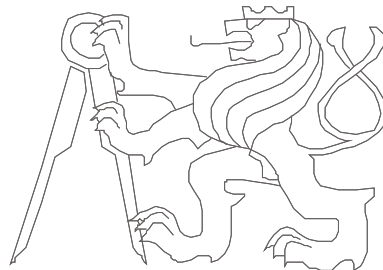


Computer Architectures

Central Processing Unit (CPU)

Pavel Píša, 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.



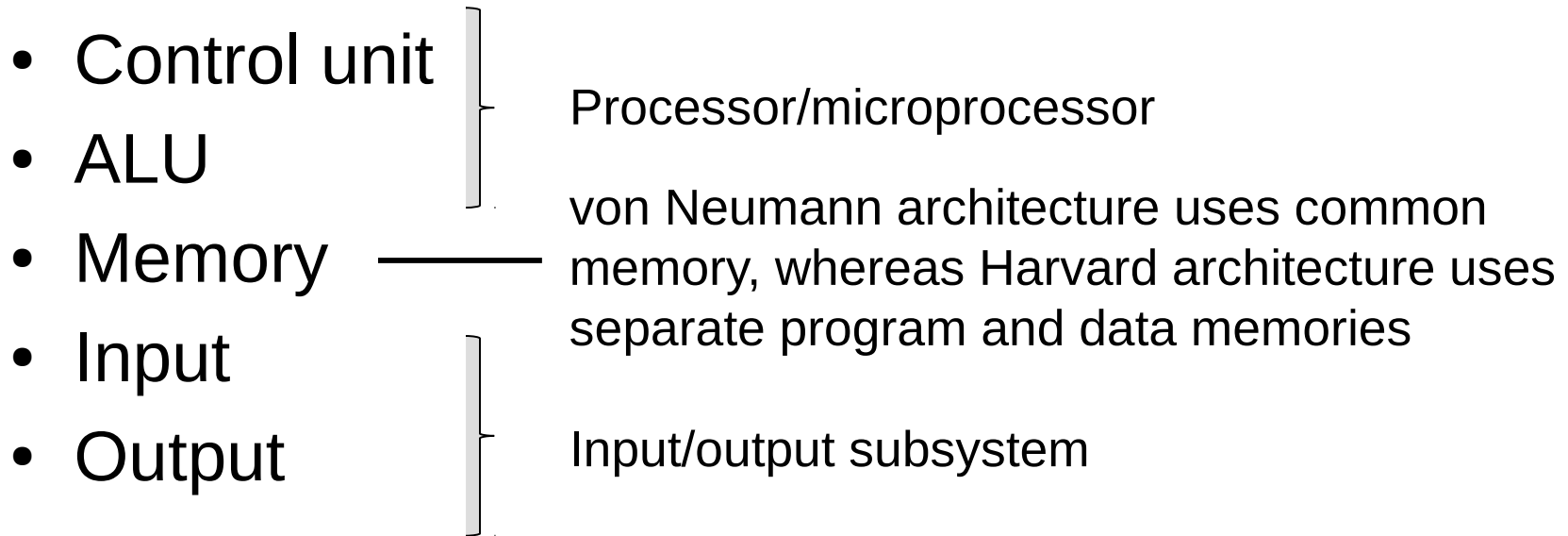
Czech Technical University in Prague, Faculty of Electrical Engineering

English version partially supported by:

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



Computer based on von Neumann's concept



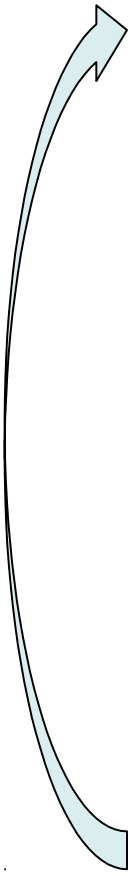
The control unit is responsible for control of the operation processing and sequencing. It consists of:

- 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

```
int pow = 1;
int x = 0;

while(pow != 128)
{
    pow = pow*2;
    x = x + 1;
}
```

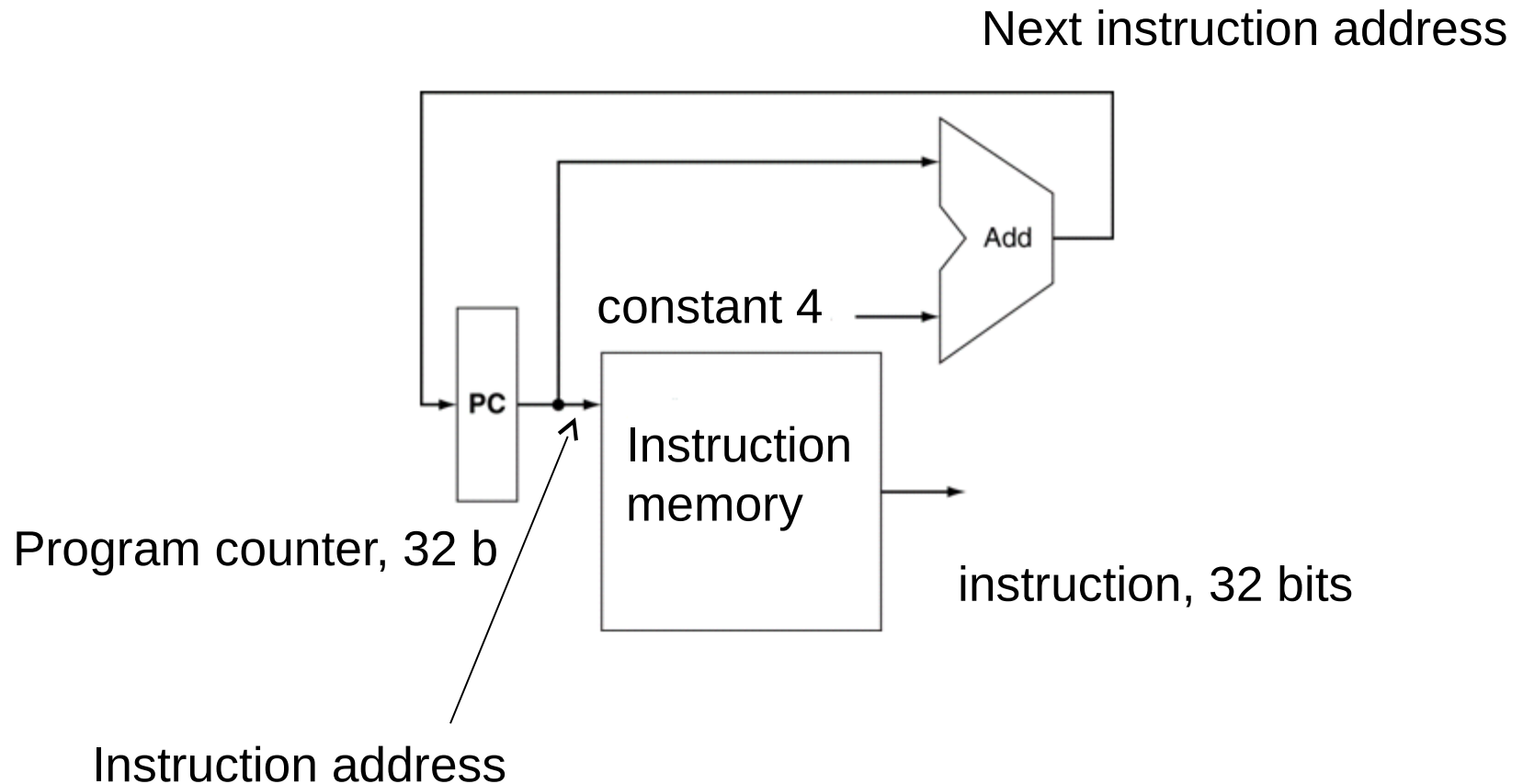
```
addi s0, $0, 1    // pow = 1
addi s1, $0, 0    // x = 0
addi t0, $0, 128  // t0 = 128

while:
    beq s0, t0, done // if pow==128, go to done
    sll s0, s0, 1    // pow = pow*2
    addi s1, s1, 1   // x = x+1

j   while
done:
```

8001FFF4	00 00 00 00		NOP
8001FFF8	00 00 00 00		NOP
8001FFFC	00 00 00 00		NOP
80020000	20 10 00 01	start()	ADDI \$16, \$00, 0x1
80020004	20 11 00 00		ADDI \$17, \$00, 0x0
80020008	20 08 00 80		ADDI \$08, \$00, 0x80
8002000C	12 08 00 04	while:	BEQ \$08, \$16, 0x4
80020010	00 00 00 00		NOP
80020014	00 10 80 40		SLL \$16, \$16, 1
80020018	08 00 80 03		J 0x8003
8002001C	22 31 00 01		ADDI \$17, \$17, 0x1
80020020	00 00 00 00	done:	NOP
80020024	00 00 00 00		NOP
80020028	00 00 00 00		NOP

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
 - Program flow change/jump instruction **beq**
- CPU will consist of control unit and ALU.
- Notes:
 - The implementation will be minimal (single cycle CPU – all operations processed in the single step/clock period)
 - The lecture 4 focuses on more realistic pipelined CPU implementation

The 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)

