

# B35APO: Computer Architectures

## Lecture 03. Central Processing Unit (CPU)

Pavel Píša  
pisa@fel.cvut.cz

Petr Štěpán  
stepan@fel.cvut.cz

License: CC-BY-SA



17. června, 2025

# Outline

- 1 Processor
- 2 Instruction Encoding
- 3 Blocks to Build Processor
- 4 Simple Single Cycle CPU Incremental Design

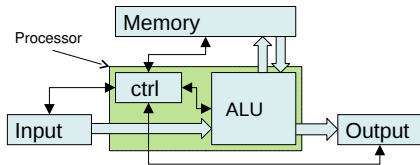
# History of QtMips and QtRVSim Simulators

- MipsIt used in past for our Computer Architecture course
- QtMips has been used for APO course from summer term 2019
  - QtMIPS – master's thesis of Karel Kočí supervised by Pavel Píša: *Graphical CPU Simulator with Cache Visualization*, available at <https://dspace.cvut.cz/bitstream/handle/10467/76764/F3-DP-2018-Koci-Karel-diploma.pdf>
  - Fixes, extension and partial internals redesign by Pavel Píša
- Switch to RISC-V architecture in 2022. Main work by Jakub Dupák in 2021, see the master's thesis *Graphical RISC-V Architecture Simulator - Memory Model and Project Management*, available at <https://dspace.cvut.cz/bitstream/handle/10467/94446/F3-BP-2021-Dupak-Jakub-thesis.pdf>
- Alternatives:
  - RARS: Risc-V Assembler and Runtime Simulator – IDE with detailed help and hints, evolved from MARS – <https://github.com/TheThirdOne/rars>
  - EduMIPS64: Java, superscalar pipeline 1x fixed and 3x FP pipelines – <https://www.edumips.org/>

# QtRVSim - Download and Presentations

- 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>
- Suse Factory TBD
- Online version <https://comparch.edu.cvut.cz/qtrvsim/app/>
- *QtRvSim - RISC-V Simulator with Cache and Pipeline Visualization*, RISC-V International Academic and Training SIG meeting, 2023, recording [https://youtu.be/J6AcPZZ\\_ISg](https://youtu.be/J6AcPZZ_ISg)
- *QtRVSim—Education from Assembly to Pipeline, Cache Performance, and C Level Programming*, FOSDEM 2023, Brussels, [https://fosdem.org/2023/schedule/event/rv\\_qtrvsim/](https://fosdem.org/2023/schedule/event/rv_qtrvsim/)

# John von Neumann Computer Block Diagram



- 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

Processor / microprocessor:

- Control Unit – CU
- Arithmetic-Logic Unit – ALU

Memory

- von Neumann architecture uses common memory, whereas Harvard architecture uses separate program and data memories
- memory contains cells, i.e., addressable units of same given size (today usually bytes – 8 bits)

Input/output subsystem:

- Input – keyboard, mouse
- Output – monitors, robotic actuators
  - Today most of peripheral devices functions as both, input, output, network interfaces, disk, SSD, and primarily input or output units has some monitoring and control channels

# Control Unit and Data Path

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

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

The general purpose registers are used to store actually processed data. For C language programs, they are used to keep some actively processed subset of local variables and temporary values required during expression evaluation.

The flow of instructions and data through processor is usually divided into

- data path – the way to load/store data from memory, pass them and retrieve them from registers and to provide them to ALU inputs and then route the results
- control path – flow of instruction, their decoding controlling data path according to algorithm

# The Most Important Registers of Processor

- PC (Program Counter) – holds address of a recent or next instruction to be processed
- IR (Instruction Register) – holds the machine instruction read from memory
- Other usually present registers
  - GPRs (General purpose registers) – directly under user control, 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) – keeps state as user or system code execution bit etc.
  - IM (Interrupt Mask) – controls acceptance of asynchronous events requests
  - FPRs (Floating point registers) – optional specialized register for hardware floating point real numbers processing
  - more specialize groups possible – vector registers, multimedia ones, etc.

