Design of a single-cycle RISC microarchitecture

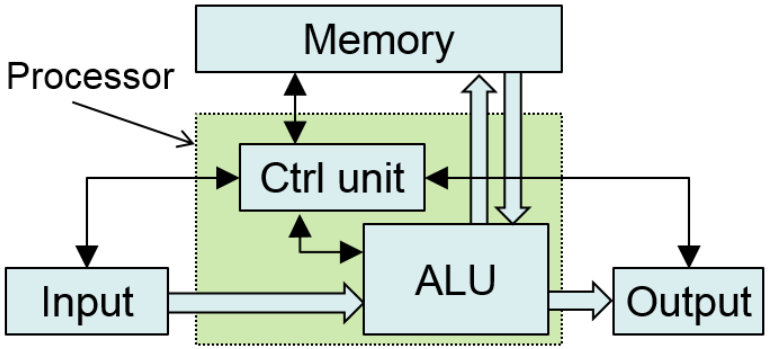
Michal Štepanovský, Pavel Tvrdík

Czech Technical University in Prague
Faculty of Information Technology
https://courses.fit.cvut.cz/BIE-APS

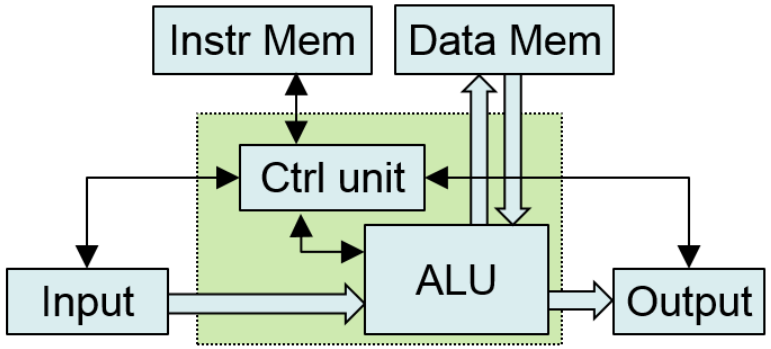
Architectures of Computer Systems (BIE-APS)
Winter Semester 2022, Lecture 3

(Version Timestamp: 1. 11. 2022 16:47)

Computer architecture: von Neumann vs. Harvard



von Neumann architecture



Harvard architecture

- 5 functional units: control unit, ALU, memory, input & output devices.
- A computer architecture is universal, it is **independent** on solved problems. It provides a mechanism to load a program into memory. The program controls what the computer does with data and which problem it solves.
- Instruction (program) and data memory:
  - **unified** $\Rightarrow$ **von Neumann**,
  - **separated** $\Rightarrow$ **Harvard**.
- The main memory consists of **cells** of the same size that are sequentially **numbered/indexed**, each cell has its **address**.
- Instructions are stored in memory sequentially.
- Control flow instructions change the Program Counter to other than subsequent instruction.

The goal of this lecture I

- Today's computers have unified main memory and the advantage of separated instruction and data memory is achieved by using a separated L1 data and L1 instruction cache memory.
- Therefore, the goal of this lecture is

to design a **single-cycle microarchitecture** of a simple computer consisting of a **CPU** and **separated instruction and data memory**.

- A **single-cycle** microarchitecture is **minimal** in a sense that all instructions are processed in a single clock cycle.
- The next Lecture 4 will be devoted to a more realistic and more complex **pipelined microarchitecture**.

Why RISC-V Architecture?

- There is an increasing demand for custom processors to meet the power and performance requirements of specific applications.
- Widely used ISAs (x86, x86-64, ARM, etc.) are licensed **intellectual property** (IP) of some companies. For example, if you wish to design an ARM-compatible processor, you need to license it from Arm Ltd., the owners of the ARM ISA IP.
- RISC-V is a free and open ISA (no fees to use it).
- RISC-V follows the RISC principles.
- RISC-V is actually a **family of ISAs**.
- The RISC-V ISA family is
  - **parameterized** and
  - **extensible** with custom-defined instructions.
- Thus, RISC-V is suitable for the whole scale of computers, starting from embedded systems up to high-performance servers.
- Free RISC-V processor cores (microarchitectures) in VHDL/Verilog are available!
- RISC-V is simple enough to be used in computer architecture courses.

RISC-V ISA Family I

- XLEN: width of integer registers in bits (32 or 64, in the future 128).
- I/E: **I**nteger/**E**mbedded (RV32E has only 16 registers).
- EXTENSIONS:
  - M: **M**ultiplication and division instructions
  - A: **A**tomic instructions
  - C: **C**ompressed instructions (i.e., shorter instructions)
  - F: Single-Precision **F**loating-Point support
  - D: **D**ouble-Precision Floating-Point support
  - Q: **Q**uad-Precision Floating-Point support
  - Zicsr: Control and Status Register instructions
  - Zifencei: Instruction-Fetch Fence instructions
  - and others

RISC-V ISA Family II

- Any RISC-V processor must implement the **base integer ISA**, which is a predefined set of 40 basic integer instructions.
- The differences between RV32I ISA and RV32E ISA:
  - RV32I has 32 32-bit GPRs, whereas RV32E has only 16 32-bit GPRs.
- RV64I is RISC-V ISA with 32 64-bit GPRs, its integer instructions are a **superset** of the base integer ISA.
- RV64E does not exist, it is not defined at the present time.
- Integer GPRs of RV32I/RV64I are denoted by `x0`...`x31`.
- Integer GPRs of RV32E are denoted by `x0`...`x15`.
- Register `x0` is always **hardwired to zero**.

Example

- **RV64IMAFDZicsr_Zifencei** is a 64-bit ISA with many extensions, e.g., double-precision floating point instructions and fence instructions for synchronization of multi-core computation.
- **RV32EC** is a 32-bit ISA suitable for embedded systems supporting only base integer instructions with the option to encode them to 16 bits where possible.

PicoRISC-V ISA

In the BIE-APS course, we will use the following subset of RV32I ISA, called **picoRISC-V ISA**.

Definition (picoRISC-V ISA)

- Instructions to read and write a value from/to the data memory: `lw` and `sw`.
- Arithmetic and logic instructions: `add`, `addi`, `sub`, `and`, `or`, and `slt`.
- Control flow instructions:
  - Conditional branching instruction `beq`.
  - Subroutine call instructions `jal` and `jalr`

Note

- This small instruction set is sufficient for writing interesting programs.
- The instructions `jal` and `jalr` include the functionality of unconditional jump instructions: `j` (jump) and `jr` (jump register), and of the return from a subroutine: `ret` (return), see Slide 35.

picoRISC-V ISA instructions in assembly language

Instruction Syntax	Instruction Semantics
`lw rd,` $\texttt{imm}_{11:0}$`(rs1)`	rd $\leftarrow$ Mem[[rs1] + $\text{imm}_{11:0}$];
`sw rs2,` $\texttt{imm}_{11:0}$`(rs1)`	Mem[[rs1] + $\text{imm}_{11:0}$] $\leftarrow$ [rs2];
`addi rd, rs1,` $\texttt{imm}_{11:0}$	rd $\leftarrow$ [rs1] + $\text{imm}_{11:0}$;
`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,` $\texttt{imm}_{12:1}$	if [rs1] == [rs2] go to [PC]+{$\text{imm}_{12:1}$,'0'}; else go to [PC]+4;
`jal rd,` $\texttt{imm}_{20:1}$	rd $\leftarrow$ [PC]+4; go to [PC]+{$\text{imm}_{20:1}$,'0'};
`jalr rd, rs1,` $\texttt{imm}_{11:0}$	rd $\leftarrow$ [PC]+4; go to [rs1]+$\text{imm}_{11:0}$;

- `rd` = **r**egister **d**estination, `rs1`(2) = **r**egister **s**ource1(2), `imm` = **imm**ediate op.
- Each immediate operand `imm` is labeled with the range ($\texttt{imm}_{\texttt{high:low}}$) of bit positions in the 32-bit immediate value being produced.
- If `low` $>$ 0, `imm` is **zero-padded**. This is explicitly indicated, e.g., {$\texttt{imm}_{20:1}$,'0'}.
- `imm` is always **sign-extended** to create a 32-bit operand.

RISC-V GPR names and recommended utilization

In assembly language, registers can be accessed by using their architectural name (x0...x31) or by using their ABI[1] name.
In BIE-APS, we will use both options how to specify the register.

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	—
x5–7	t0–2	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

[1] Application binary interface (ABI) is a low-level interface between two programs.

Program Examples I

Summation of values in registers:

```
addi x1,x0,4      // x1 ← 4;
addi x2,x0,20     // x2 ← 20;
add  x3,x1,x2     // x3 ← [x1]+[x2];
```

Incrementation of memory cell with address 12:

```
lw   x1,12(x0)    // x1 ← Mem[12];
addi x1,x1,1      // x1 ← [x1]+1;
sw   x1,12(x0)    // Mem[12] ← [x1];
```

Assignment conditioned by different register contents:

```
beq  x1,x2,L1     // if [x1]==[x2] go to L1;
addi x2,x0,5      // x2 ← 5; (Assigned only if [x1]≠[x2])
L1:
```

Program Examples II

Subroutine call (gcd = greatest common divisor):

```
int gcd (int n1, int n2){
  while(n1!=n2){
    if(n1 > n2)
      n1 -= n2;
    else
      n2 -= n1;
  }
  return n1;
}

void main(){
  register int n1 = 25;
  register int n2 = 15;
  gcd(n1, n2);
}
```

```
gcd:
  beq a0,a1,done  // Are we done?
  slt t0,a0,a1    // t0 ← [a0]<[a1];
  beq t0,x0,L     // [a0]<[a1]?
  sub a1,a1,a0    // a1 ← [a1]−[a0];
  beq x0,x0,gcd   // go to gcd;
L:sub a0,a0,a1    // a0 ← [a0]−[a1];
  beq x0,x0,gcd   // go to gcd;
done:
  jalr x0,x1,0    // return;

main:
  addi a0,x0,25   // a0 ← 25; (n1)
  addi a1,x0,15   // a1 ← 15; (n2)
  jal  x1,gcd     // Call gcd
```

Compilation and machine coding

Program in C

```
int pow = 1;
int x = 0;
while(pow != 128)
{
  pow = pow*2;
  x = x + 1;
}
```

Program in JSA

```
addi s0,x0,1
addi s1,x0,0
addi t0,x0,128
while:
  beq s0,t0,done
  slli s0,s0,1
  addi s1,s1,1
  j while
done:
```

Machine code[2]

```
00 10 04 13
00 00 04 93
08 00 02 93
00 54 08 63
00 14 14 13
00 14 84 93
ff 5f f0 6f
```

[2] Machine code in hexadecimal coding.

Machine encoding of RV32I/picoRISC-V ISA instructions

At the **machine code level**, each instruction is encoded into 32b word using one of 6 formats:

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

- 5-bit encoding of **rs1**, **rs2**, **rd** allows to encode 32 GPRs.
- **opcode** = operation code, **funct3**,**funct7** = extended opcode.
- Bit 31 in I,S,B,U,J-type formats is the **sign bit** for extension of the immediate operand to 32 bits.
- That is why the mapping of the bits of the immediate operand into the machine code is so complicated:
  the msb is always bit 31 of the instruction word.

Encoding of immediate operands and branch instructions

- Branching instructions (B,J-type) use the branching address that needs to be (at least) multiple of 2 (i.e., the lsb of the branch target address must be zero).
- This is due to the requirement to support both 32 bit and 16 bit (compressed) instructions.
- A usual solution is to shift the immediate operand left by one bit **in hardware**.
- However, RISC-V ISA encodes this shift operation **in instruction itself**.
- We will explain this solution on the next slide by comparing S-type and B-type formats.

Encoding of immediate operands I

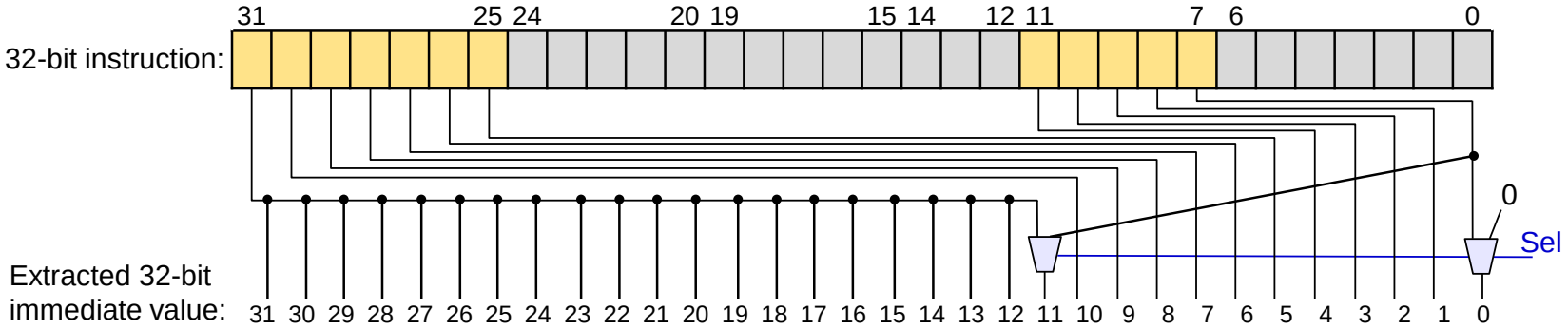
Comparison of S and B formats:

31 25	24 20	19 15	14 12	11 7	6 0	
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type

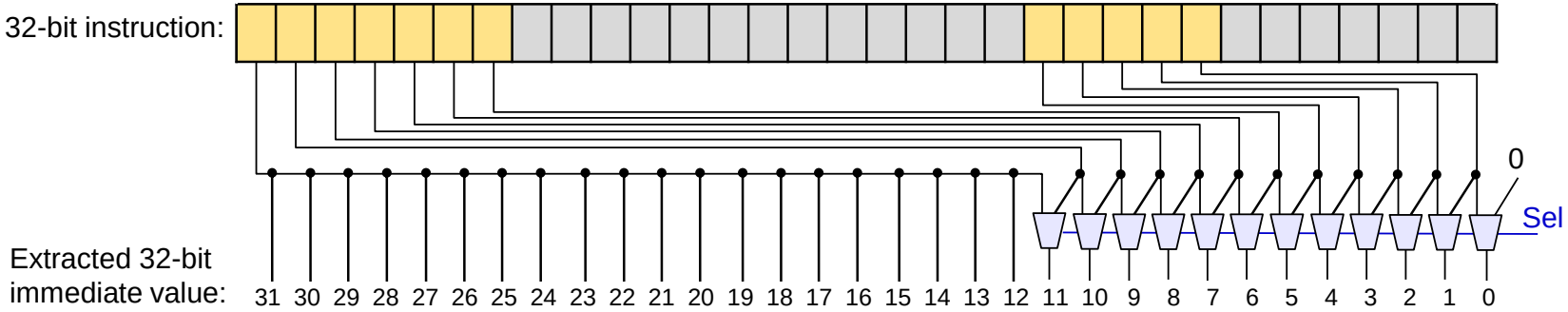
- The only difference between the S-type and B-type formats is that the B-type 12-bit immediate field encodes branch offsets in multiples of 2.
- Therefore, imm[0] is not part of the instruction encoding, because it is always 0, and thus, the instruction contains bits imm[12:1].
- In order to minimize hardware cost, it is required that the most of the immediate operand bits in the instruction stay at the same positions.
- The first option would be to encode imm[12] instead of imm[0], i.e., in bit 7 of the instruction (inst[7]).
- However, it is also required that the sign bit is encoded in bit 31 of the instruction (inst[31]).
- Thus, imm[12] has to be placed in inst[31]. And therefore, inst[11] is placed at the original place of imm[0], i.e., inst[7].