Opcode encoding

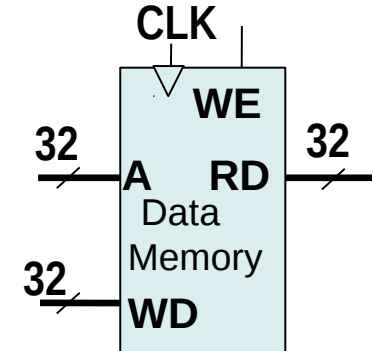
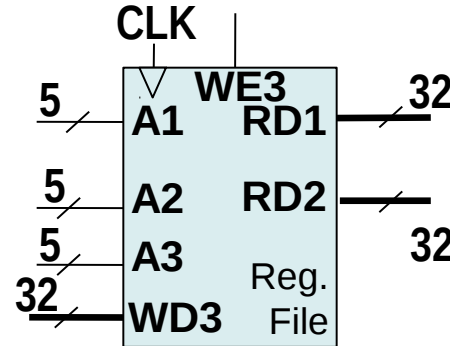
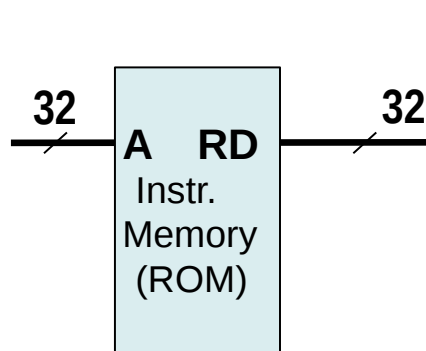
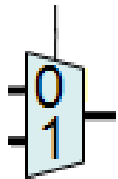
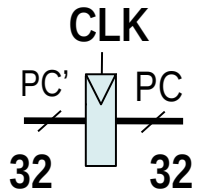
Decode opcode to the ALU operation

- Load/Store (I-type): F = add – add offset to the address base
- Branch (I-type): F = subtract – used to compare operands
- R-type: F depends on funct field

There are more I-type operations which use ALU in the real MIPS ISA

Instruction	Opcode	Func	Operation	ALU function	ALU control
lw	100011	XXXXXX	load word	add	0010
sw	101011	XXXXXX	store word	add	0010
beq	000100	XXXXXX	branch equal	subtract	0110
add	000000 R-type	100000	add	add	0010
sub		100010	subtract	subtract	0110
and		100100	AND	AND	0000
or		100101	OR	OR	0001
slt		101010	set-on-less-than	set-on-less-than	0111

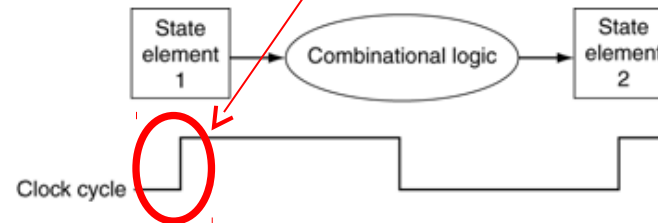
CPU building blocks



Write at the rising edge of CLK when WE = 1

Multiplexer

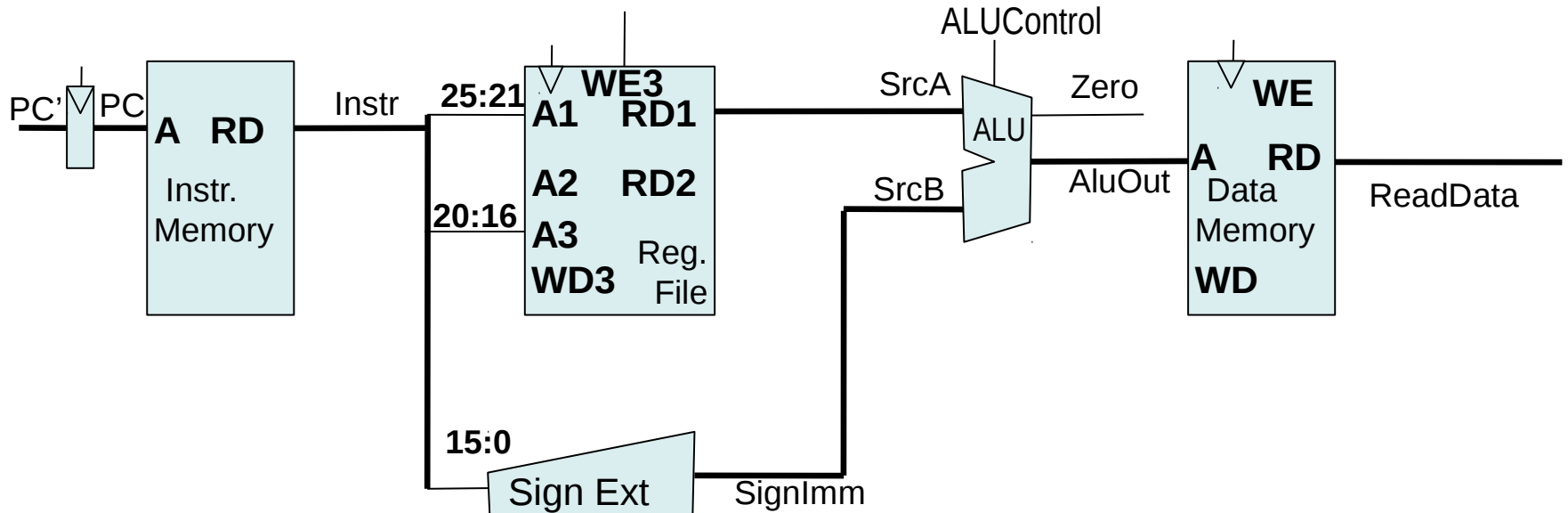
Read after "enough time" for data propagation



Single cycle CPU – implementation of the load instruction

lw: type I, rs – base address, imm – offset, rt – register where to store fetched data

I	opcode(6), 31:26	rs(5), 25:21	rt(5), 20:16	immediate (16), 15:0
---	------------------	--------------	--------------	----------------------

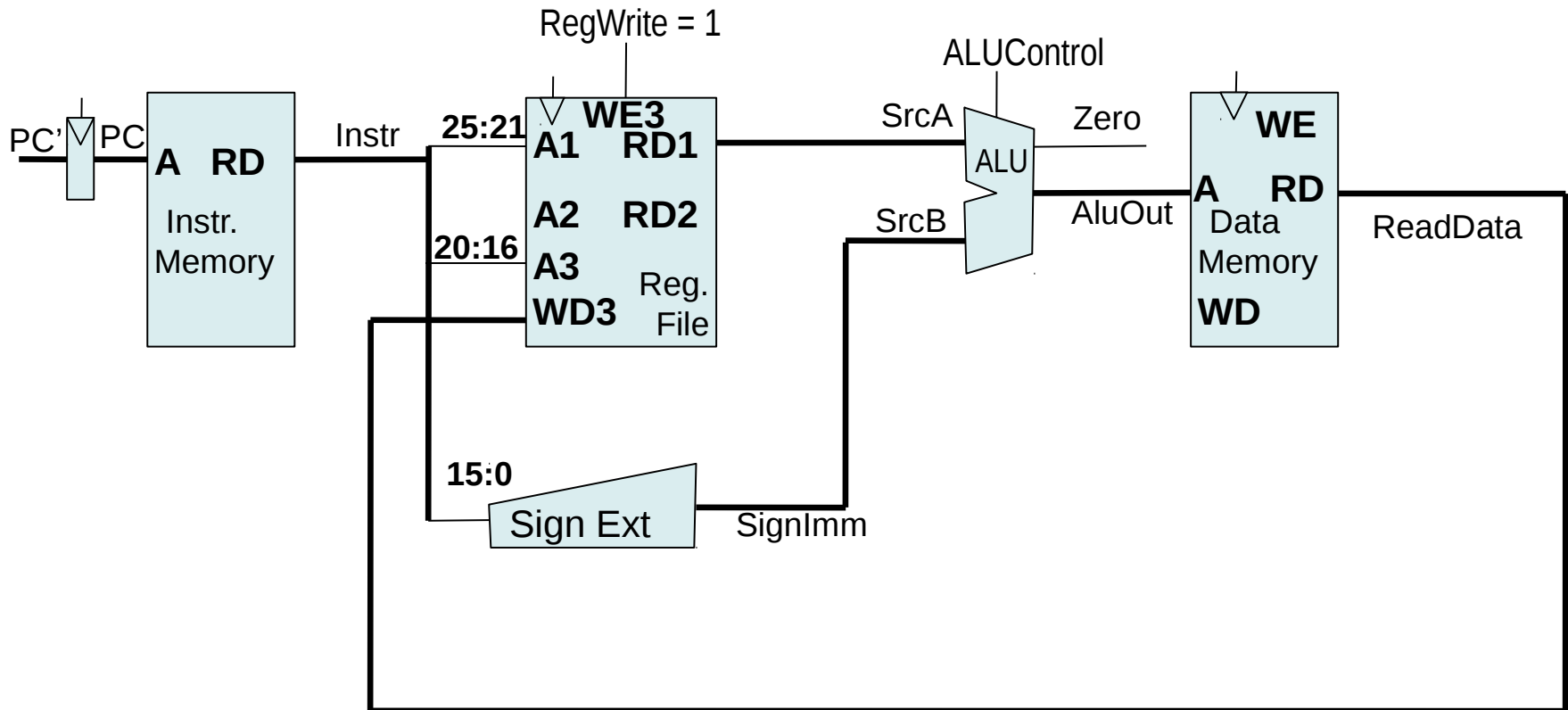


Single cycle CPU – implementation of the load instruction

lw: type I, rs – base address, imm – offset, rt – register where to store fetched data

I	opcode(6), 31:26	rs(5), 25:21	rt(5), 20:16	immediate (16), 15:0
---	------------------	--------------	--------------	----------------------