# The Main Instruction Cycle of the CPU

- 1 Initial setup/reset – set initial PC value, PSW, etc.
- 2 Read the instruction from the memory
  - $PC \rightarrow$  to the address bus
  - Read the memory contents (machine instruction) and transfer it to the IR
  - $PC + I \rightarrow PC$ , point  $PC$  to next instruction,  $I$  is length of actually processed 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) – more in lecture 9
- 6 Repeat from the step 2

Compilation: C  $\rightarrow$  Assembler  $\rightarrow$  Machine Code

	<code>_start:</code>	
	<i>// int x = 157;</i>	
	<i>addi a0, zero, 157</i>	0x00000200: 09d00513
	<i>// int y = -1;</i>	
	<i>addi t1, zero, -1</i>	0x00000204: fff00313
<i>/* ffs as log2(x)*/</i>	<i>// while(x != 0) {</i>	
<b>int</b> x = 157;	<i>beq a0, zero, done</i>	0x00000208: 00050863
<b>int</b> y = -1;	<code>loop:</code>	
	<i>// x = x / 2;</i>	
<b>while</b> (x != 0) {	<i>srlr a0, a0, 1</i>	0x0000020c: 00155513
x = x / 2;	<i>// y = y + 1;</i>	
y = y + 1;	<i>addi t1, t1, 1</i>	0x00000210: 00130313
}	<i>// }</i>	
	<i>bne a0, zero, loop</i>	0x00000214: fe051ce3
	<code>done:</code>	

# Outline

- 1 Processor
- 2 Instruction Encoding**
- 3 Blocks to Build Processor
- 4 Simple Single Cycle CPU Incremental Design

# Instruction Encoding Constrains

Analysis what fits in given instruction length?

- Idea encode instruction byte, 8 bits, 256 combinations
  - Instruction has to encode operation code (**opcode**) and operands which should be processed by operation (if not restricted to implicit only)
  - consider 8 registers, 3 bits to encode, two operands instructions only  
 $reg_{rsd} = reg_{rsd} + reg_{s1}$  or  $reg_{rd} = MEM[reg_{s1}]$
  - 6 bits to select registers  $\rightarrow$  only 2 bits left  $\rightarrow$  4 operations in total
  - that is too small, alternative stack machine with implicit sources on top of stack (usually followed by pop) and implicit destination push to stack top, usually complex
  - extended bytes, immediate operands in byte following opcode and register
- 16-bit instruction encoding, 65536 combinations
  - consider 16 registers, for three operands only 4 bits, 16 two operand instructions left ( $4 + 3 * 4$  bits)
  - usually only two operand ( $8 + 2 * 4$  bits) instructions where 8 bits left for opcode, 256 combinations

# Instruction Encoding Constrains

- 32-bit instruction encoding, 4294967296 combinations
  - three operand  $reg_{rd} = reg_{rs1} + reg_{rs2}$  instructions fit
  - common choice 32 registers, and even then 128 thousands three operand instructions ( $17 + 3 * 5 = 32$  bits)
- The immediate values are required as well, add one to register, set register to specific value
  - but addresses to whole memory are required for global variables for example, if extended instruction words are undesirable, only small offsets (12 or 16 bits) fit
  - the larger ones has to be stored into memory and referenced by smaller offset which fits into instruction
  - PC relative addressing or addressing against global pointer **gp**
  - RISC (RISC-V, MIPS, SPARC, POWER, etc.) – typical fixed length encoding is 32 bits, 12 or 16-bit immediate
  - CISC (x86, m68k) – instruction variable length encoding, immediate and even registers for extended addressing in followup bytes, words

## RISC-V – Instruction Length Encoding

		xxxxxxxxxxxxxxxxaa	16-bit ( $aa \neq 11$ )
	xxxxxxxxxxxxxxxx	xxxxxxxxxxxxbbb11	32-bit ( $bbb \neq 111$ )
...xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxxx011111	48-bit
...xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxxx011111	64-bit
...xxxx	xxxxxxxxxxxxxxxx	xnnnxxxxxx011111	$(80 + 16 \cdot nnn)$ -bit ( $nnn \neq 111$ )
...xxxx	xxxxxxxxxxxxxxxx	x111xxxxxx011111	reserved for $\geq 192$ -bit

Address:

base+4

base+2

base

## RISC-V – Registers

Register	ABI Name	Description	Saver
x0	zero	Hardwired to zero	
x1	ra	Return address (from subroutine)	Caller
x2	sp	Stack pointer (for variables and save)	Callee
x3	gp	Global pointer (base for global data)	
x4	tp	Thread pointer (thread local store)	
x5-7	t0-2	Temporaries (intermediates in computation)	-
x8	s0/fp	Frame pointer (base for local variable and call frame)	Callee
x9	s1	Saved registers (for local variables)	Callee
x10-11	a0-1	Function arguments (when function called) / Return values at end of function	-
x12-17	a2-7	Function arguments	-
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	-
pc	pc	Program Counter (actual executed instruction)	
f0-31		Floating point registers	
		Machine control and status registers	