Encoding of immediate operands II

Extracting S-immediate and B-immediate value from the instruction:



Conventional approach would use only 1 instruction type and HW shifting:



- Shifting operation in HW (multiplication by 2) requires more multiplexors.
- This is even more crucial when multiple instruction formats are required (such as U-type with 20-bit imm operand).
- The RISC-V imm. encoding adds just a negligible time to the program compilation.

Encoding of immediate operands III

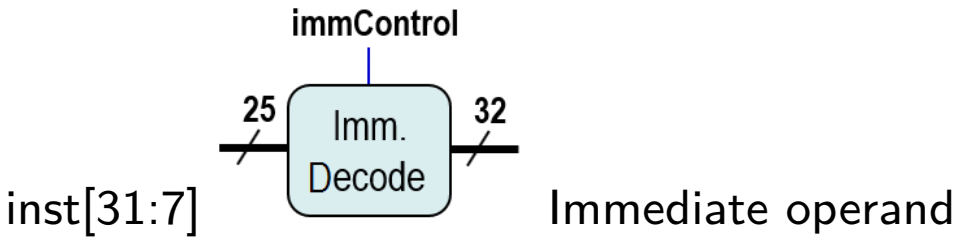
According to previous slides, there are 5 ways to construct immediate operands:

31 … 0						
— inst[31] —				inst[30:25]	inst[24:21]	inst[20]	I-immediate
— inst[31] —				inst[30:25]	inst[11:8]	inst[7]	S-immediate
— inst[31] —			inst[7]	inst[30:25]	inst[11:8]	0	B-immediate
inst[31]	inst[30:20]	inst[19:12]	— 0 —				U-immediate
— inst[31] —		inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J-immediate

- The fields are labeled with the **instruction bit positions** used to construct the value of the immediate operand.
- The mapping of the bits of a immediate operands to the instruction 32-bit word in all types of formats is chosen to maximize the mapping overlap.
- For instance, bit 0 of the 32-bit immediate value may come from inst[20], inst[7], or it is set to 0.
- Bit inst[31] is always the msb used in sign-extension to 32-bit value.

Decoding of immediate operands

- The instruction type (R-, I-, S-, B-, J-type) is encoded in bits inst[6:0], called **opcode**.
- Remaining bits inst[31:7] in I-, S-, B-, J-type are used to encode the immediate operand.
- However, the encoding of the immediate operand in the machine word is quite diverse among those formats as we could just see.
- Therefore, to extract and construct a 32-bit number out of this encoding is rather complicated.
- In our design, we will consider a special hardware decoding unit called **Immediate Decoder** to construct a 32-bit immediate operand from the instruction word.



where the control signal `immControl` is derived from the instruction format.

Encoding of picoRISC-V instructions I

opcode is used to encode a specific instruction or a group of related instructions:

opcode	Meaning for RV32I	For our picoRISC-V
0110011	R-type	`add`, `sub`, `slt`, `or`, `and`
0010011	I-type: ALU-imm	`addi`
0000011	I-type: Memory load	`lw`
0100011	S-type: Memory store	`sw`
1100011	B-type: Branch	`beq`
1101111	J-type: jal	`jal`
1100111	I-type: jalr	`jalr`

R-type instructions with opcode 0110011 are distinguished with **funct7** and **funct3**.

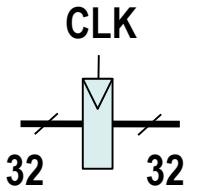
funtc7	**funtc3**	Instruction
0000000	000	`add`
0100000	000	`sub`
0000000	010	`slt`
0000000	110	`or`
0000000	111	`and`

Encoding of picoRISC-V instructions II

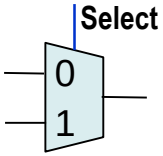
Instruction encoding of all picoRISC-V instructions:

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]		rs1	010	rd	0000011	`lw` (I)
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	`sw` (S)
0000000	rs2	rs1	000	rd	0110011	`add` (R)
0100000	rs2	rs1	000	rd	0110011	`sub` (R)
0000000	rs2	rs1	010	rd	0110011	`slt` (R)
0000000	rs2	rs1	110	rd	0110011	`or` (R)
0000000	rs2	rs1	111	rd	0110011	`and` (R)
imm[11:0]		rs1	000	rd	0010011	`addi` (I)
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	`beq` (B)
imm[20|10:1|11|19:12]				rd	1101111	`jal` (J)
imm[11:0]		rs1	000	rd	1100111	`jalr` (I)

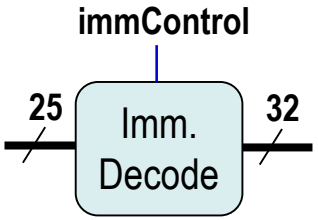
CPU building blocks (recap from BIE-SAP)



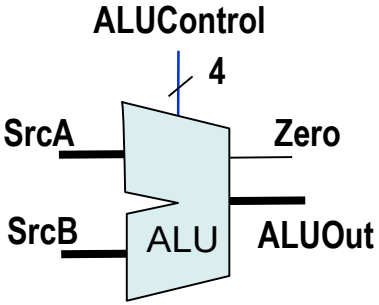
Register



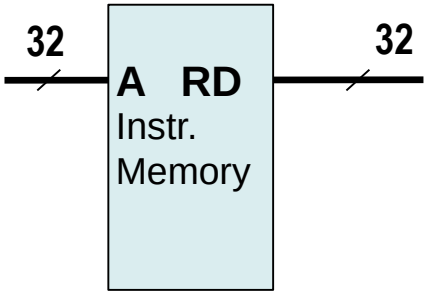
Multiplexor

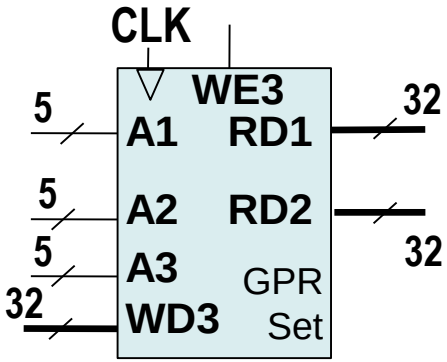


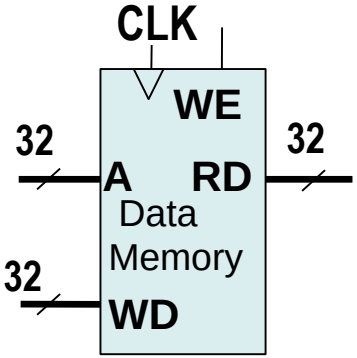
Immediate operand extraction



Arithmetic-Logic Unit



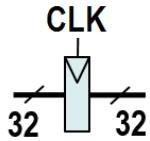




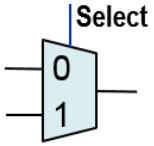
Write at the rising edge of CLK when WE = 1

Read after “enough time” for data propagation

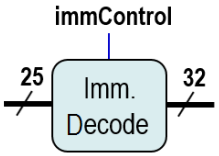
Explanation of building blocks I



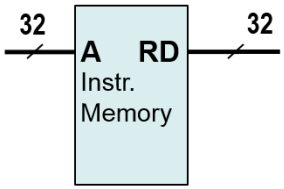
Register triggered (inputs are transferred to outputs) with a clock positive edge. The register state is latched until the next clock positive edge.



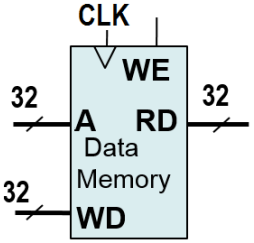
Multiplexer switches one of the inputs to the output on the basis of the Select signal.



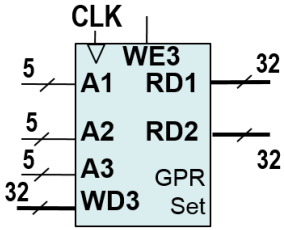
Extracting and building up an immediate operand from the instruction.



Instruction memory provides an instruction from address A to port RD. Memory read is combinational.



2-port data memory. Port RD serves to read data (combinational), port WD serves to write data. Data write needs the WE control input on and is triggered with the positive clock edge (CLK).



3-port GPR set of 32 32-bit registers. Inputs A1 and A2 address read ports RD1 and RD2. Reg. reading is combinational. Write port WD3 are addresses with input A3. Write into a register is enabled with the WE3 control input on and is triggered with the positive clock edge (CLK).

Paths

	Thick (black) line: 32-bit data path (= 32 parallel wires).
	Thin (black) line: Data path other than 32 bits.
	Thin blue line: Control signal from the control unit.
	Nongalvanic wire crossing.
	Wire/path split (galvanic connection).
15:0	Selection of bits 15:0 from a data path.

Instruction `lw` format, syntax, and semantics

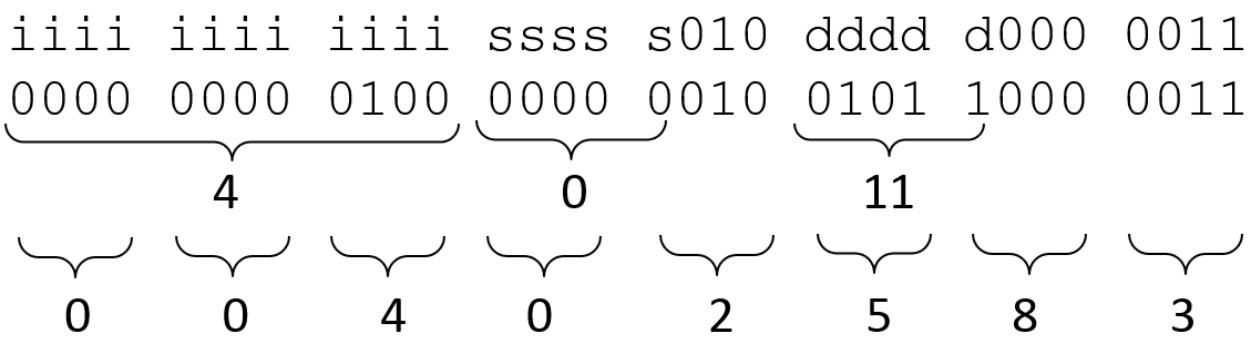
`lw` = load word from data memory into a register

Syntax:	`lw rd,` $\texttt{imm}_{11:0}$`(rs1)`
Semantics:	rd $\leftarrow$ Mem[[rs1] + $\text{imm}_{11:0}$];
Encoding:	`iiii iiii iiii ssss s010 dddd d000 0011`

The base address offset is encoded as the 12-bit immediate operand.

Example 1

`lw x11,0x4(x0)` = Load word from memory address 0x4 into reg. x11.



Hexadecimal code of `lw x11,0x4(x0)` in machine language: 0x00402583.

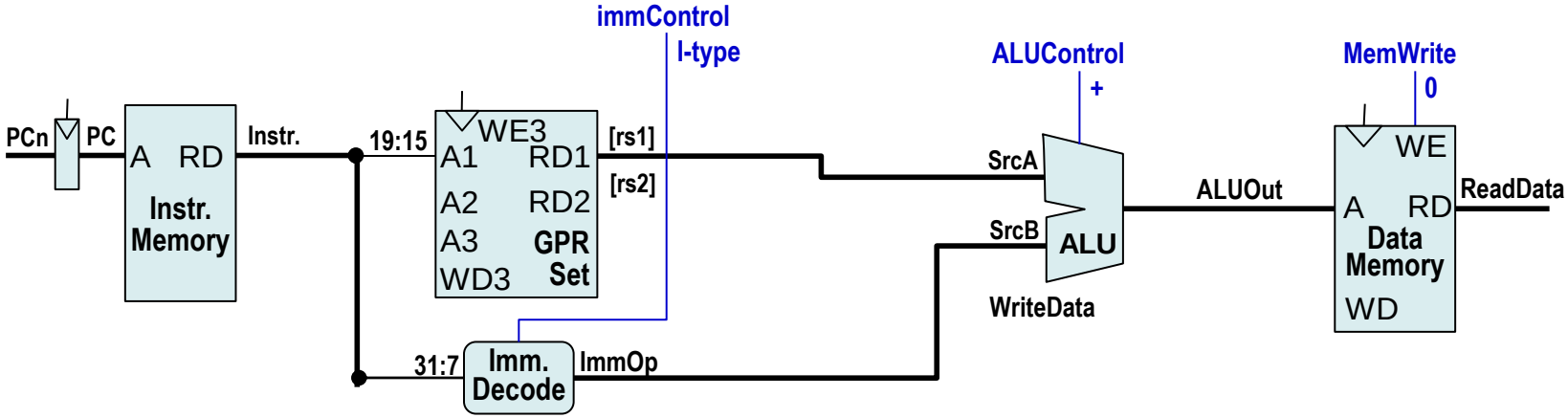
Note that in this lecture, the **hexadecimal coding** is prefixed with 0x.