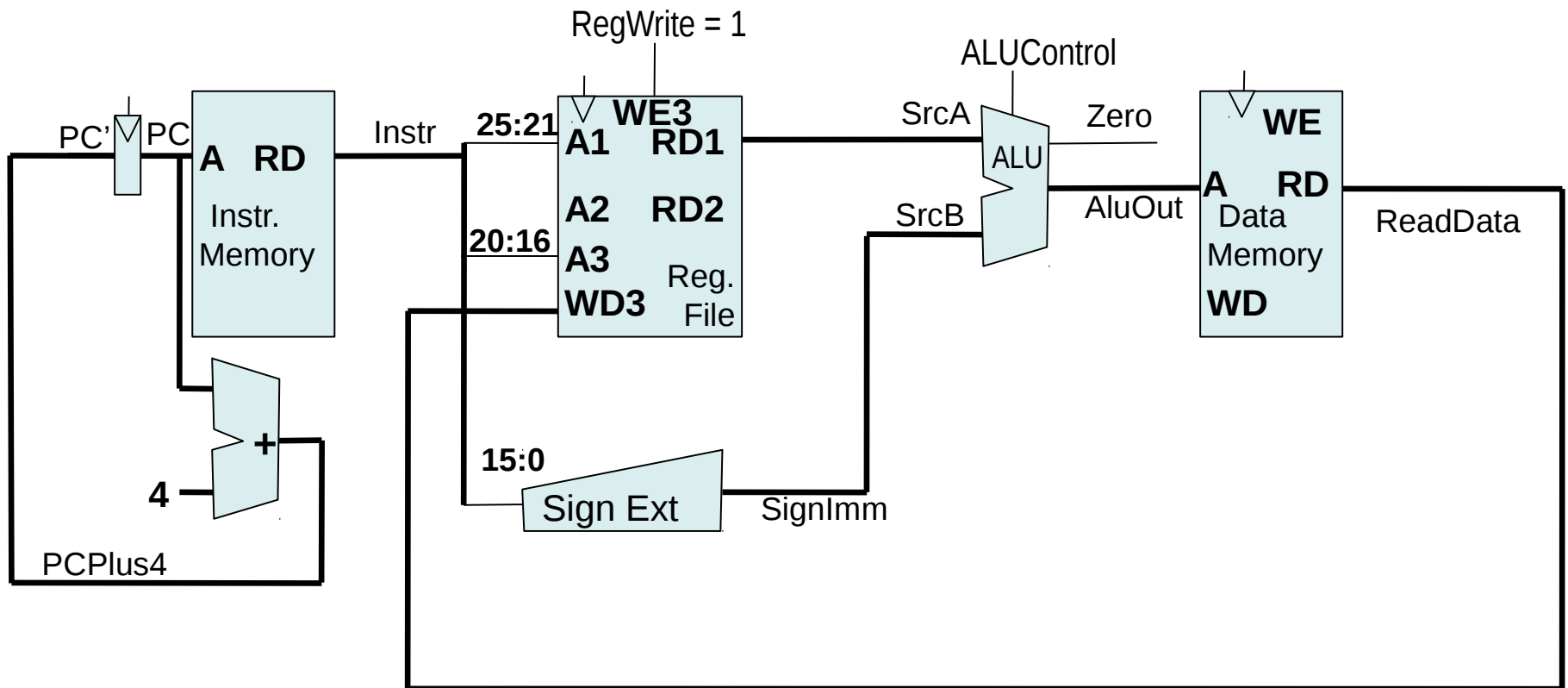
Write at the rising edge of the clock



Single cycle CPU – implementation of the load instruction

lw: type I, rs – base address, imm – offset, rt – register where to store fetched data

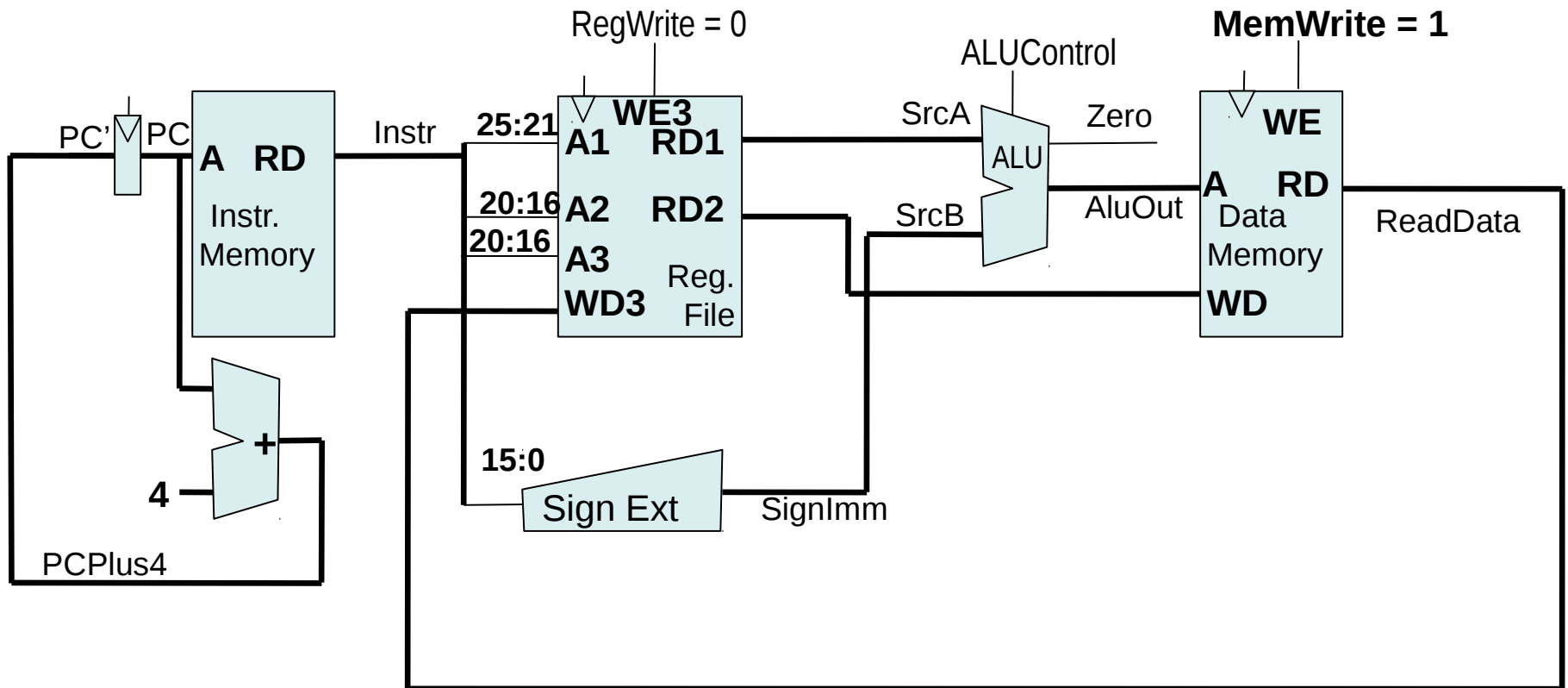
I	opcode(6), 31:26	rs(5), 25:21	rt(5), 20:16	immediate (16), 15:0
---	------------------	--------------	--------------	----------------------



Single cycle CPU – implementation of the store instruction

sw: type I, rs – base address, imm – offset, rt – select register to store into memory

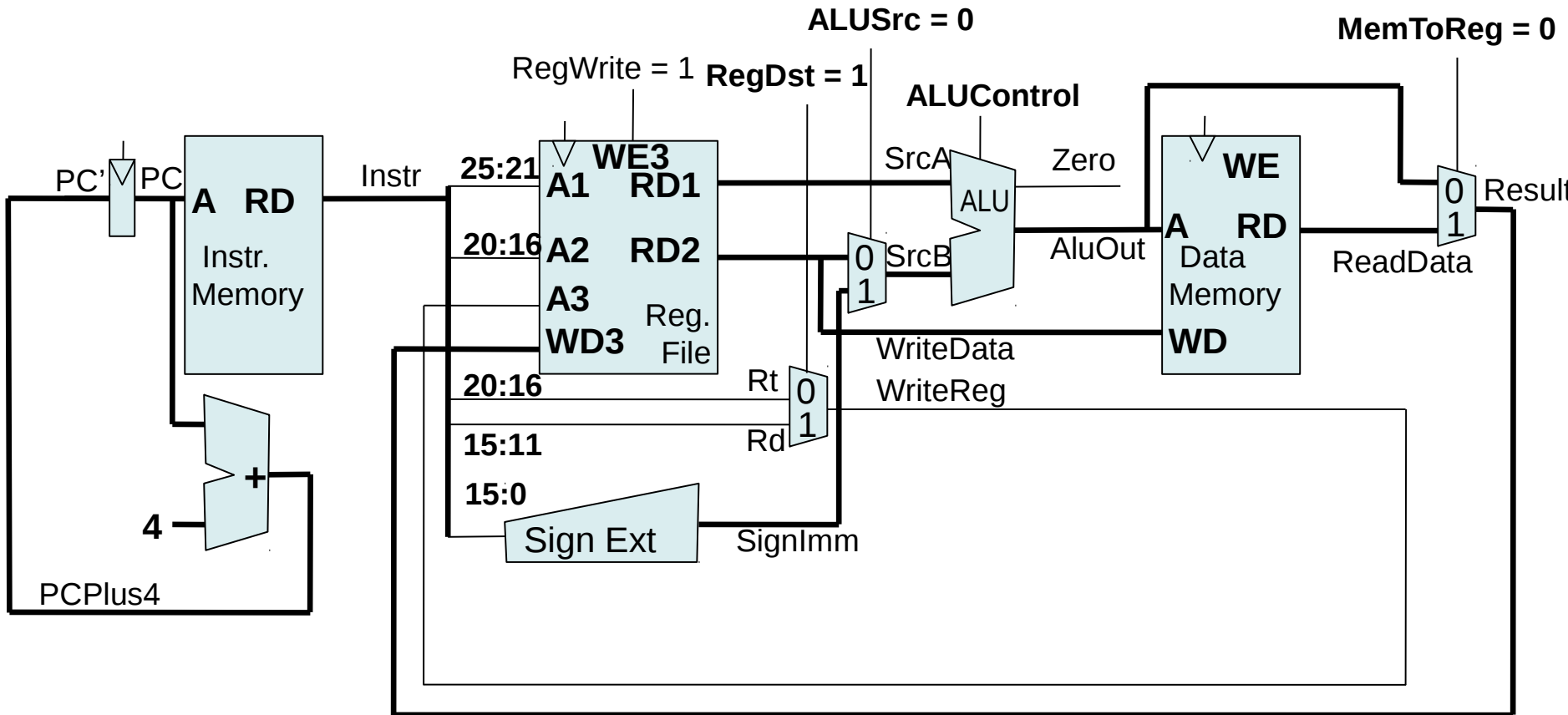
I	opcode(6), 31:26	rs(5), 25:21	rt(5), 20:16	immediate (16), 15:0
---	------------------	--------------	--------------	----------------------