# 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`, load of upper bits `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

# The MIPS Instruction Formats and Instruction Types

Older but fits into three simple formats:

Type	31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 6	5 ... 0
R	opcode(6)	rs(5)	rt(5)	rd(5)	shamt(5)	funct(6)
I	opcode(6)	rs(5)	rt(5)	immediate(16)		
J	opcode(6)	address(26)				

- type (R,I,J) is defined for each 6-bit opcode combination
- 5 bits allows to encode 32 GPRs (0 / zero is hardwired to 0 / discard)
- rs – source register; rd – destination register; rt – alternative destination or the second source
- immediate, address – encoded direct operand value for ALU or address for branches
- shamt – immediate / constant for bit shift operations (<<, >>)
- funct – for R type specifies ALU operation, addition, subtraction, shift, etc.

# The RISC-V Instruction Format and Instruction Types

The six basic formats of the instructions are considered:

Type	31	30...25	24...21	20	19...15	14...12	11...8	7	6...0
R	fnct7		rs2		rs1	fnct3	rd		opcode
I	imm[11:0]				rs1	fnct3	rd		opcode
S	imm[11:5]		rs2		rs1	fnct3	imm[4:0]		opcode
B	imm [12]	imm [10:5]	rs2		rs1	fnct3	imm[4:1]	imm [11]	opcode
U	imm[31:12]						rd		opcode
J	imm [20]	imm[10:1]		imm [11]	imm[19:12]		rd		opcode

- instruction type (R,I,S,B,U,J) and encoding length known from 7-bit **opcode**
- rs1, rs2 – source registers; rd – destination register
  - the register fields in given role on fixed position for all encodings
- immediate – directly encoded operands for computation and branches, distributed into unused register fields
- fnct3, fnct7 – specifies executed operation (addition, subtraction, shift, etc.) and memory operands width (byte, word, etc.)

# RISC-V Encoding Codes Overview

The primary selection of operation is **opcode**:

Opcode	The group / operation	Actual operations for our subset
0110011	R-type (see func7 and func3)	add, sub, slt, or, and
0010011	ALU-imm. (see func 3)	addi, slti, ori, andi
0000011	Memory load (func3 with)	lw
0100011	memory store (func3 width)	sw
1100011	Branch (func3 condition)	beq
1101111	Subroutine call	jal
1100111	Return from subroutine	jalr
0000111	Load immediate into upper register bits	lui

Meaning of `fnct3` and `fnct7` for R-type instructions:

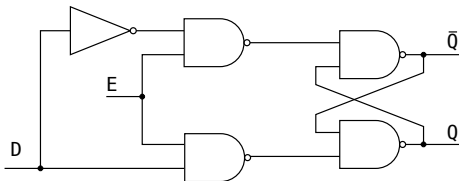
fnct7	fnct3	ALU operation
0000000	000	add
0100000	000	sub
0000000	010	slt
0000000	110	bitwise or
0000000	111	bitwies and

# Outline

- 1 Processor
- 2 Instruction Encoding
- 3 Blocks to Build Processor**
- 4 Simple Single Cycle CPU Incremental Design

# Realize Register in Hardware

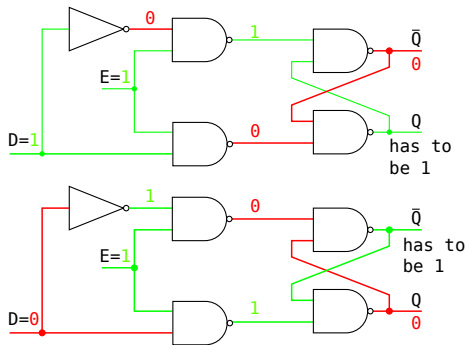
Basic realization of register from d-latch circuit



This circuit is a big change from the previous approach, because there is a feedback (loop) – the output of the gate is the input of the same gate, or the input of the other gate whose output is the input of the first gate. So what does it do?