Single-cycle CPU — implementation of instruction `lw` I

`lw rd, imm`$_{11:0}$`(rs1)` rd ← Mem[[rs1] + imm$_{11:0}$];

rs1 = base address register, **imm** = offset, **rd** = destination register.

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
imm[11:0]	rs1	funct3	rd	opcode	I-type

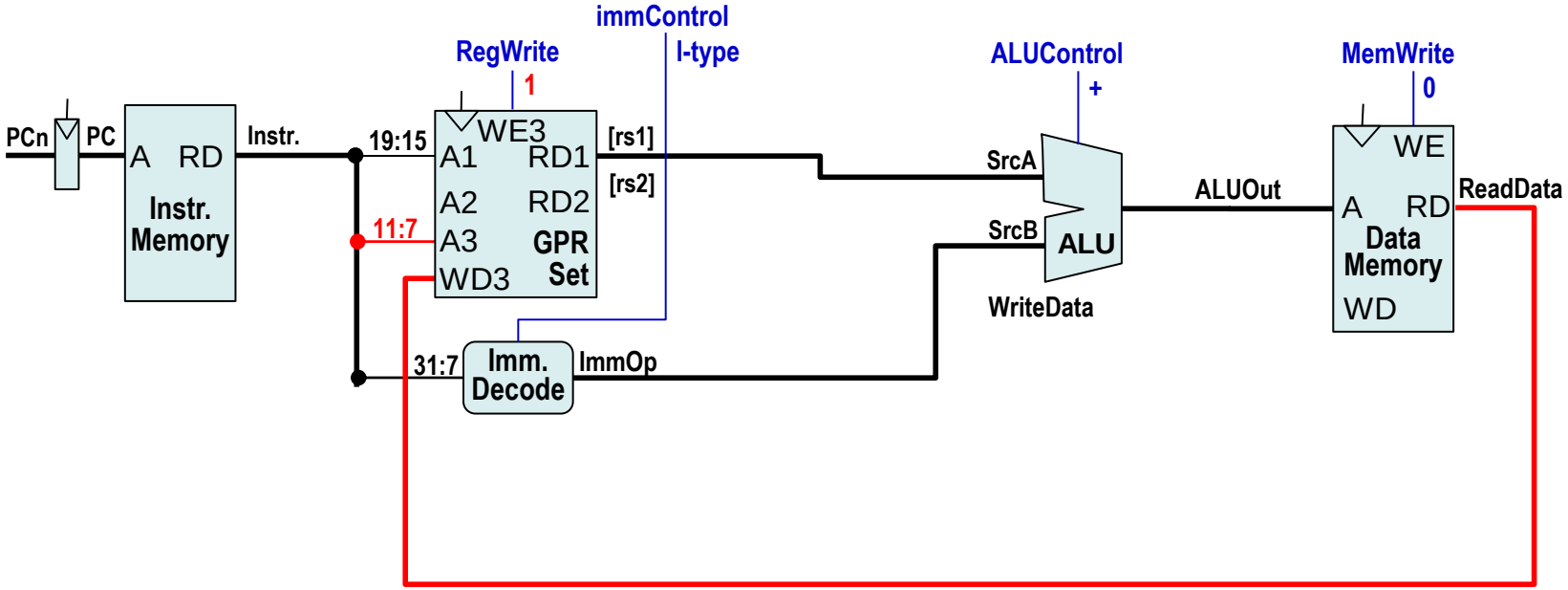


Single-cycle CPU — implementation of instruction `lw` II

`lw rd, imm`$_{11:0}$`(rs1)` rd ← Mem[[rs1] + imm$_{11:0}$];

rs1 = base address register, **imm** = offset, **rd** = destination register.

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
imm[11:0]	rs1	funct3	rd	opcode	I-type

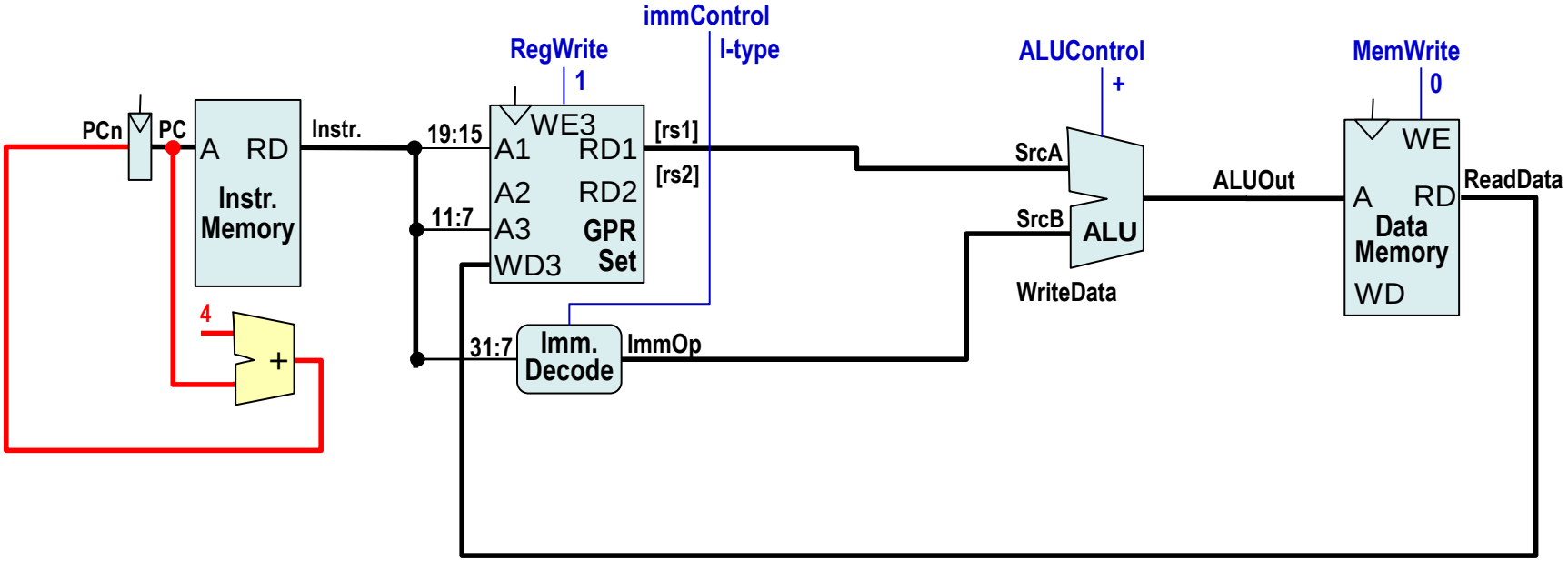


Single-cycle CPU — implementation of instruction `lw` III

`lw rd, imm`$_{11:0}$`(rs1)` rd ← Mem[[rs1] + imm$_{11:0}$];

rs1 = base address register, **imm** = offset, **rd** = destination register.

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
imm[11:0]	rs1	funct3	rd	opcode	I-type

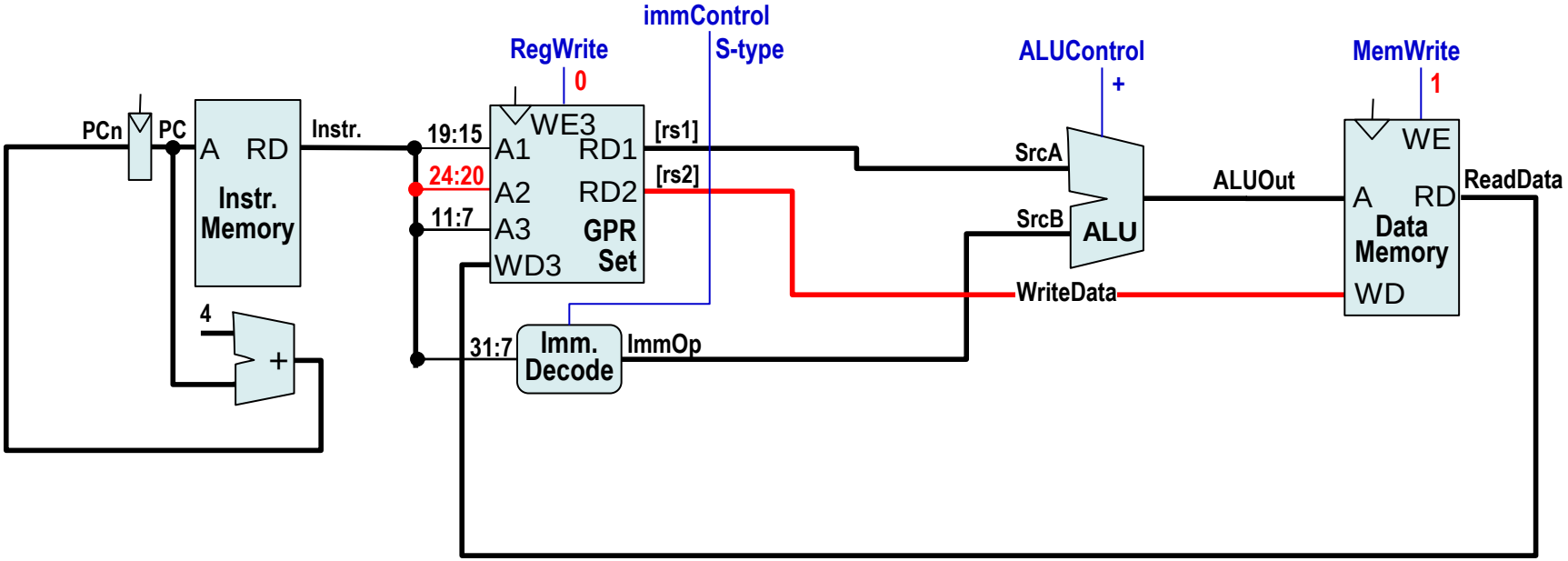


Single-cycle CPU — implementation of instruction `sw`

`sw rs2, imm`$_{11:0}$`(rs1)` $\quad$ Mem[[rs1] + imm$_{11:0}$] ← [rs2];

rs1 = base address register, **imm** = offset, **rs2** = source register.

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0	
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

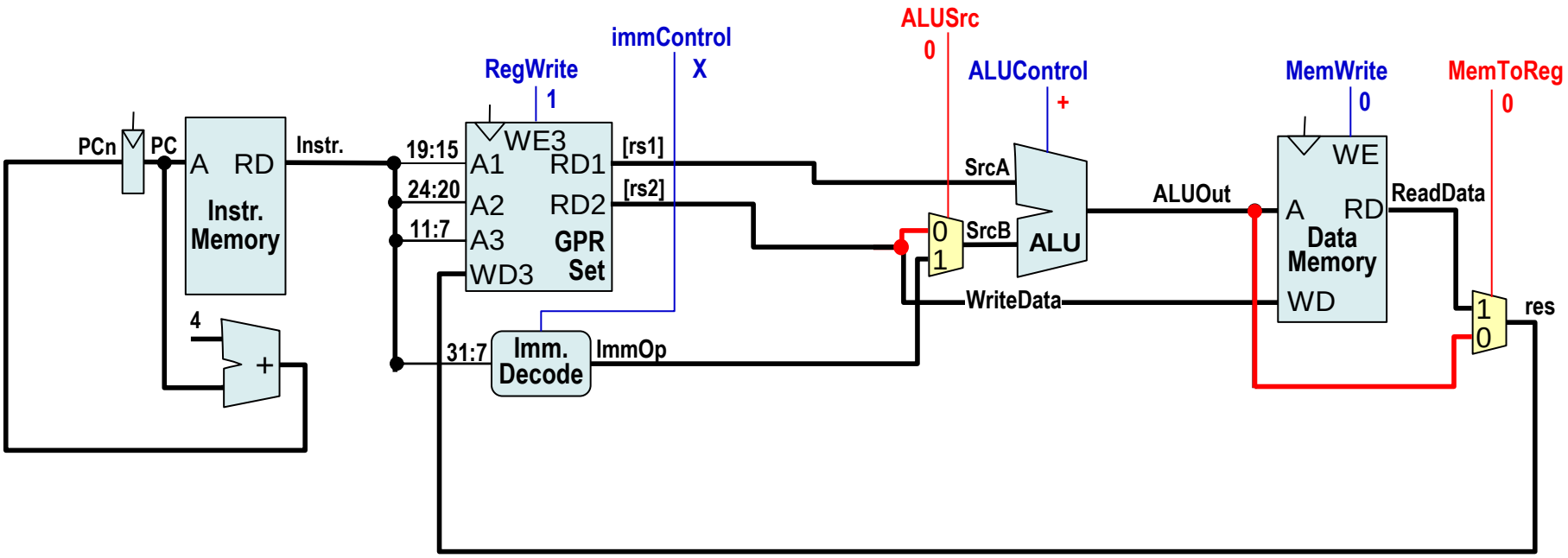


Single-cycle CPU — implementation of instruction add

`add rd, rs1, rs2` rd ← [rs1] + [rs2];