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

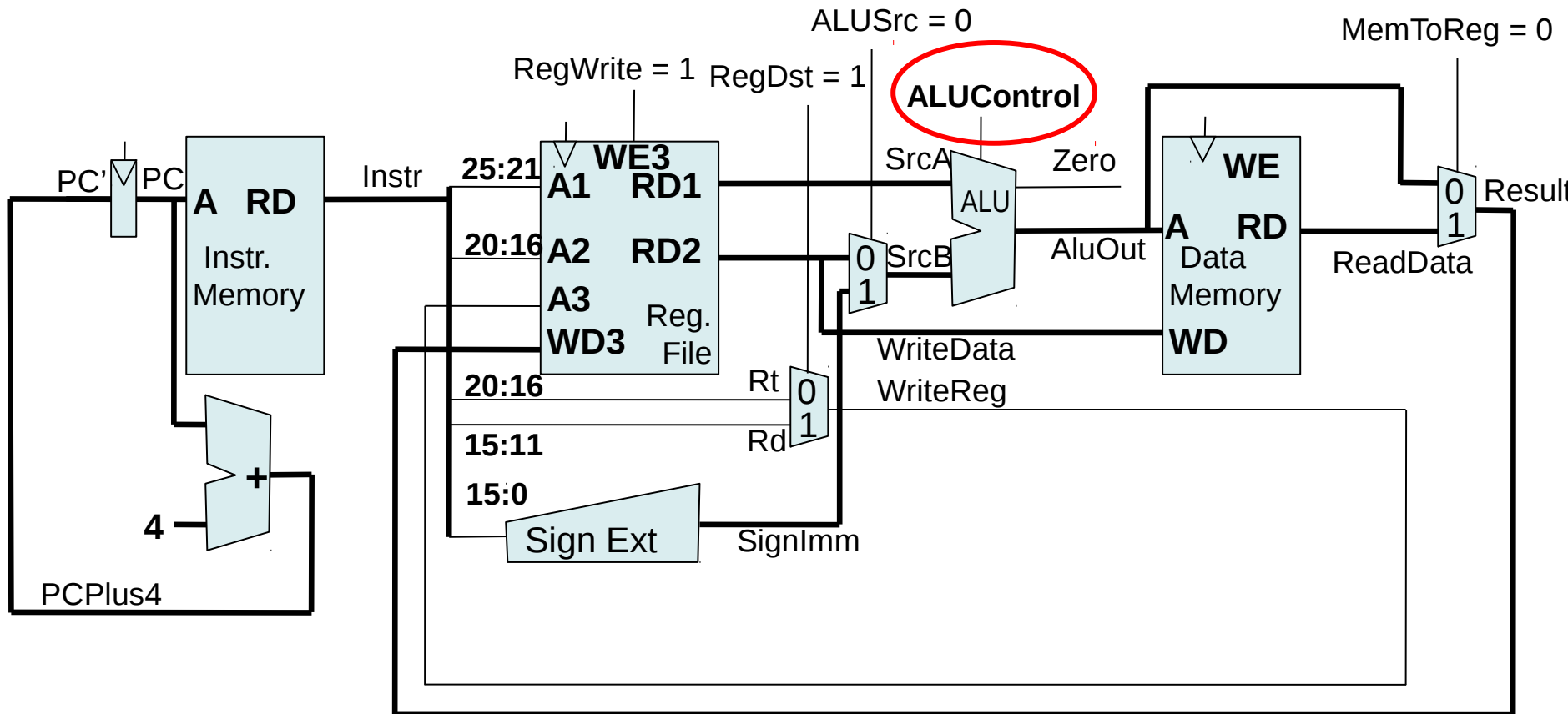
add: type R, rs, rt – source, rd – destination, funct – select ALU operation = add

R	opcode(6), 31:26	rs(5), 25:21	rt(5), 20:16	rd(5), 15:11	shamt(5)	funct(6), 5:0
---	------------------	--------------	--------------	--------------	----------	---------------



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

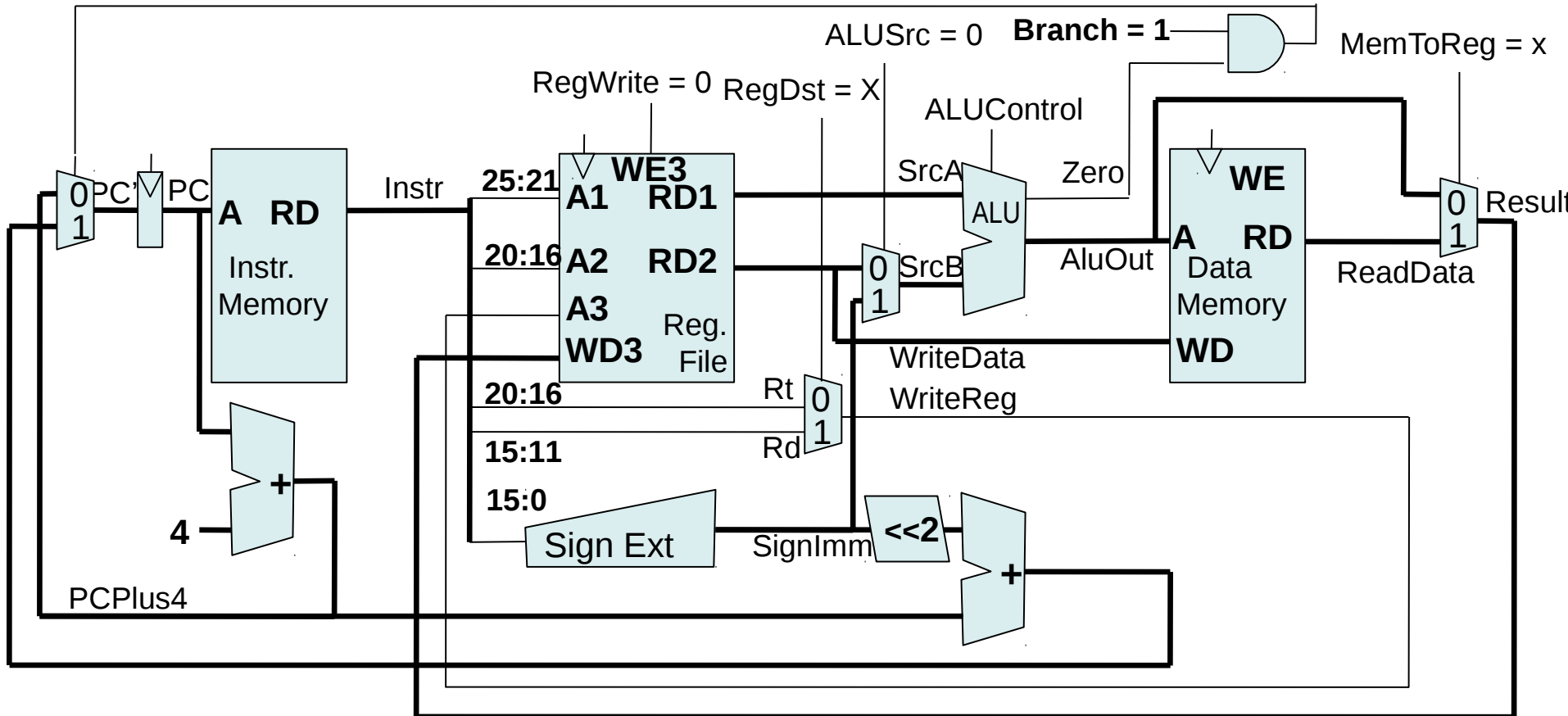
Only difference is another ALU operation selection (ALUControl). The data path is the same as for **add** instruction