# Realize Register in Hardware

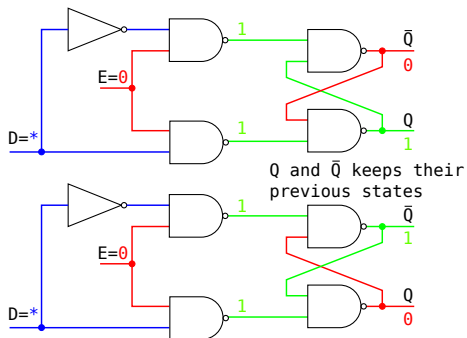
The circuit actual function is determined by E input (Enable).  
Analyze circuit for E=1 the first:



- The D input propagates Q output after delay
  - when  $D = 1$  then output has to become  $Q = 1$ , because one of inputs to last gate is 0, and NAND output becomes 1
  - when  $D = 0$  then output has to become  $\bar{Q} = 1$ , again one input to corresponding NAND gate is 0, when  $\bar{Q} = 1$ , then output  $Q = 0$ , because both inputs of driving gate are 1, NAND output has to be 0

# Realize Register in Hardware

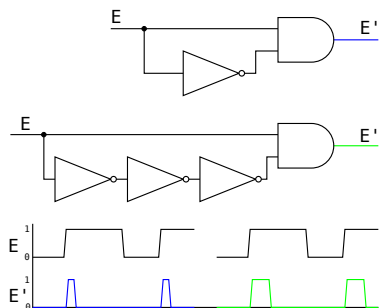
If the enable input is deactivated ( $E=0$ ):



- The state of  $Q, \overline{Q}$  outputs depends only on previous value of  $Q, \overline{Q}$
- The one of states  $Q = 1, \overline{Q} = 0$  or  $Q = 0, \overline{Q} = 1$  is preserved from last situation when  $E = 1$

# Realize Register in Hardware

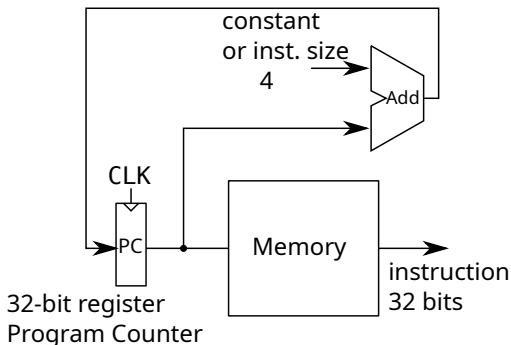
Activation by clock signal rising edge:



- Is it possible that output of such circuit reaches active (1) state?
- for ideal/matthematic case no ( $C$  and  $\overline{C}$ ) = 0
- in reality, however, the gate output is delayed compared to the direct bypass signal and for this delay time the output is  $E' \vee 1$

- if this value is too short compared to the d-latch circuit settle time, then it is possible to add more consecutive negations (green curve)
- the variant of sequential circuit which state is recorded at clock signal edge is called d flip-flop

# Hardware Realization of CPU Instruction Cycle

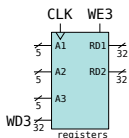


The PC register is updated to  $PC+4$  (advanced by instruction length) at clock (CLK) signal rising edge which causes start of the fetch of following instruction from memory.

# Processor Basic Building Blocks



single 32-bit wide register, the input store at CLK rising edge



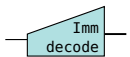
32 registers in register file, A1, A2 inputs select which register values are connected to RD1 and RD2 outputs; if WE3 (write enable) inputs is active then WD3 input state is written to register designated by A3 at CLK signal rising edge



multiplexer copies one of its input signals to the output according to Select input value

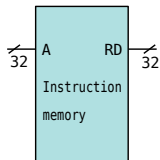


the operation chosen by ALUControl signal is applied to inputs SrcA and SrcB. Result is available on ALUout signal and flags reflects result conditions, i.e., sign or indicate zero value

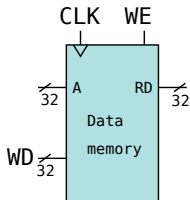


chooses bits used for immediate encoding and applies sign extension to 32 bits signed value

# Processor Basic Building Blocks – Memory



Instruction memory provides data stored in the memory cell/word indexed by address input A on its output RD. Data are available after interval "long enough" to stabilize address decoder and propagate value to the output RD (the read access time).



Data memory. In read mode, it delivers memory cell/word content indexed by address A onto RD output (again read access time has to be respected). If the input WE (write enable) is active then data stable for long enough setup time are stored into selected word at rising edge of the clock (CLK). The time from data and address ready to stable data writes into cell is called write access time.