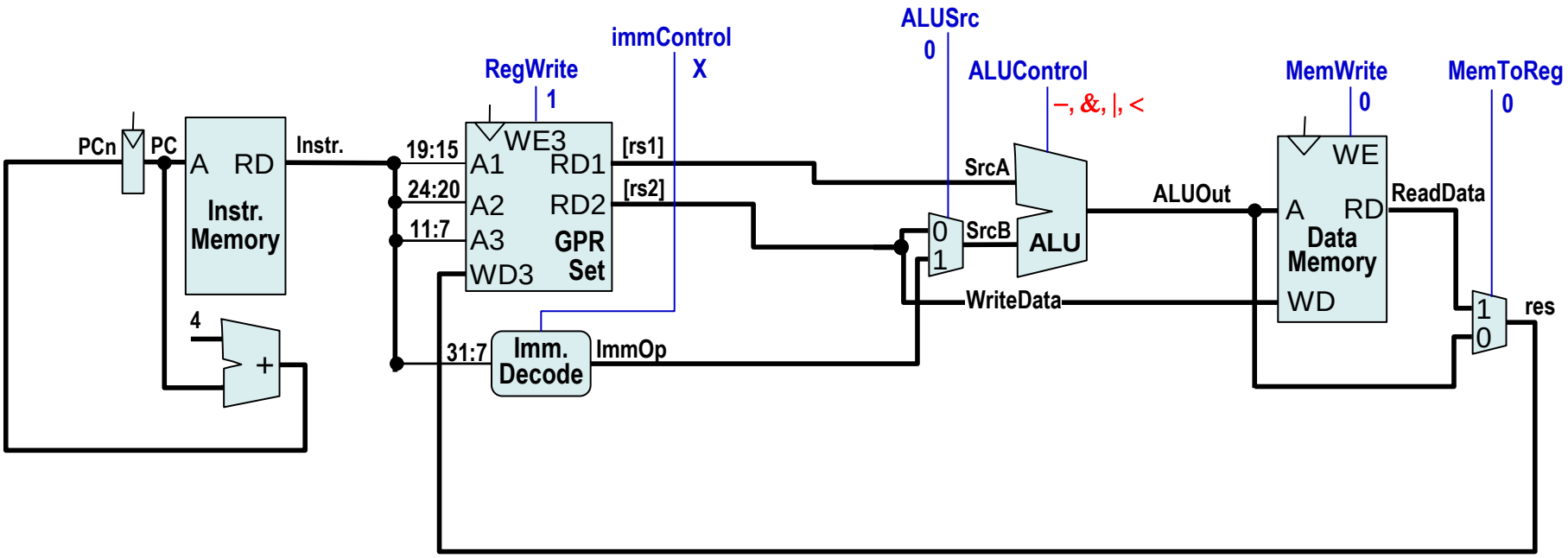
rs1, **rs2** = source reg., **rd** = destination reg., **funct7** = add operation

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type



Single-cycle CPU — impl. of instr. sub, and, or, slt

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

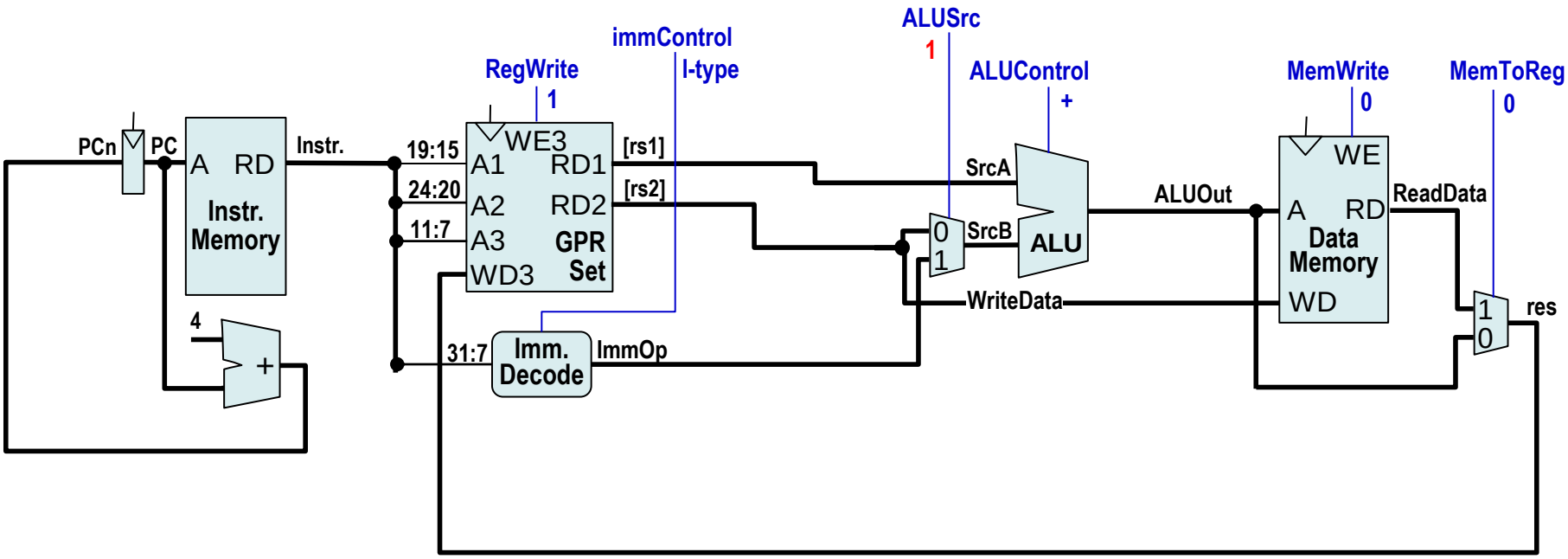


Single-cycle CPU — implementation of instruction addi

`addi rd, rs1, imm`$_{11:0}$ rd ← [rs1] + imm$_{11:0}$;

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
imm[11:0]	rs1	funct3	rd	opcode	I-type

Instruction `addi` is of type I, similarly as `lw`. The data path therefore already exists. The control path must be updated.

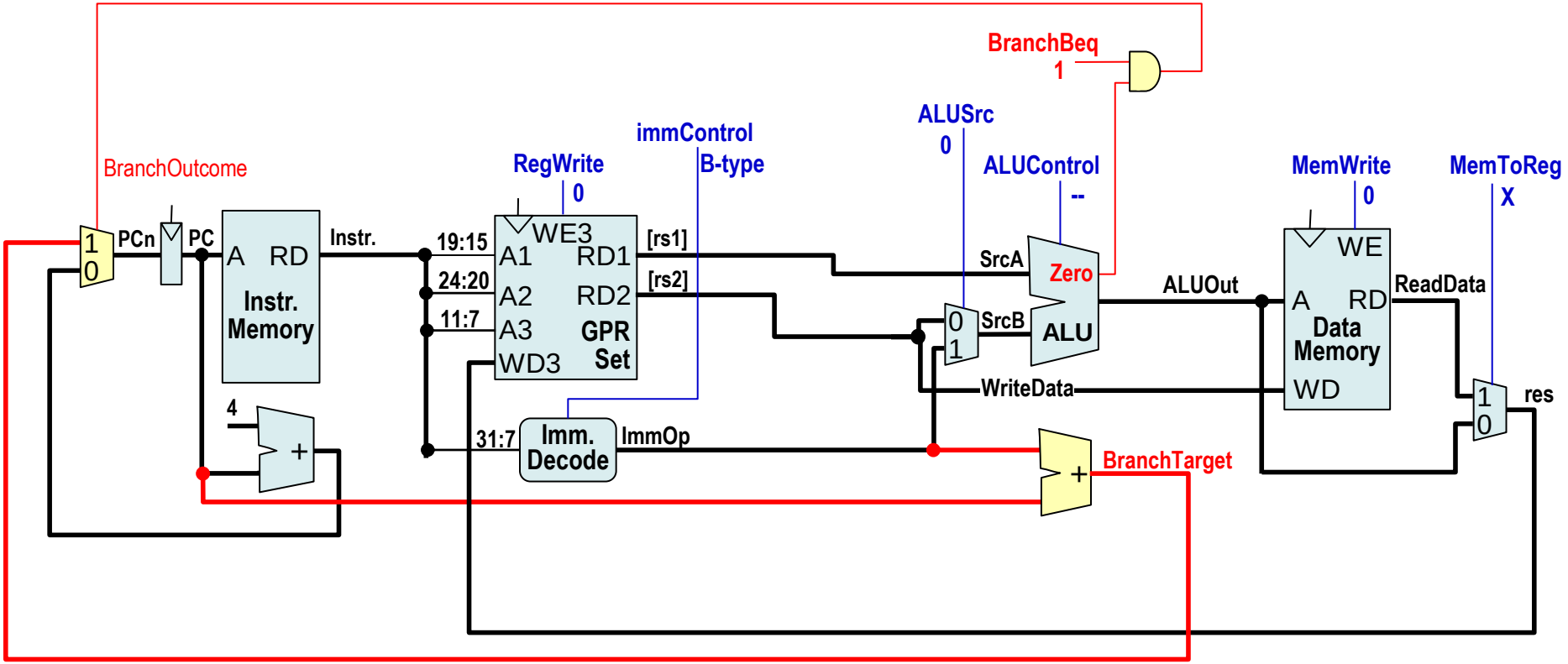


Single-cycle CPU — implementation of instruction beq

`beq rs1, rs2, imm`$_{12:1}$ if [rs1] == [rs2] go to [PC]+{imm$_{12:1}$,'0'};
else go to [PC]+4;

Here, {imm$_{12:1}$,'0'} represents the PC-relative offset. The reason why immediate operand is zero-padded with only one bit is that in compressed format (e.g. RV32IC) the instruction length can be 2 Bytes only.

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0	
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type

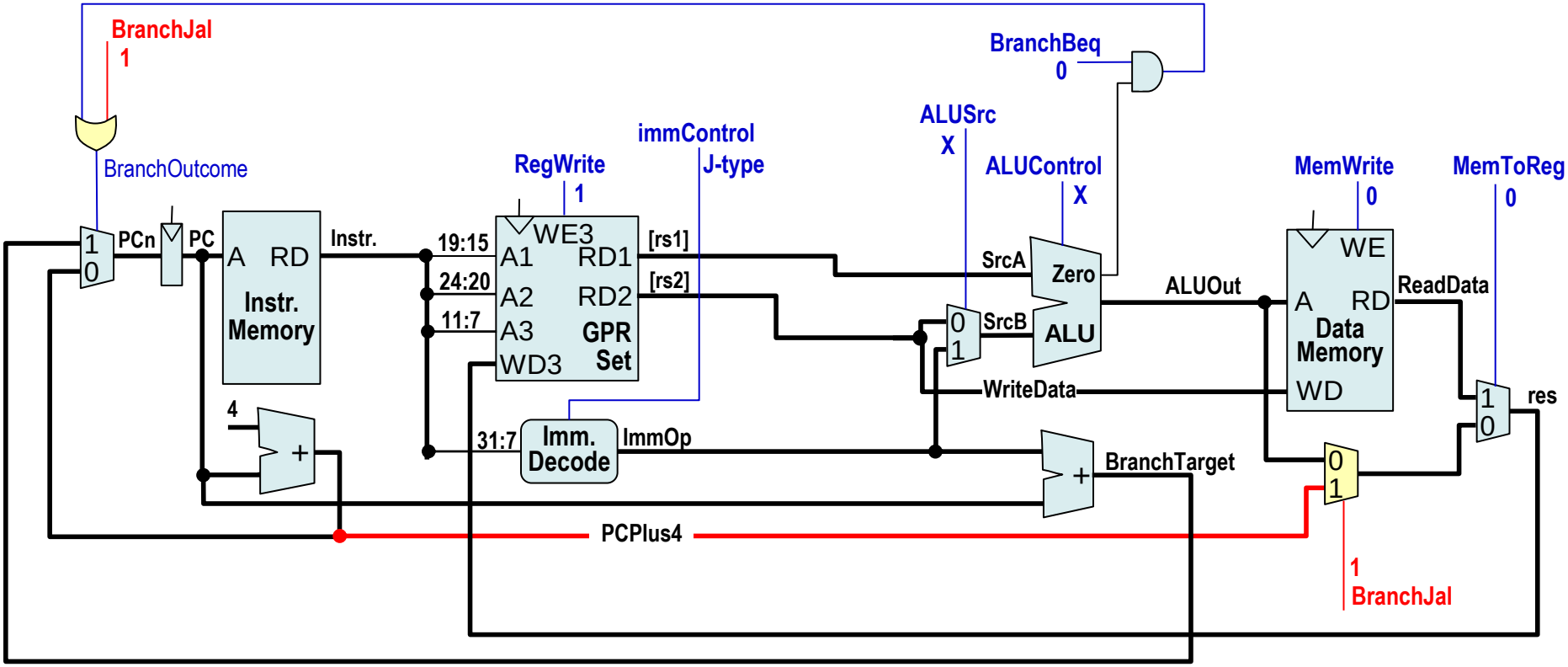


Single-cycle CPU — implementation of instruction `jal`

`jal rd, imm`$_{20:1}$ rd ← [PC]+4; go to [PC]+{imm$_{20:1}$,'0'};

Here, again {imm$_{20:1}$,'0'} represents the PC-relative offset.