Single cycle CPU – implementation of **beq**

beq – branch if equal; imm–offset; $PC' = PC + 4 + \text{SignImm} * 4$

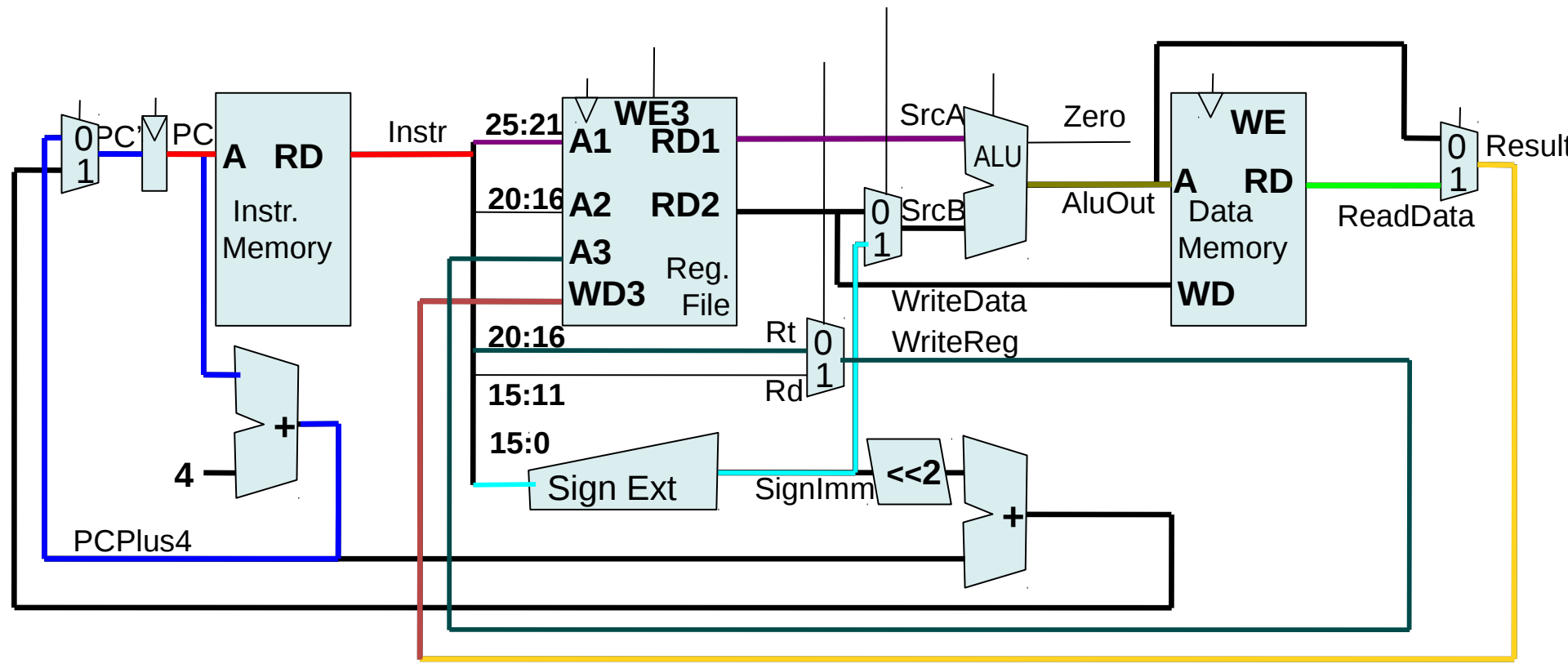
I	opcode(6), 31:26	rs(5), 25:21	rt(5), 20:16	immediate (16), 15:0
---	------------------	--------------	--------------	----------------------



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 **1w** 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 $T_c = 1020 \text{ ns} \rightarrow f_{CLK} \text{ max} = 980 \text{ kHz}$,

$IPS = 980e3 = 980 \text{ 000 instructions per second}$

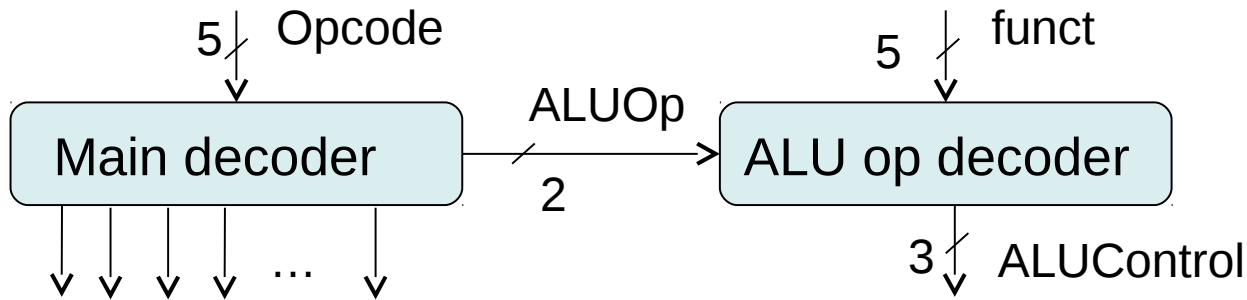
Notes

- 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

Single cycle CPU – Control unit

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				

Control signals values reflect **opcode** and **funct** fields



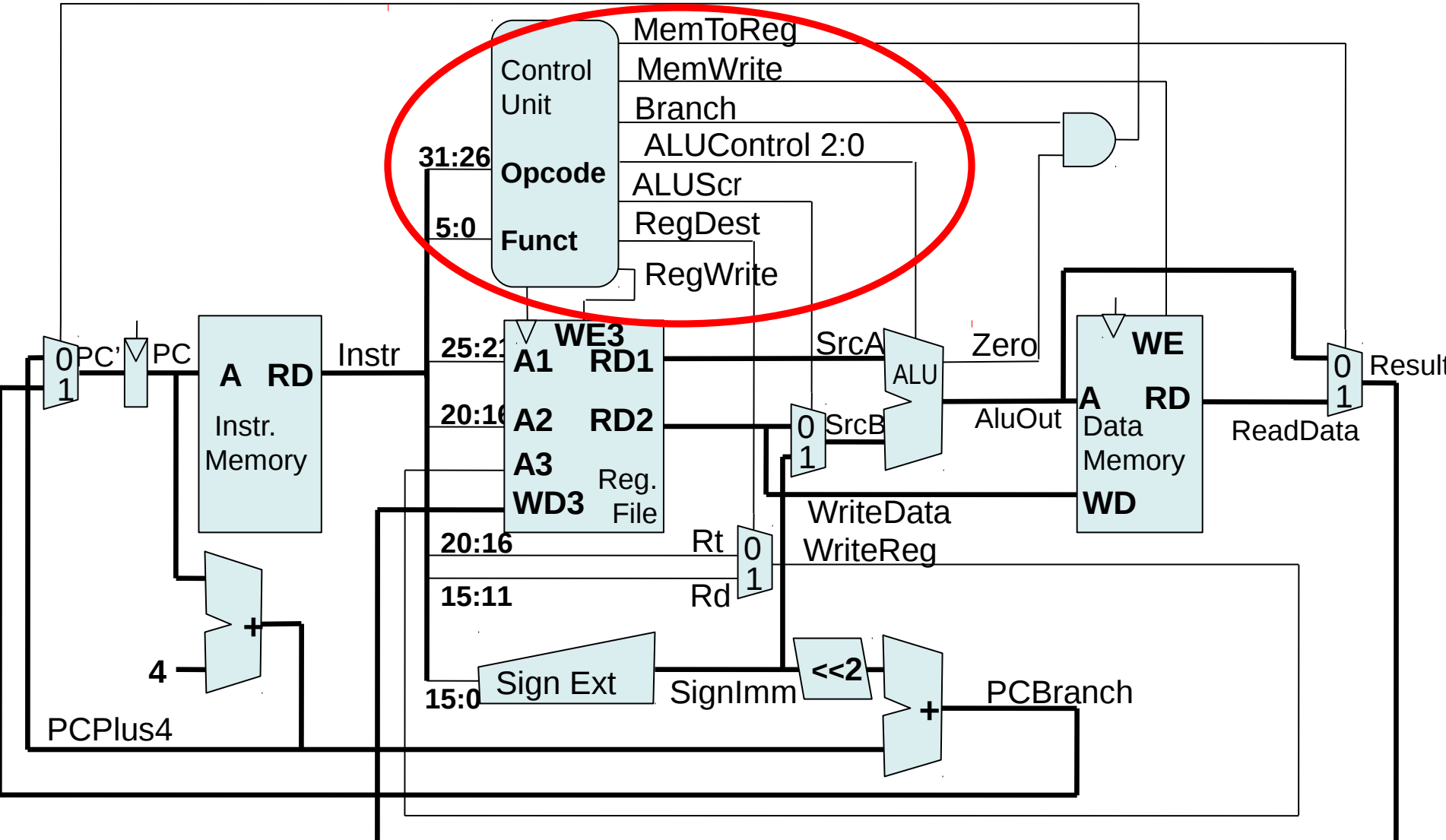
ALUOp	
00	addition
01	subtraction
10	according to funct
11	-not used-

	Opcode	Reg Write	RegDst	ALUSrc	ALUOp	Branch	Mem Write	MemTo Reg
R-type	000000	1	1	0	10	0	0	0
lw	100011	1	0	1	00	0	0	1
sw	101011	0	X	1	00	0	1	X
beq	000100	0	X	0	01	1	0	X

ALU Control (ALU function decoder)

ALUOp (selector)	Funct	ALUControl
00	X	010 (add)
01	X	110 (sub)
1X	add (100000)	010 (add)
1X	sub (100010)	110 (sub)
1X	and (100100)	000 (and)
1X	or (100101)	001 (or)
1X	slt (101010)	111 (set les than)