# Outline

- 1 Processor
- 2 Instruction Encoding
- 3 Blocks to Build Processor
- 4 Simple Single Cycle CPU Incremental Design**

# The Load Word Instruction - LW

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

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

Example: Reads 32-bit word stored at address 0x400 in memory into register 2:

lw x2, 0x400(x0)

iiii iiiiiiii ssss s 010 dddd d000 0011  
 0100 0000 0000 0000 0 010 0001 0000 0011

0x400                      0      func3                      2                      opcode

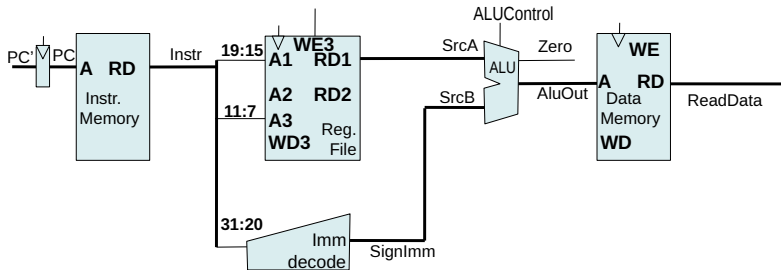
0x 40 00 21 03 – machine code lw x2, 0x400(x0)

Remark: x0 register provides fixed zero value which is not changed even by write

# The Load Word Instruction – Implementation

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

Typ	31	30...25	24...21	20	19...15	14...12	11...8	7	6...0
I	imm[11:0]				rs1	fnct3	rd		opcode

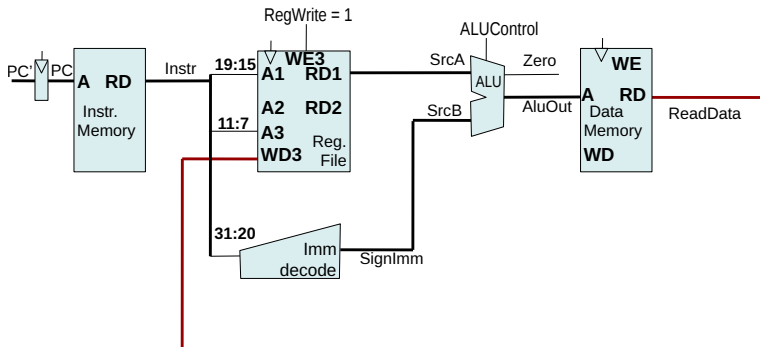


# The Load Word Instruction – Implementation

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

Typ	31	30...25	24...21	20	19...15	14...12	11...8	7	6...0
l	imm[11:0]				rs1	fnct3	rd		opcode

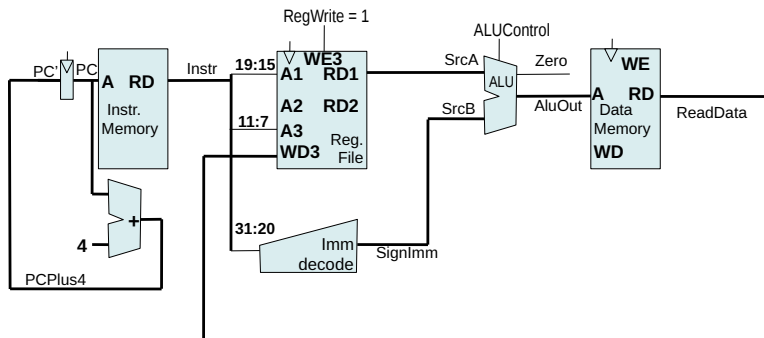
Write to register at the rising edge of the clock



# The Load Word Instruction – Implementation

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

Typ	31	30...25	24...21	20	19...15	14...12	11...8	7	6...0
I	imm[11:0]				rs1	fnct3	rd		opcode



## QtRvSim - RISC-V Simulator

The image shows the QtRvSim RISC-V simulator interface with several components labeled:

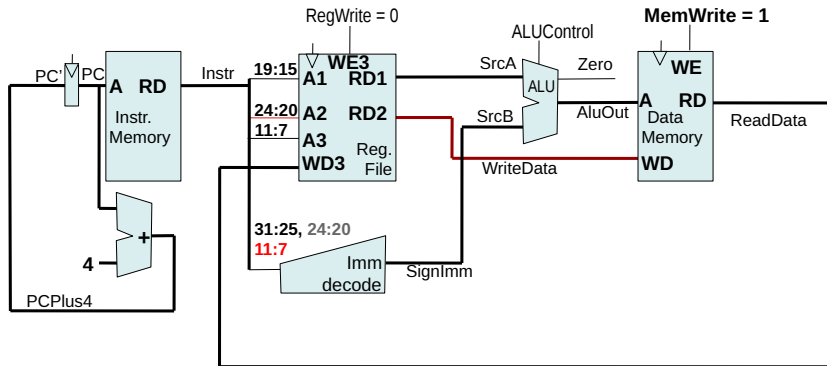
- Load**: A button in the top toolbar.
- Run**: A button in the top toolbar.
- Single Step**: A button in the top toolbar.
- Make**: A button in the top toolbar.
- Exceptions control**: A panel on the right side of the interface.
- Registers**: A panel on the left side showing the state of CPU registers.
- Assembler**: A panel on the left side showing the assembly code being executed.
- Editor**: A panel on the left side showing the source code being edited.
- Code**: A panel on the left side showing the current instruction being fetched.
- Terminal**: A panel on the right side showing the output of the program.
- Peripherals**: A panel on the right side showing the state of various peripherals.
- CPU core view**: A central diagram showing the internal structure of the CPU core, including the ALU, Register File, and Memory Controller.
  - single cycle**: A feature of the CPU core.
  - pipelined**: A feature of the CPU core.
- Cache**: A panel on the right side showing the state of the CPU cache.
- Data memory**: A panel on the right side showing the state of the CPU data memory.



# The Store Word Instruction – Implementation

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

Typ	31	30...25	24...21	20	19...15	14...12	11...8	7	6...0
S	imm[11:5]		rs2		rs1	fnct3	imm[4:0]		opcode



# Instruction for Two Registers Addition – ADD

**add** – **addition** – add content of two registers and store it to destination one

Description	Add together values in two registers ( $rs1 + rs2$ ) and stores the result in register $rd$
Operation	$[rd] \leftarrow [rs1] + [rs2]$
Syntax	<code>add rd, rs1, rs2</code>
Encoding	0000 000t tttt ssss s000 dddd d011 0011 t – rs2; s – rs1; d – rd

Example: Add values in registers 2 and 3 and store result into register 4:

`add x4, x2, x3`

0000 000 **t tttt** **ssss s** 000 **dddd d** 011 0011

0000 000 **0 0011** **0001 0** 000 **0010 0** 011 0011

*funct7*

3

2

*func3*

4

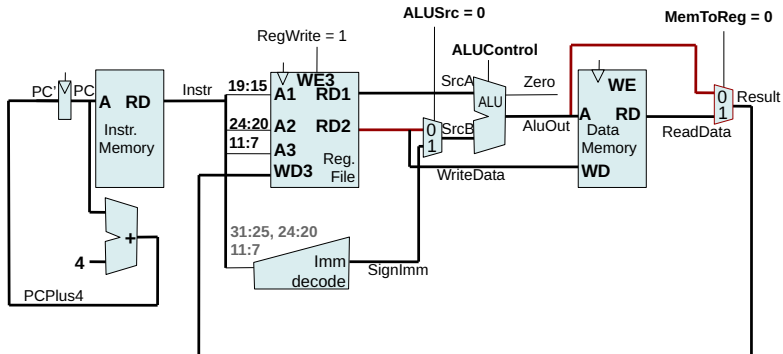
*opcode*

0x 00 31 02 33 – machine code `add x4, x2, x3`

# Two Registers Addition – Implementation

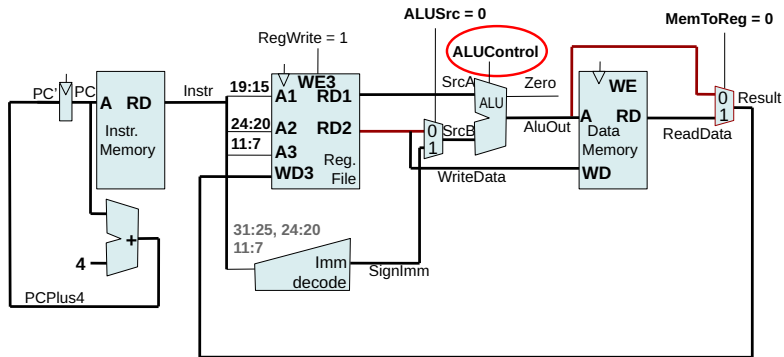
add: rs1, rs2 – sources, rd – destination register to store result

Typ	31	30...25	24...21	20	19...15	14...12	11...8	7	6...0
R		fnct7	rs2		rs1	fnct3		rd	opcode



# More Arithmetic Instructions – SUB, AND, OR, SLT

Another ALU operation selection (ALUcontrol) is only difference to addition. The data path is the same as for add instruction.