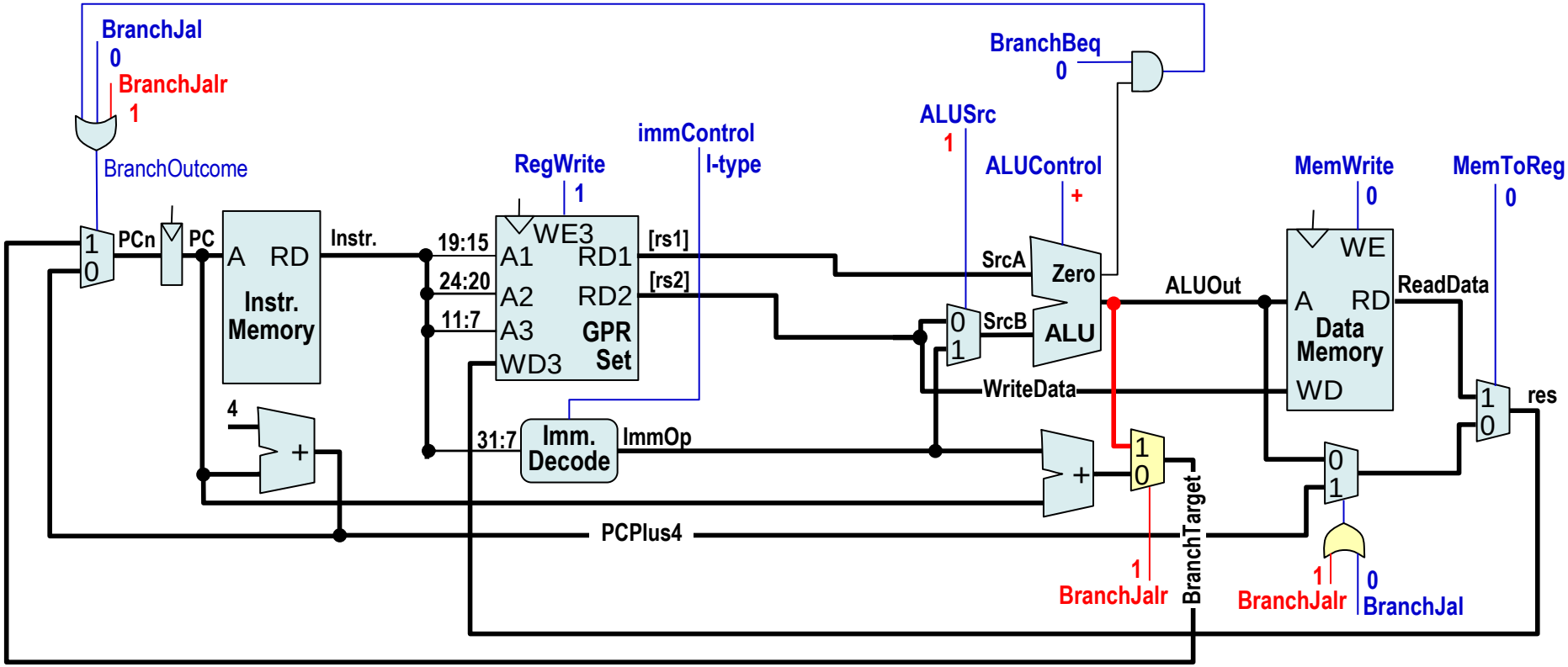
31 … 12	11 … 7	6 … 0	
imm[20|10:1|11|19:12]	rd	opcode	J-type



Single-cycle CPU — implementation of instruction `jalr`

`jalr rd, rs1, imm`$_{11:0}$ rd ← [PC]+4; go to [rs1]+imm$_{11:0}$;

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
imm[11:0]	rs1	funct3	rd	opcode	I-type



Why are `jal` and `jalr` so important?

The instructions `jal` and `jalr` include the functionality of unconditional jump instructions: `j` (jump) and `jr` (jump register), and the return from the subroutine: `ret` (return). From `j`, `jr` and `ret` the following semantics is required:

Syntax	Semantics
`j` $\mathtt{imm}_{20:1}$	go to [PC]+{$\mathtt{imm}_{20:1}$,'0'};
`jr rs1`	go to [rs1];
`ret`	go to [x1];

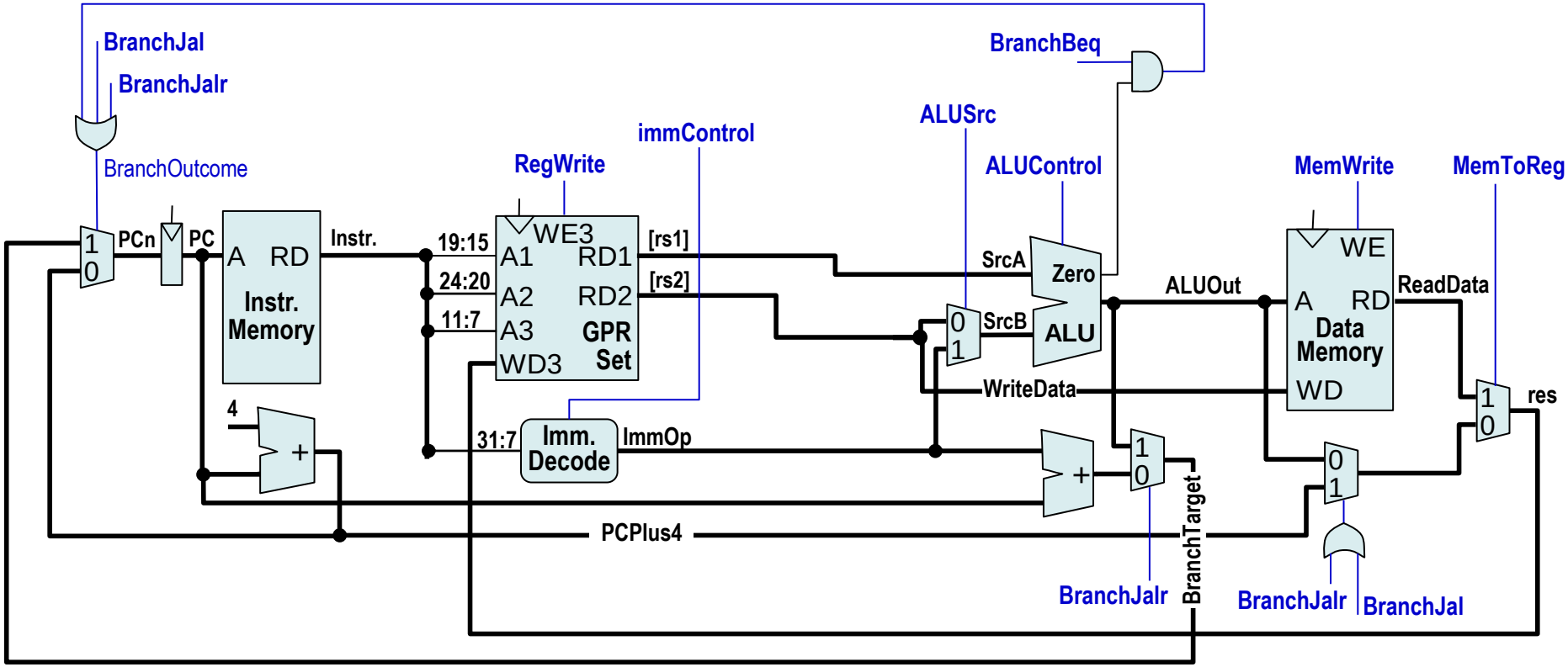
In fact, this is already implemented.

Syntax	Implementation	Semantics
`j` $\mathtt{imm}_{20:1}$	`jal x0,`$\mathtt{imm}_{20:1}$	x0 ← [PC]+4; go to [PC]+{$\mathtt{imm}_{20:1}$,'0'};
`jr rs1`	`jalr x0,rs1,0`	x0 ← [PC]+4; go to [rs1]+0;
`ret`	`jalr x0,x1,0`	x0 ← [PC]+4; go to [x1]+0;

- Subroutine call according to RISC-V calling convention is `jal x1,`$\mathtt{imm}_{20:1}$, see Slide 9. Therefore, implementation of `ret` expects that `x1` contains the return address.
- Another possibility for `j` is `beq`: `j` $\mathtt{imm}_{12:1}$ = `beq x0,x0,`$\mathtt{imm}_{12:1}$.
- Thus, there is no need to implement `j`, `jr` and `ret` in our microarchitecture.

Clock frequency of a single-cycle microarchitecture

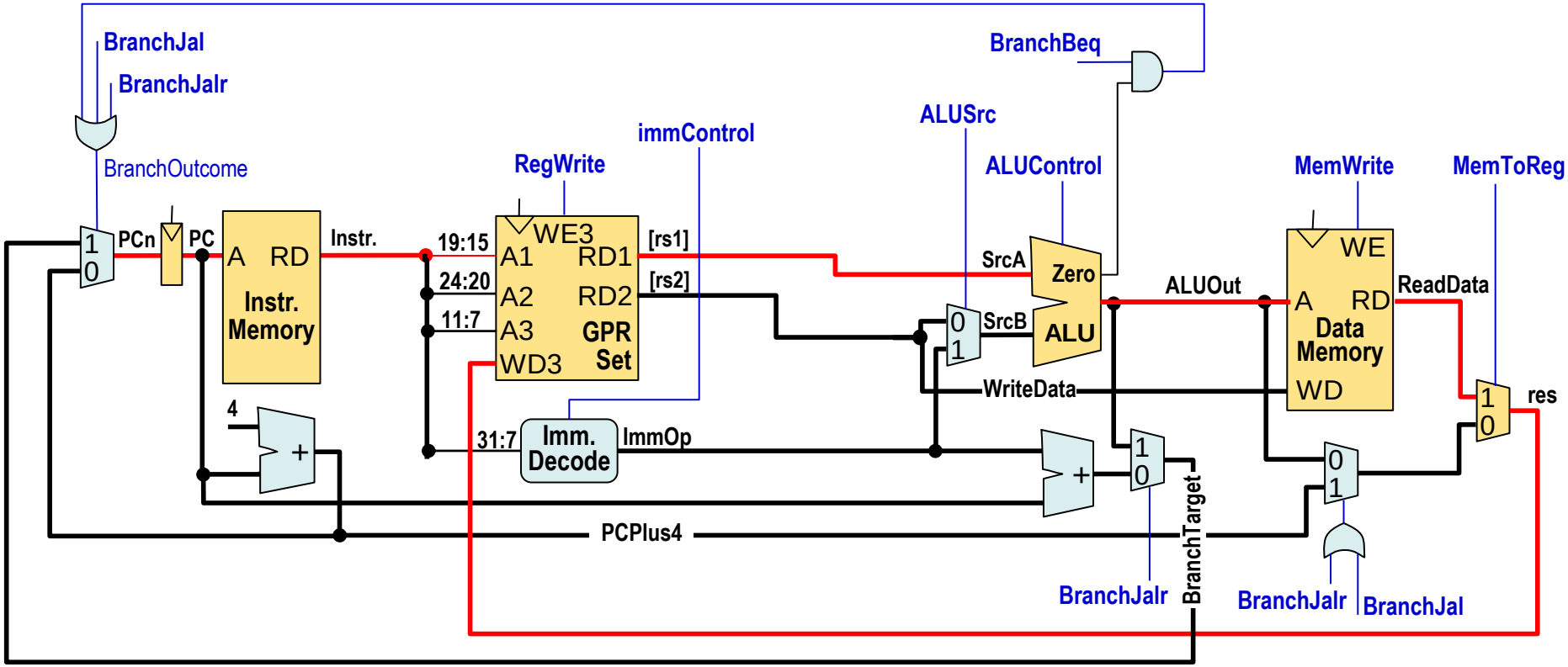
- What maximal possible clock frequency can we have?
- We need to determine what is the latency on the critical path.
- We need to analyze all instructions.



Single-cycle CPU – performance: $IPS = IC/T_{CLK} = IPC * f_{CLK}$

The critical (longest) path is for instruction `lw`.
Latency on the critical path is (red color in the scheme):

$$T_{CLK} = t_{PC} + t_{Mem} + t_{GPRread} + t_{Mux} + t_{ALU} + t_{Mem} + t_{Mux} + t_{GPRsetup}$$



Single-cycle CPU – performance: $IPS = IC/T_{CLK} = IPC * f_{CLK}$

$T_{CLK} = t_{PC} + t_{Mem} + t_{GPRread} + t_{Mux} + t_{ALU} + t_{Mem} + t_{Mux} + t_{GPRsetup}$.
Let's consider the following parameters:

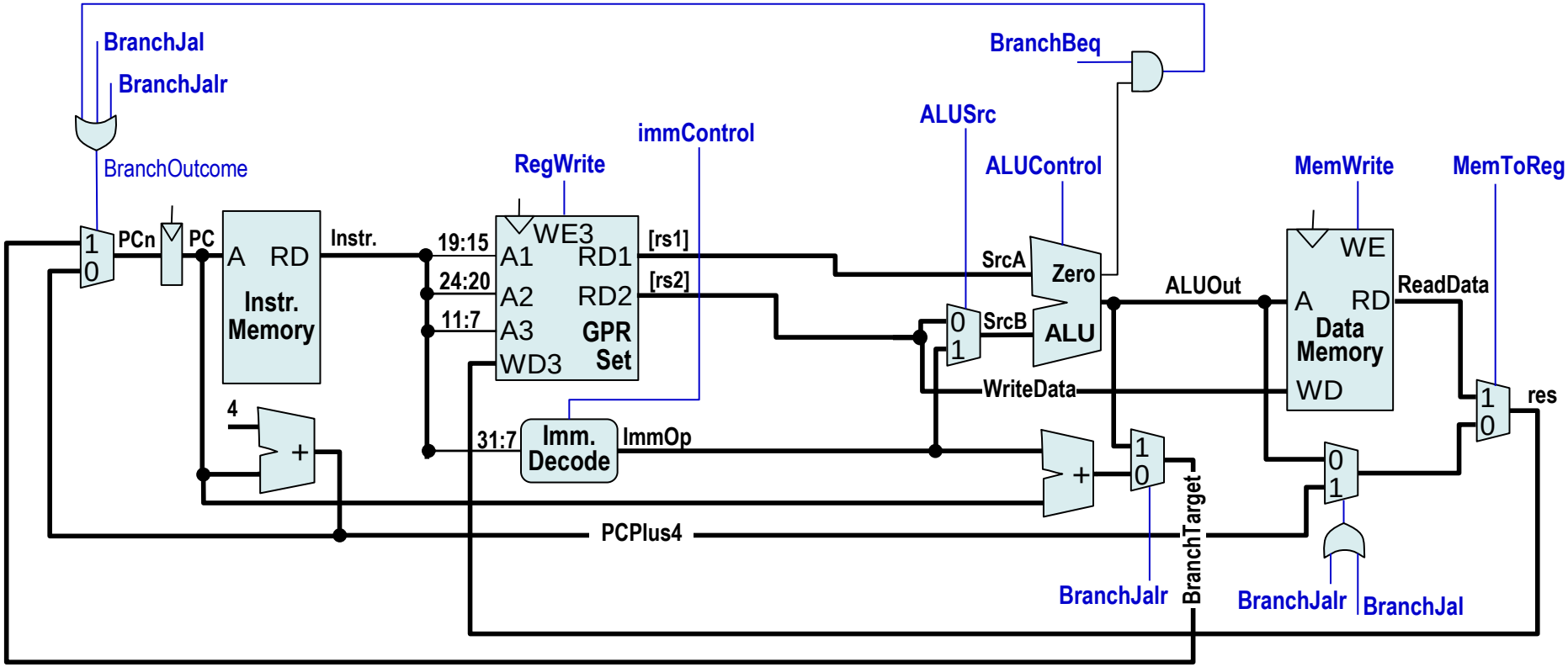
- $t_{PC} = 0.3$ ns
- $t_{Mem} = 20$ ns
- $t_{GPRread} = 1.5$ ns
- $t_{ALU} = 2$ ns
- $t_{Mux} = 0.1$ ns
- $t_{GPRsetup} = 0.1$ ns

Then $T_{CLK} = 44$ ns and therefore $f_{CLKmax} = 22.7$ MHz
and $IPS = IPC * f_{CLK} = 22\,700\,000$ IPS $= 22.7$ MIPS.