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

- $M(\text{MBR}) \leftarrow M(\text{MAR})$

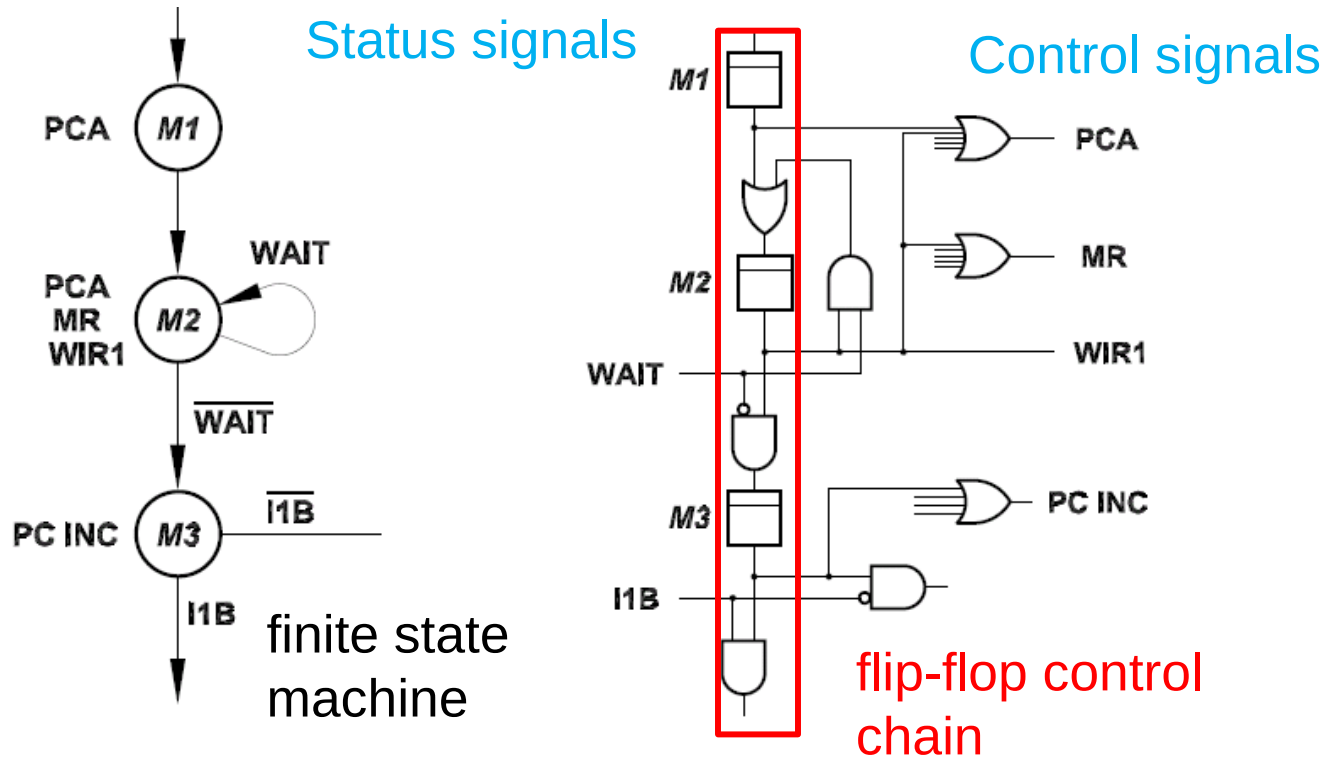
read the value from the memory

- IF $F(S)$ THEN $R(A) \leftarrow R(\text{MDR})$

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

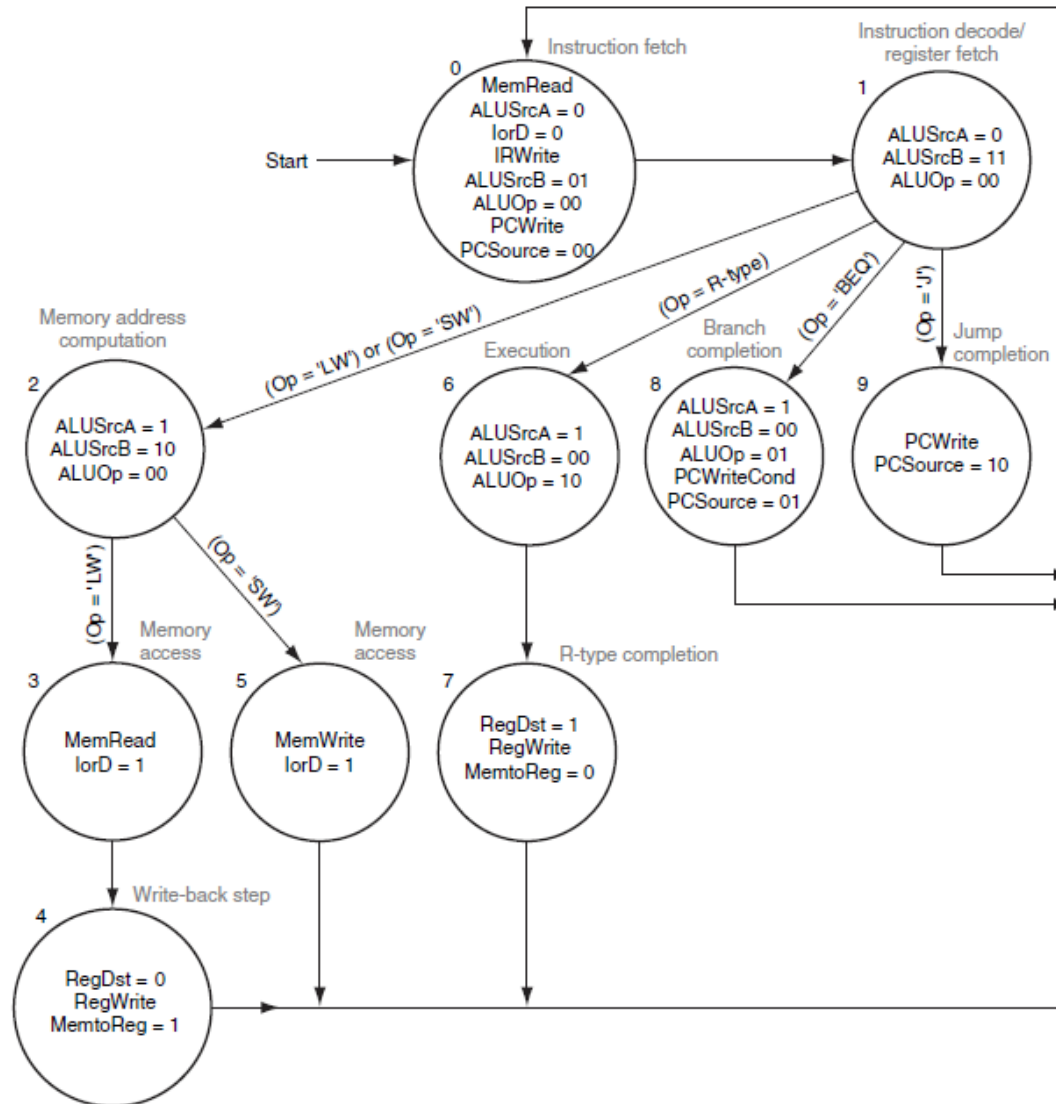
One flip-flop per state control unit



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).

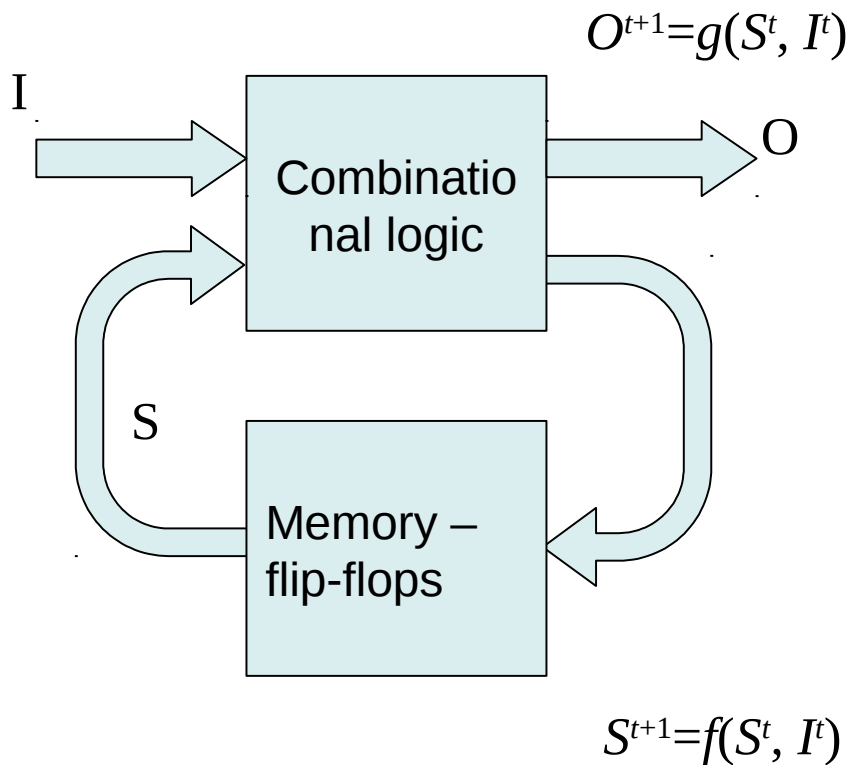
Note: The names of signals and states shown in this example do not correspond to the previously discussed MIPS CPU model!

