a) Horizontal microinstruction



(b) Vertical microinstruction

Wrap up structure of microprogrammed control unit

- Microprogrammed control unit is a computer in computer
 - Raμl is equivalent to PC,
 - Microcode memory is equivalent to program memory
 - μOP is equivalent to IR

Microprogrammed versus hardwired control unit

- Hardwired CU is faster and modifications for pipelined execution are possible (multiple execution stages activated in parallel)
- Price/gate count considerations
 - Hardwired is cheaper if simple (optimized instructions encoding)
 - Microprogrammed is cheaper when complex instructions/operations have to be processed
- Flexibility – microprogrammed CU can be modified more easily
- Microcode memory
 - ROM – fixed
 - RWM – instruction set can be changed/extended/fixed at CPU startup/configuration phase (i.e. used to patch bugs)

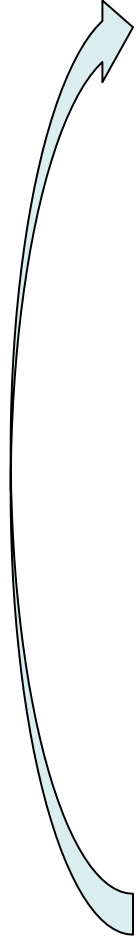
Conclusion for microprogrammed control units

- Microprogram is yet another layer between externally visible machine instructions and execution units.
- The concept of translating or interpreting (externally visible) instructions by control unit is common in CPUs, GPUs, disc and network controllers.
- The software/micro-program based implementation allows to realize more complex machine instructions without significant HW complexity increase.
- This microprogramming allows to define final function(s). Microcode is stored in (ROM, PLA, flash) inside CU.
- However, the sequential execution of microinstructions by CU leads to the low IPS rate, so more sophisticated solutions are used or microcode is left only for legacy part of instruction set support.

RISC versus CISC CPU

- RISC (Reduced Instruction Set Computers)
 - The CPU architectures where machine instructions encoding is optimized for simple decoding and fast execution. Exact structure is not prescribed and definition is fuzzy. More unambiguous is Load-Store concept.
 - Usual properties: all instructions are of the same length and can be executed in “single” cycle.
 - MIPS, SPARC, PowerPC, ARM, RISC-V
- CISC (Complex Instruction Set Computers)
 - Different machine instructions have different lengths.
 - Instructions are usually designed for dense code.
 - Motorola, Intel x86.

The instruction cycle with exception processing

1. Initial setup/reset – set initial PC value, PSW, etc.
 2. Read the instruction from the memory
 - PC \rightarrow to the address bus
 - Read the memory contents (machine instruction) and transfer it to the IR,
 - PC+I \rightarrow PC, where I is length of the instruction
 3. Decode operation code (opcode)
 4. Execute the operation
 - compute effective address, select registers, read operands, pass them through ALU and store result
 5. Check for exceptions/interrupts. If pending, service them
 6. If not repeat from the step 2
- 

Interrupts and exceptions

- External interrupts/exceptions
 - Method to process external asynchronous events. Processing cycle is stopped, CPU state is saved then the event is serviced. After the service is finished, CPU state is restored and the execution of interrupted program flow continues
- Exceptions synchronous with code execution
 - abnormal events – page faults, protection, debugging
 - software exceptions