Note: $t_{GPRsetup}$ is the setup time needed before the rising edge of the clock. The factual write to a register overlaps with the beginning of the very next instruction. This overlapping instruction can read this newly written value without conflicts (after $t_{GPRread}$, the correct value is guaranteed).

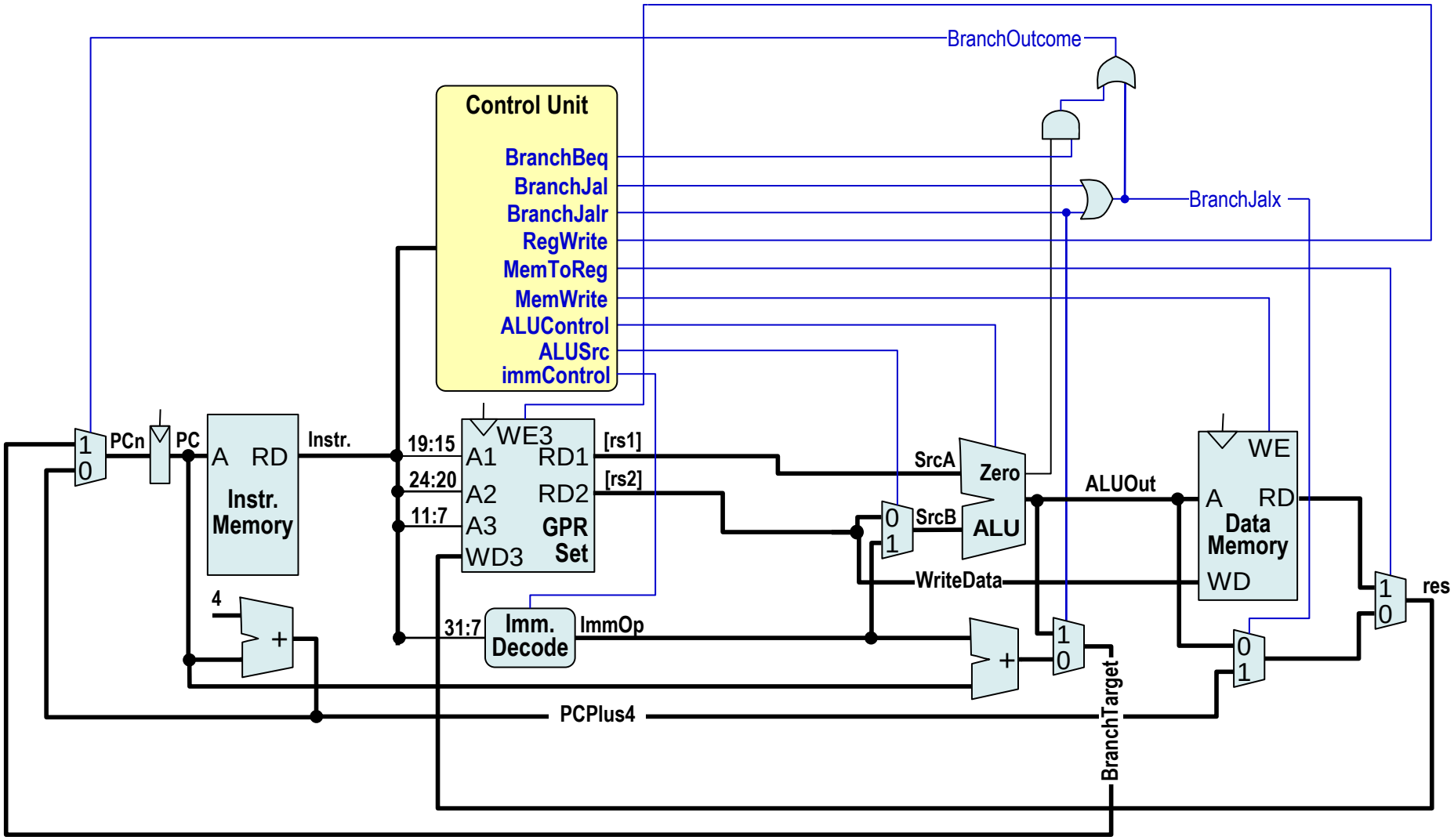
Recap

- We have designed the data paths for instruction processing.
- Now we have to build the controlling part of the processor integrating all the control signals.



Design of a Control Unit

- A control unit (CU) (yellow box) generates control signals required for processing the current instruction.



Encoding of picoRISC-V instructions I

opcode	Meaning	For our picoRISC-V
0110011	R-type (see funct7 and funct3)	`add`, `sub`, `slt`, `or`, `and`
0010011	ALU-imm (see funct3)	`addi`
0000011	Memory load (see funct3)	`lw`
0100011	Memory store (see funct3)	`sw`
1100011	Branch (see funct3)	`beq`
1101111	jal	`jal`
1100111	jalr	`jalr`

funtc7	**funtc3**	Instruction
0000000	000	`add`
0100000	000	`sub`
0000000	010	`slt`
0000000	110	`or`
0000000	111	`and`

Encoding of picoRISC-V instructions II

Slides 13, 20

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]		rs1	010	rd	0000011	`lw` (I)
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	`sw` (S)
0000000	rs2	rs1	000	rd	0110011	`add` (R)
0100000	rs2	rs1	000	rd	0110011	`sub` (R)
0000000	rs2	rs1	010	rd	0110011	`slt` (R)
0000000	rs2	rs1	110	rd	0110011	`or` (R)
0000000	rs2	rs1	111	rd	0110011	`and` (R)
imm[11:0]		rs1	000	rd	0010011	`addi` (I)
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	`beq` (B)
imm[20|10:1|11|19:12]				rd	1101111	`jal` (J)
imm[11:0]		rs1	000	rd	1100111	`jalr` (I)

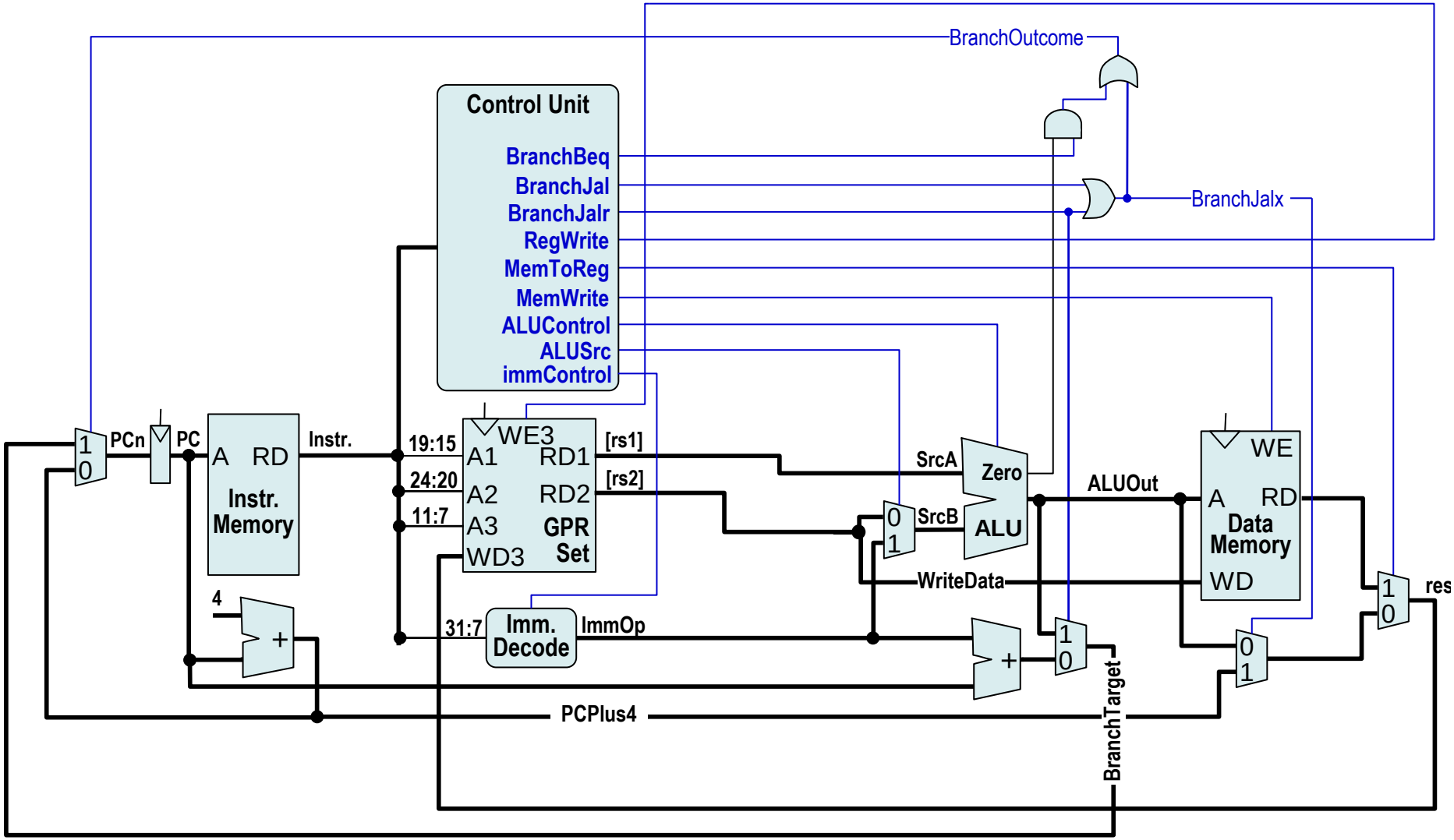
Single-cycle CPU – CU design

Control signal values are defined in the following table:

Instruction	Opcode	Funct3	Funct7	ALUSrc	ALUControl	MemWrite	MemToReg	RegWrite	BranchBeq	BranchJal	BranchJalr	ImmControl
`lw`	0000011	010	don't care	Control signal values are given on the previous slides. (This is left to students as an exercise.)								
`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									

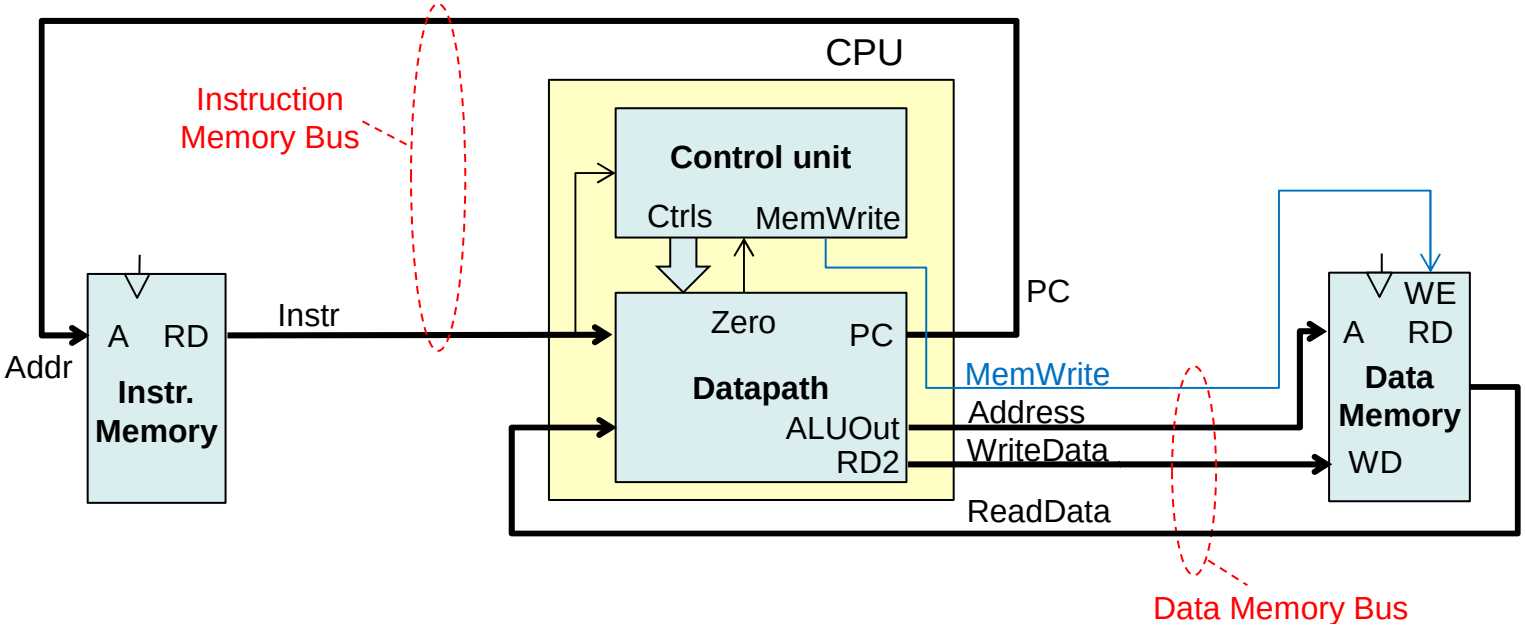
Therefore, the single-cycle picoRISC-V CU can be implemented as a **combinational circuit**.