Finite state machine – Example

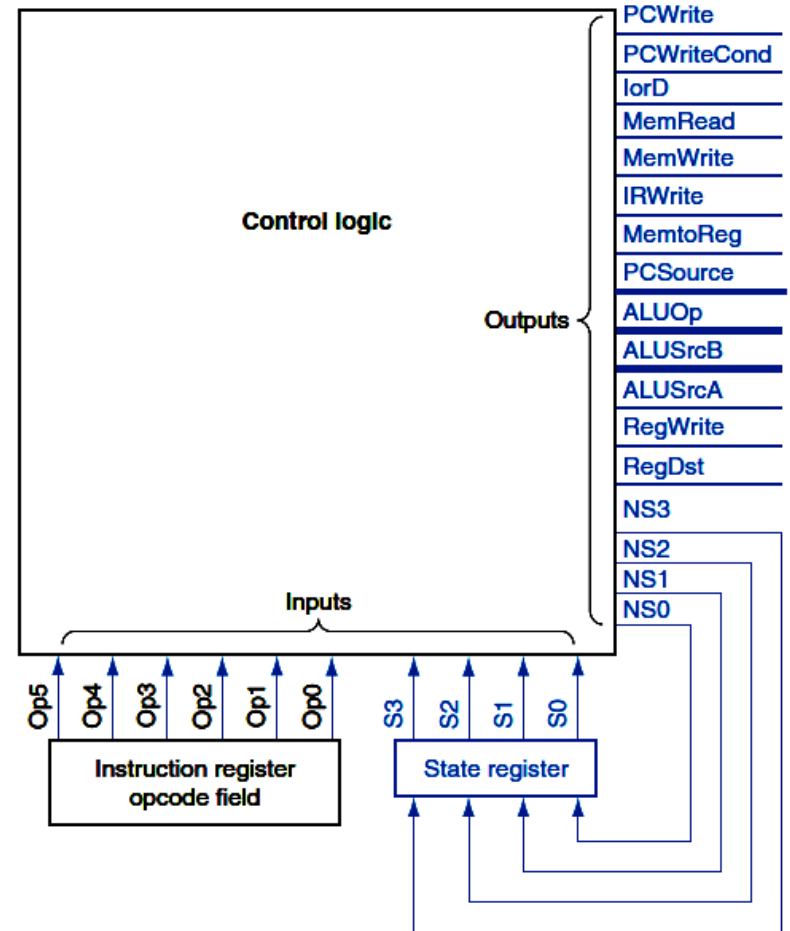


Finite state machine of CU – example

General model of logic sequential circuit (Huffman)



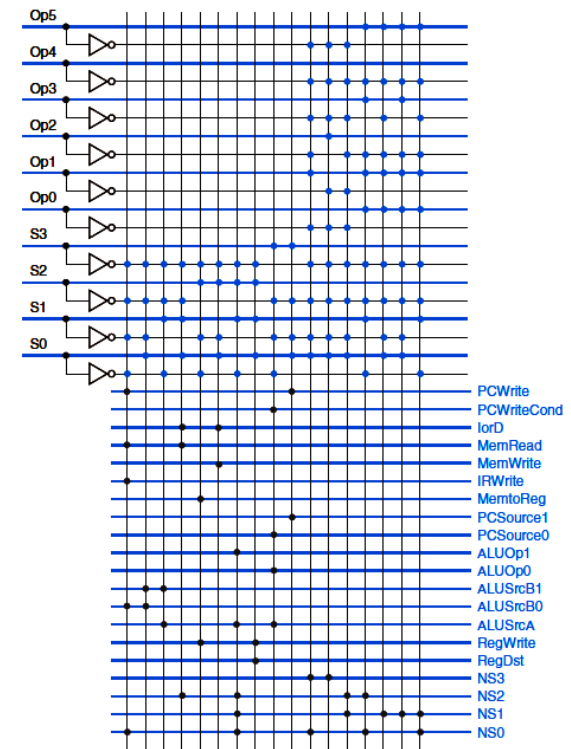
Control unit



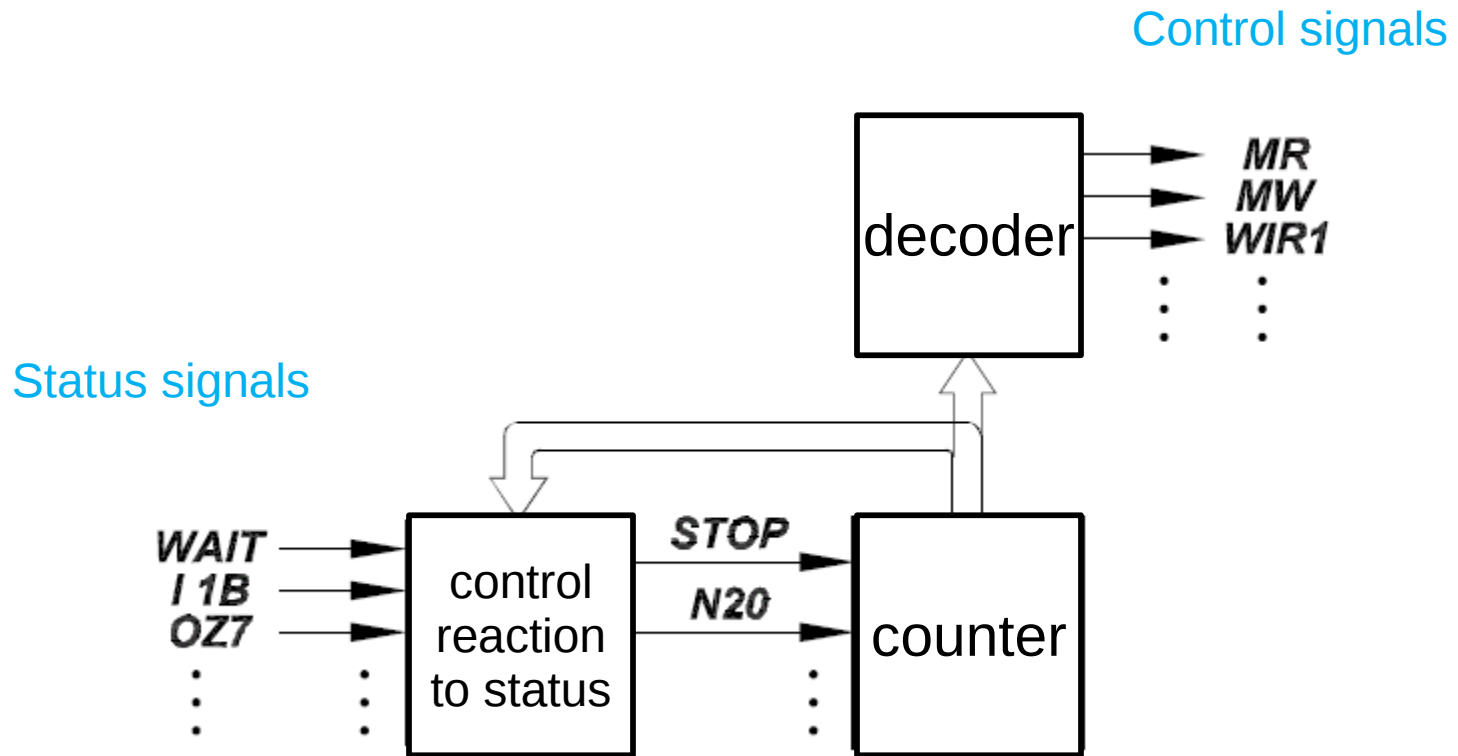
Finite state machine – Example

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)