# Add Immediate Value to register – ADD

**addi** – **addition immediate** – add immediate constant value to register and store result into destination register

Description	Add rs1 and imm12 value and store result into rd
Operation	$[rd] \leftarrow [rs1] + imm12$
Syntax	addi rd, rs1, imm12
Encoding	iiii iiiiiiii ssss s000 dddd d001 0011 i – immediate; s – rs1; d – rd

Example: Increment register 7 value by 4 (store result into same register 7 as is source):

addi x7, x7, 4

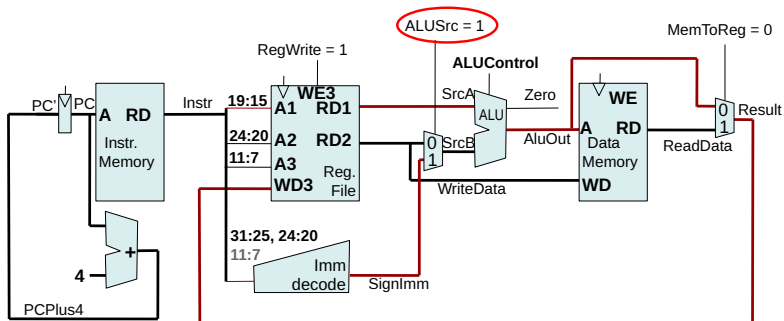
iiii iiiiiiii iiiiiiii ssss s 000 dddd d001 0011  
 0000 0000 0100 0011 1000 0011 1001 0011  
 └──────────┬──────────┬──────────┬──────────┬──────────┬──────────┬──────────┬──────────┘  
 0x004                    7            func3            7            opcode

0x 00 43 83 93 – machine code addi x7, x7, 4

# Operations with Immediate Value – ADDI, ORI, ANDI

**addi – add immediate:**  $[rd] \leftarrow [rs1] + \text{imm12}$  – add immediate constant value **imm** to register **rs1** and store result into destination register **rd**

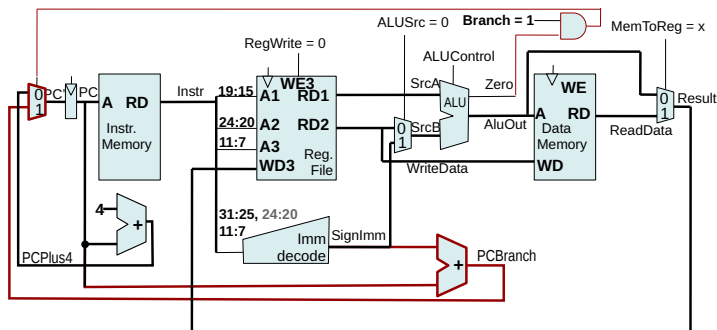
Typ	31	30...25	24...21	20	19...15	14...12	11...8	7	6...0
I	imm[11:0]				rs1	fnct3	rd		opcode



# Branch on Equal Instruction – BEQ

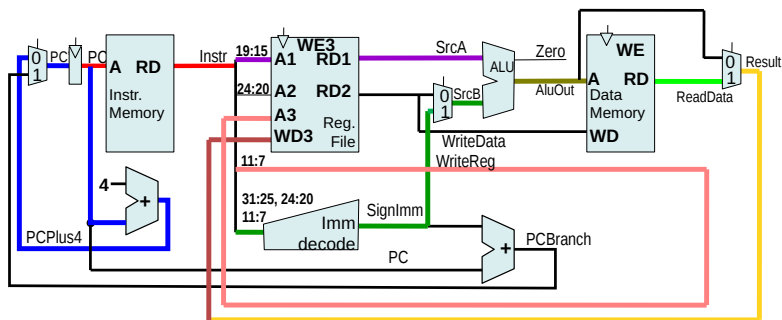
**beq** – branch if equal:  $[pc] \leftarrow [pc] + \text{SignImm}$  – adds offset *imm* in the range -4096 to +4094 to the actual instruction address (*pc*) and continue execution at that address *pc* if the values stored in *rs1* and *rs2* are equal