We are done! Our picoRISC-V single-cycle microarchitecture



Single-cycle CPU – a generalization

- The main task of this lecture was a design of simple microarchitecture consisting of a CPU and separated instruction and data memory.
- In our design, the CPU has independent data buses for both memories.



- In general, a memory bus consists of address, data, and control wires.
- To make our design simple, there are two buses to data memory: 32-bit `WriteData` and 32-bit `ReadData`. Since in our design, concurrent memory load and store operations cannot be executed, it would be better to use full-duplex memory bus with transfer direction control – see BIE-SAP Lecture 10.

Generic task of a CU

- The task of a 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 a state that instructions (can be) are executed.
- Generally, the task of a CU is to **generate sequences of control signals for computer subsystems in such an order that prescribed operations (arithmetic, data exchange, instruction flow control, etc.) are executed**.

The design of a CU

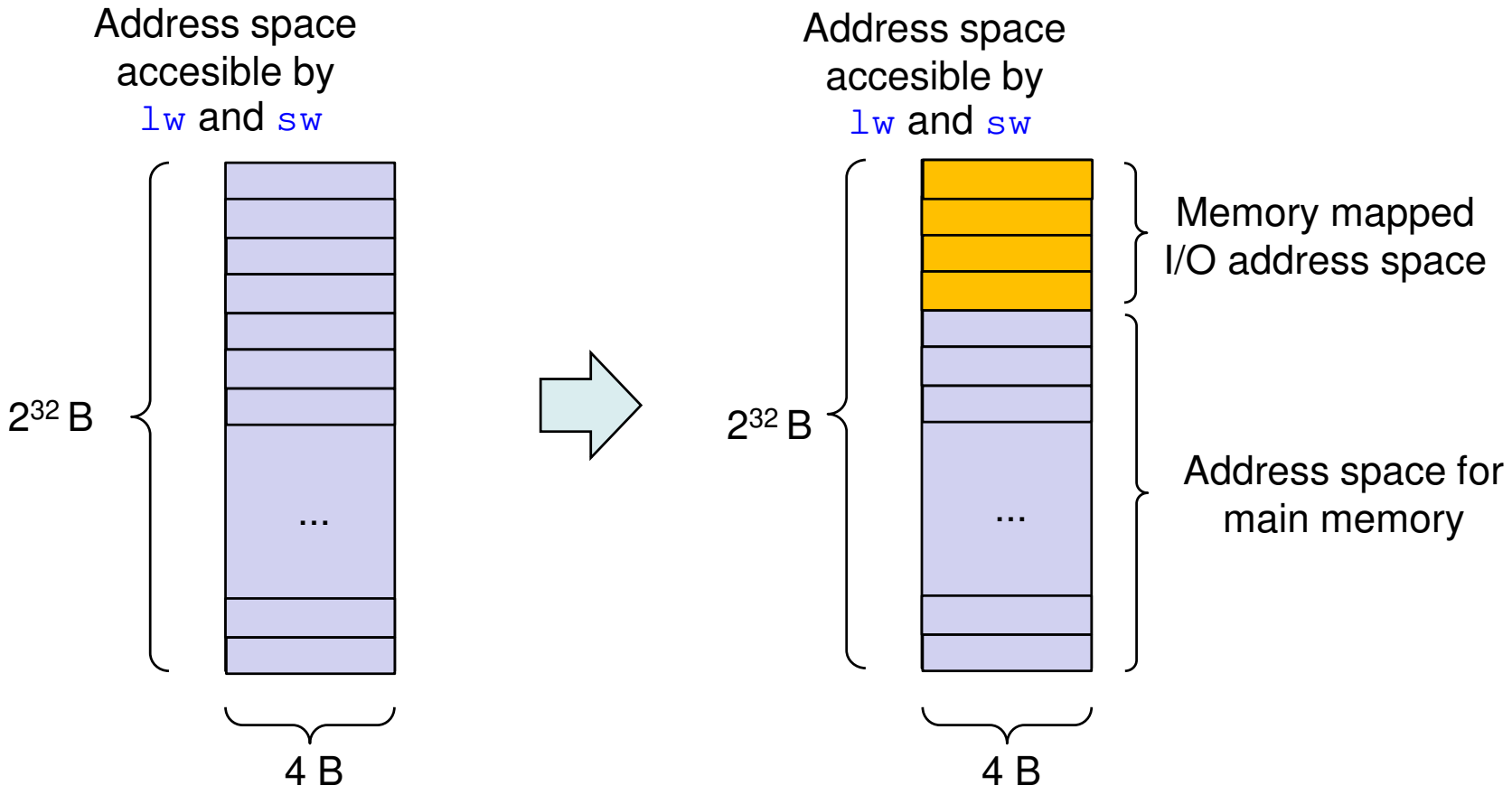
- A CU is typically a sequential circuit.
- A CU generates control signals at appropriate times:
  - Memory Select, Write Enable (WE), clock gating.
  - Data path switching (= multiplexer control).
  - Determination of ALU operations.
- It reacts to the status signals (CU inputs):
  - In our case, CU reacts only to the Zero ALU output signal.
  - In real CPUs, many more conditions can influence instruction cycle – interrupts, exceptions, etc.

Possible hardware implementations of CU

- **Hardwired CU**:
  - combinational logic (this was our case),
  - sequential logic, i.e., finite state machine (Mealy, Moore, ...).
- **Microprogrammed CU**:
  - A **microinstruction** is a group of elementary operations that control data flow and sequencing of instruction execution in a processor at the level of simplest operations, such as moving the content of a register to ALU, etc.
  - A sequence of microinstructions is called a **microprogram**.
  - Instructions in ISA (`add`, `sub`, `lw`, `jal`, ...) are implemented as microprograms. That is, one ISA instruction can be implemented by one or several microinstructions.
  - A microprogram is stored in a **control memory** of CU.
  - The opcode of an instruction determines the address of the instruction's microprogram.
  - Advantages: **flexibility** (new microprogram = "new" microprocessor).
  - Disadvantages: **complex** and **not convenient for pipelined processors**, since individual pipeline stages process different instructions and hazards arise.

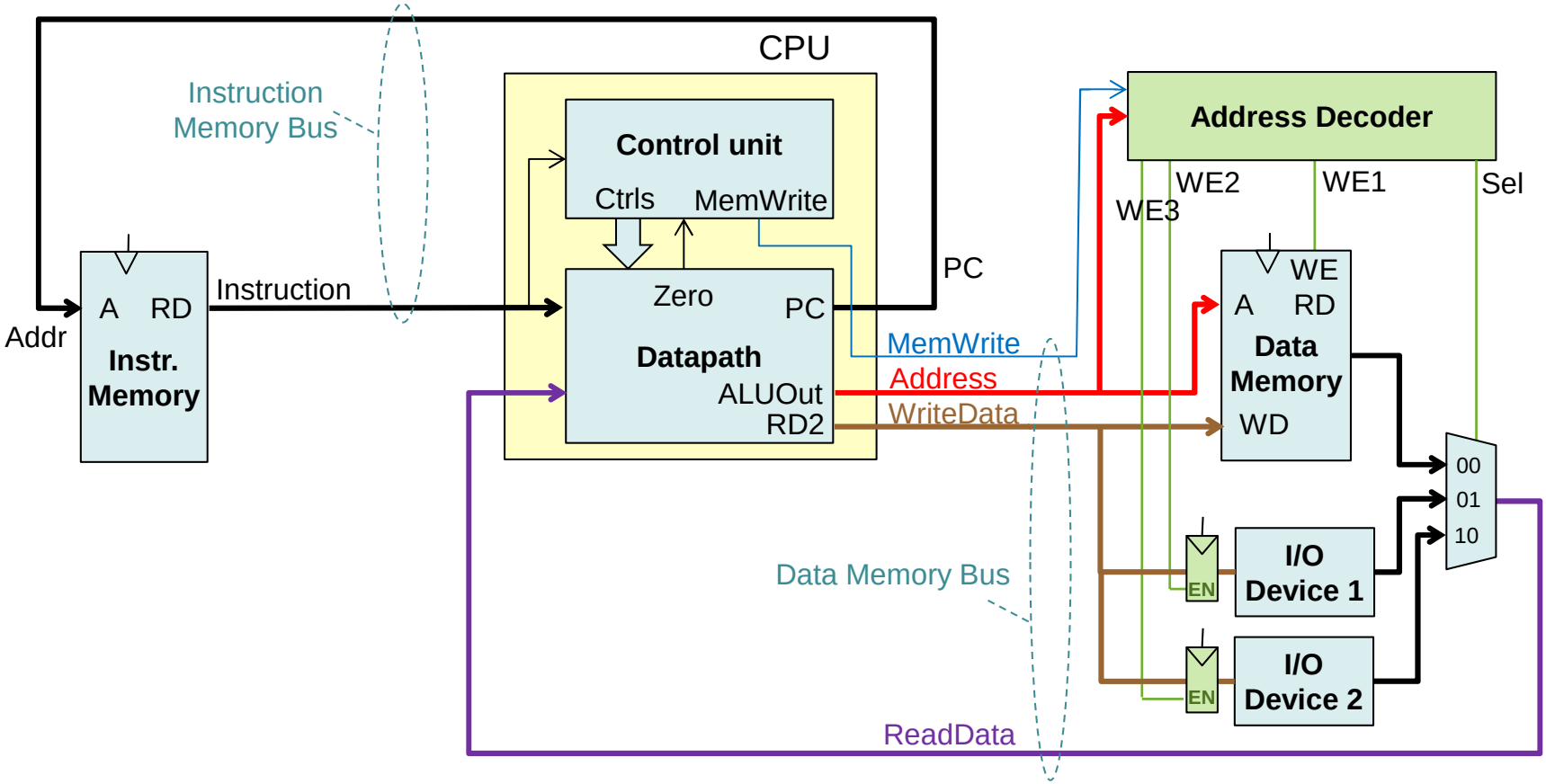
Communication with input/output (I/O) devices

The idea: To communicate with I/O peripheral devices (keyboards, monitors, printers), we can use the same interface as with main memory (instructions `lw` and `sw`). This is called **memory mapped I/O**.



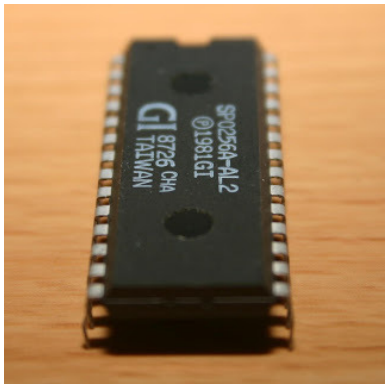
Implementation of memory mapped I/O

The **Address Decoder** monitors the address on the Data Memory Bus and signal `MemWrite`. If it detects a match with some I/O address, it takes the appropriate action.

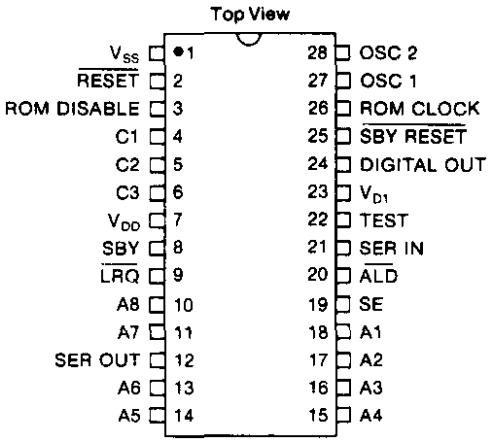


Speech Synthesizer Example

- In English, there are about 60 elementary spoken sounds (called **allophones**) from which all individual words of spoken English consist.
- The SP0256 chip is able to generate all these allophones, it contains addressable encoded allophones.
- **The task**: Write a SP0256 **driver** in our picoRISC-V ISA that is able to read an array of 6-bit codes (= SP0256 addresses) of 5 allophones starting from main memory address `0x00000100` and send these codes sequentially to a SP0256 chip so that the chip will consecutively generate these sounds using a loudspeaker attached to its loudspeaker `DigitalOut` pins.