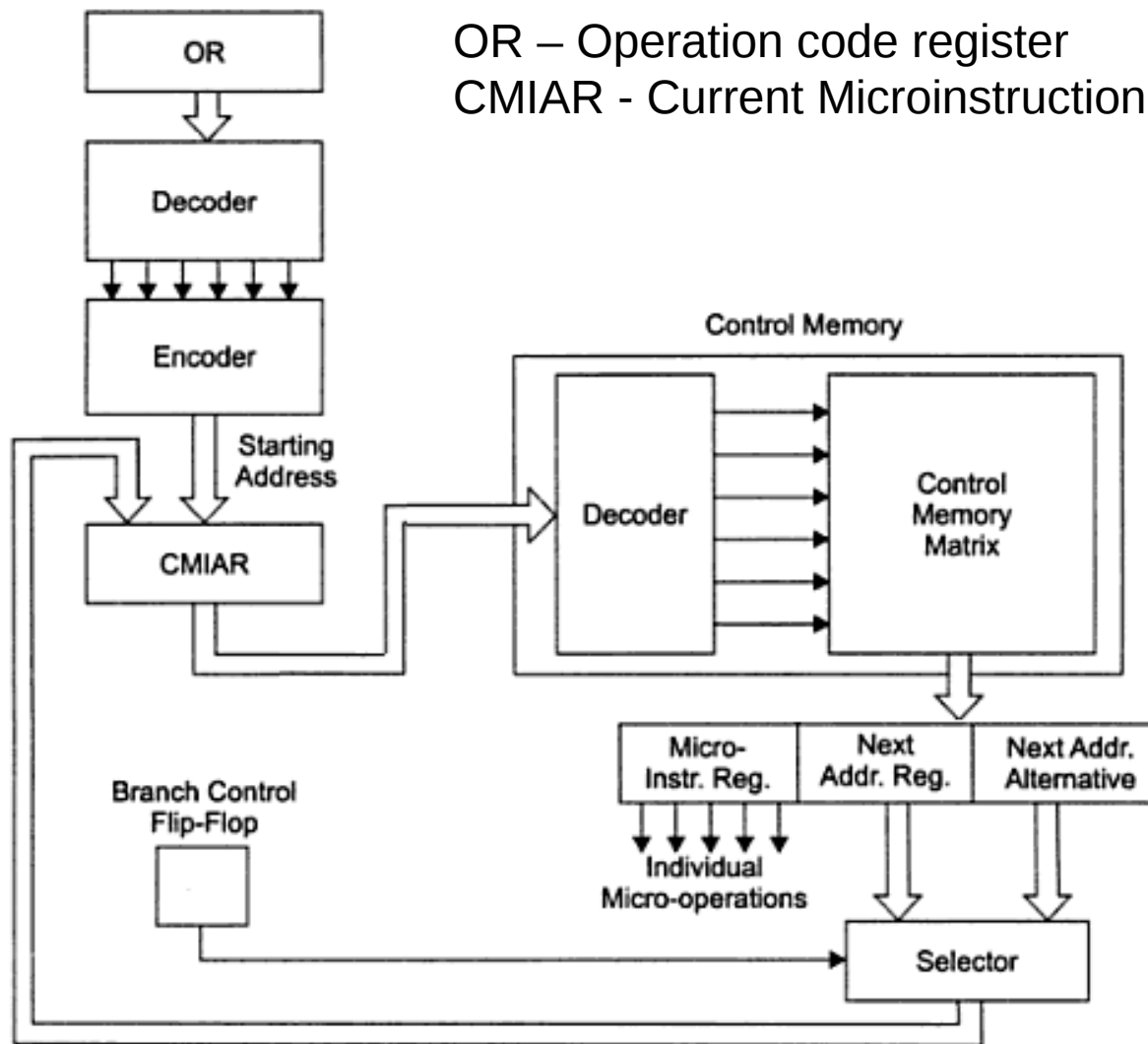


Explicit counter based control unit



Note: again for concept illustration only

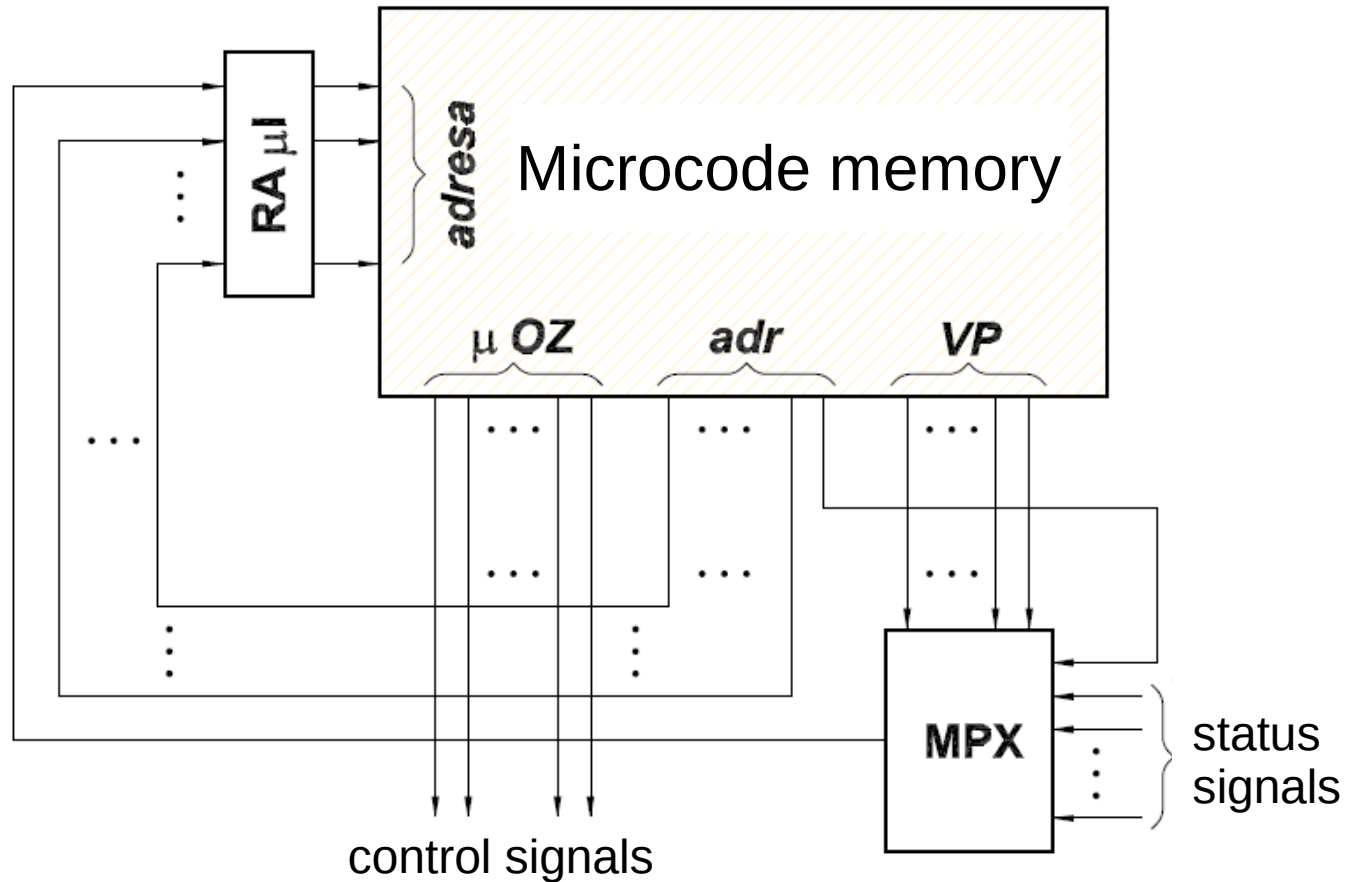
Microprogrammed control unit



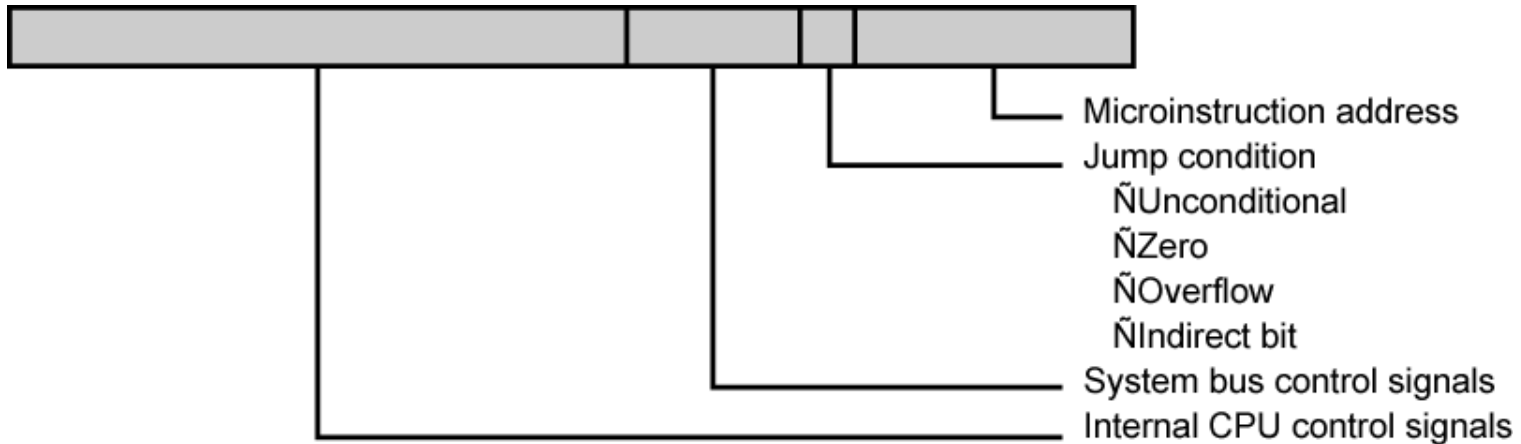
OR – Operation code register

CMIAR - Current Microinstruction Address Register

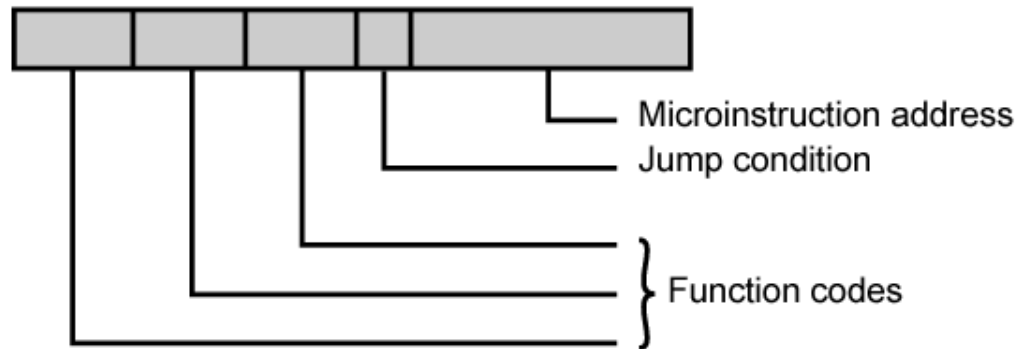
Horizontal microprogrammed control unit



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)

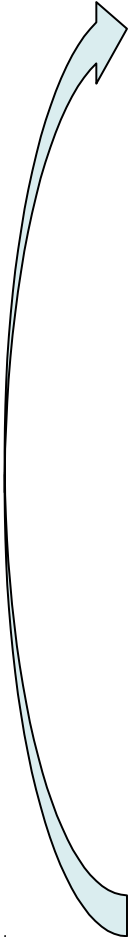
Conclusion for microprogrammed control units

- 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
- 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 → 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