Typ	31	30...25	24...21	20	19...15	14...12	11...8	7	6...0
B	imm [12]	imm [10:5]	rs2		rs1	fnct3	imm[4:1]	imm [11]	opcode



Single cycle CPU – Throughput:  $IPS = IC / T$ 

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



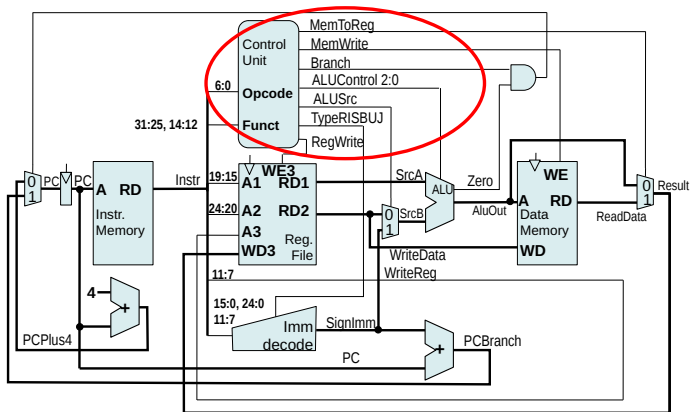
# Single cycle CPU – Longest Path

What is the maximal possible frequency of the CPU?

- The all combinational circuits in the longest path has to settle (propagate values) with enough setup time, the worst case is `lw` instruction:
  - $t_{PC} = 0,3 \text{ ns}$
  - $t_{Mem} = 20 \text{ ns}$
  - $t_{RFread} = 1,5 \text{ ns}$
  - $t_{ALU} = 2 \text{ ns}$
  - $t_{Mux} = 0,1 \text{ ns}$
  - $t_{RFsetup} = 0,1 \text{ ns}$
- The sum for the longest cycle
 
$$T_{CLK} = t_{PC} + t_{Mem} + t_{RFread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{RFsetup} = 44 \text{ ns}$$
- The corresponding frequency  $f_{CLK} = \frac{1}{T_{CLK}} = 22,7 \text{ MHz}$ , that is 22 700 000 instructions per seconds = 22,7 MIPS
- We need to speedup execution, the topics for lecture 5

# Processor Control Unit – Purpose

It transform machine instruction bits **opcode** and **fnct3,7** to the control signals **ALUControl**, **RegWrite**, **MemWrite**, **ALUSrc**, **MemToReg**, **Branch** and some more for complete CPU



# Processor Control Unit – Implementation

Podle Opcode lze nadefinovat výstupy řadiče:

Instruction	Opcode	Funct3	Funct7	ALUControl	ALUSrc	RegWrite	MemWrite	MemToReg	Branch
lw	0000011	010	-	+	1	1	0	1	0
sw	0100011	010	-	+	1	0	1	0	0
add	0110011	000	0000000	+	0	1	0	0	0
sub	0110011	000	0100000	-	0	1	0	0	0
slt	0110011	010	0000000	<	0	1	0	0	0
or	0110011	110	0000000		0	1	0	0	0
and	0110011	111	0000000	&	0	1	0	0	0
addi	0010011	000	-	+	1	1	0	0	0
beq	1100011	000	-	==	0	0	0	x	1

# Possible Ways to Implement Control Unit

- Realized directly by a logic circuit design:
  - Combinational logic (our example),
  - Sequential logic circuit - state machine, etc.
- Microprogrammed control unit (controller by microprogram):
  - The microprogram is stored in the control memory of the controller and consists of microinstructions.
  - The microprogram implements machine instructions visible to the programmer (add, sub, lw, xor, jmp, . . . ).
  - The instruction opcode specifies the address of the first microinstruction in control memory from which the microprogram for that instruction begins.
  - Each of the ISA instructions is executed using one or more microinstructions.
  - Advantage: a controller flexibility: ISA can be updated, extended by changing the microprogram
  - Disadvantages: more complex and unsuitable for pipelined processors where each stage executes a different instruction. Today used for sequential translation of such CISC instructions which are too complex to be translated into one or more RISC like microoperations directly by combinational logic