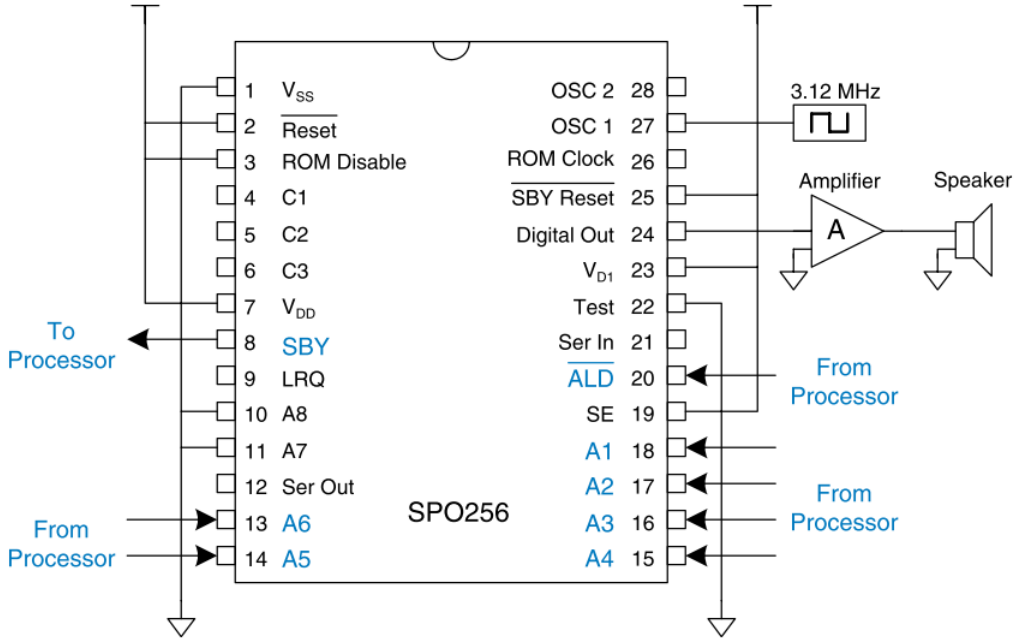


http://little-scale.blogspot.com/2009/02/sp0256-al2-creative-commons-sample-pack.html



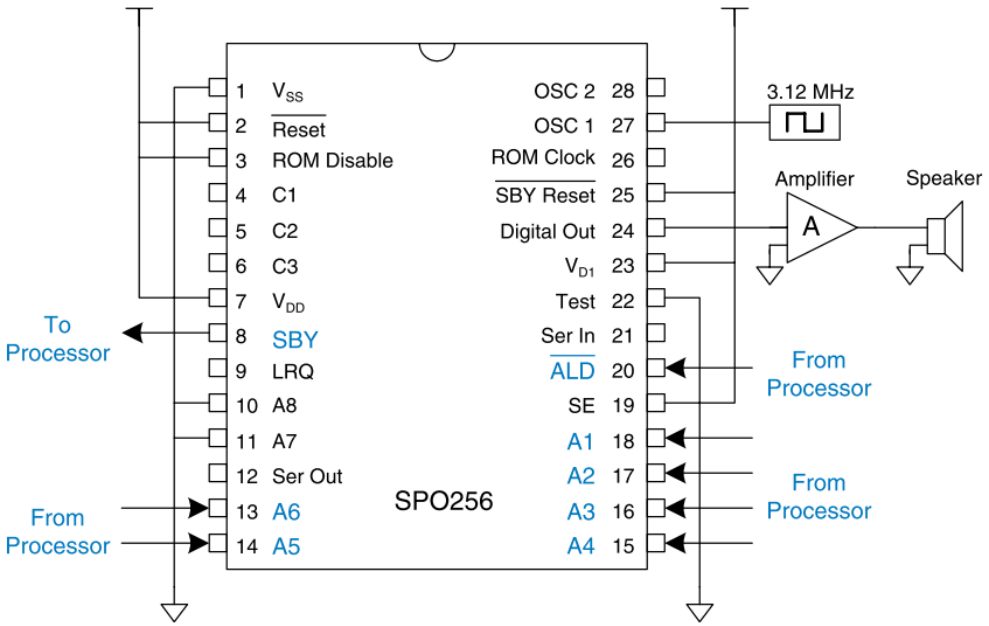
Speech Synthesizer Example: SP0256

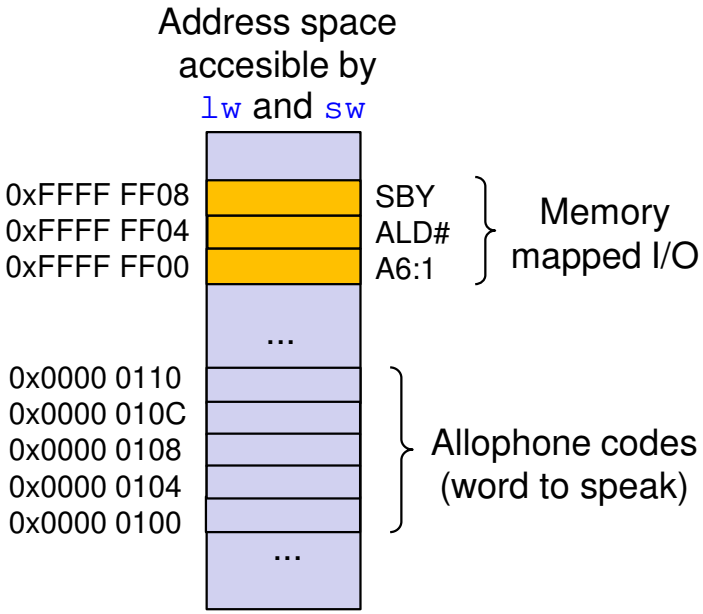
- Pins `A6:1` receive a 6-bit allophone code from the CPU.
- The allophone sound digital signal is sent to the `DigitalOut` pin.
- `SBY` is an output status pin:
  - If `SBY` = 1, SP0256 is standing by and is ready to receive a new allophone code. Otherwise no input is accepted.
- Pin Address Load $\overline{\texttt{ALD}}$ is an input control pin:
  - On the falling edge of $\overline{\texttt{ALD}}$, SP0256 reads a new allophone code supplied to `A6:1`.



Speech Synthesizer Example: Memory Mapping of the Chip

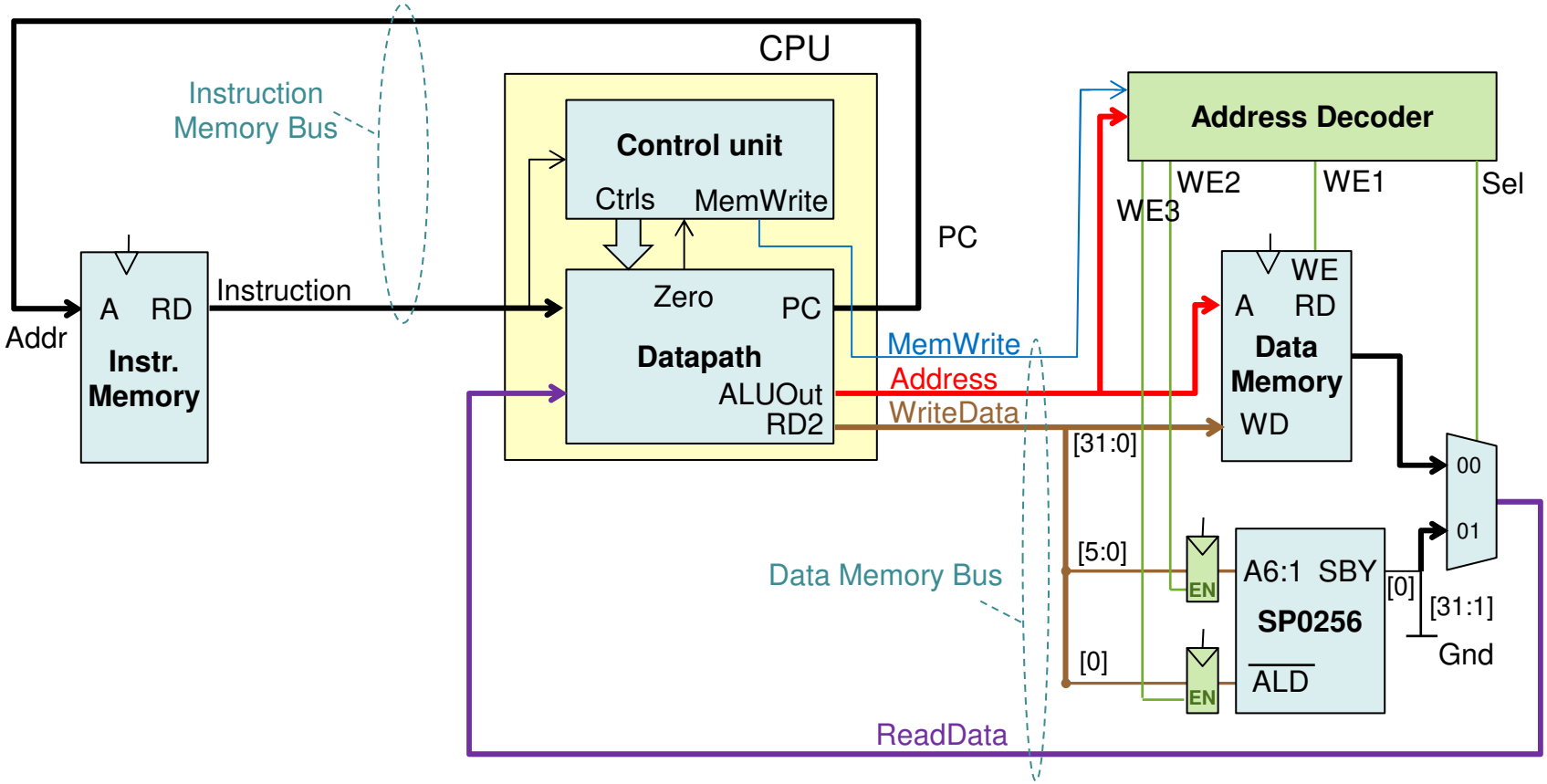
- Assume the following memory mapping:
  - Port `A6:1` to the address `0xFFFF FF00`,
  - $\overline{\texttt{ALD}}$ to the address `0xFFFF FF04`,
  - `SBY` to the address `0xFFFF FF08`.





Speech Synthesizer Example: picoRISC-V + SP0256

- Lower 6 bits of `WriteData` bus are connected to pins `A6:1`.
- The lsb of `WriteData` bus is connected to pin $\overline{\texttt{ALD}}$.
- Similarly, pin `SBY` is connected to the `ReadData` lsb.



Speech Synthesizer Example: Driver

1. Set $\overline{\text{ALD}}$ to 1.
2. Wait until SP0256 sets SBY to 1 (indication of its readiness).
3. Write a 6-bit allophone code to pins A6:1.
4. Set $\overline{\text{ALD}}$ to 0 (command to SP0256 to start to generate the sound).

```
init:
 addi x1,x0,1        // [x1] = 1 (value to write to ALD#)
 addi x2,x0,20       // [x2] = array size *4 (20 bytes)
 addi x3,x0,0x100    // [x3] = array base address
 addi x4,x0,0        // [x4] = 0 (array index)
start:
 sw x1,0xF04(x0)     // ALD#=1
loop:
 lw x5,0xF08(x0)     // [x5] = SBY (monitor the state)
 beq x5,x0,loop      // loop while SBY == 0
 add x6,x3,x4        // [x6] = address of an allophone
 lw x7,0(x6)         // [x7] = allophone
 sw x7,0xF00(x0)     // [A6:1] = allophone
 sw x0,0xF04(x0)     // [ALD#] = 0 (to initiate speech)
 addi x4,x4,4        // increment array index
 beq x4,x2,done      // are all allophones done?
 beq x0,x0,start     // if not, repeat
done:                // otherwise stop
```

- Note that the CPU checks whether SP0256 is ready by periodic checking of its SBY output. This is called **polling** or **busy waiting**. It would be better to link SBY to a CPU **interrupt** system to enable some useful computation in the meantime.

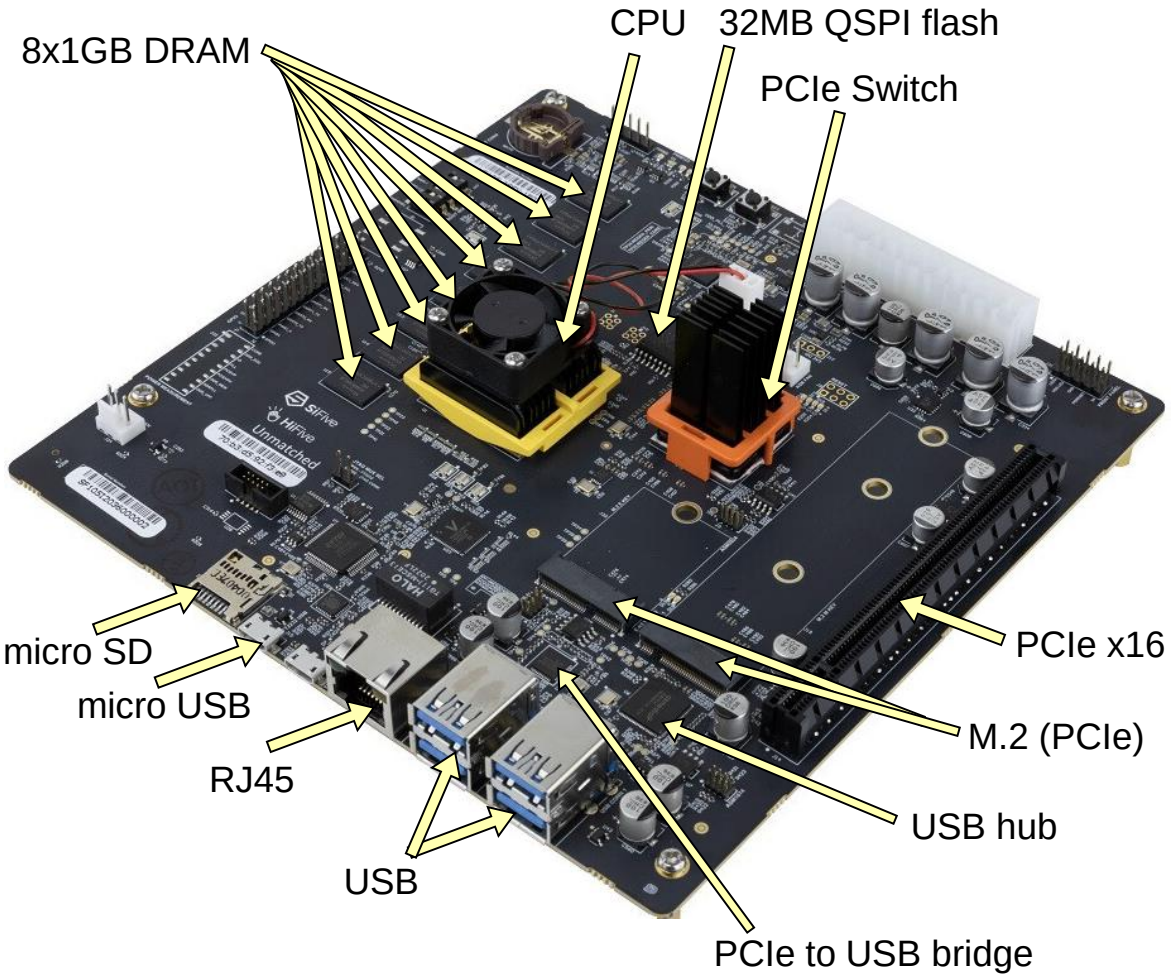
Communication with I/O in general

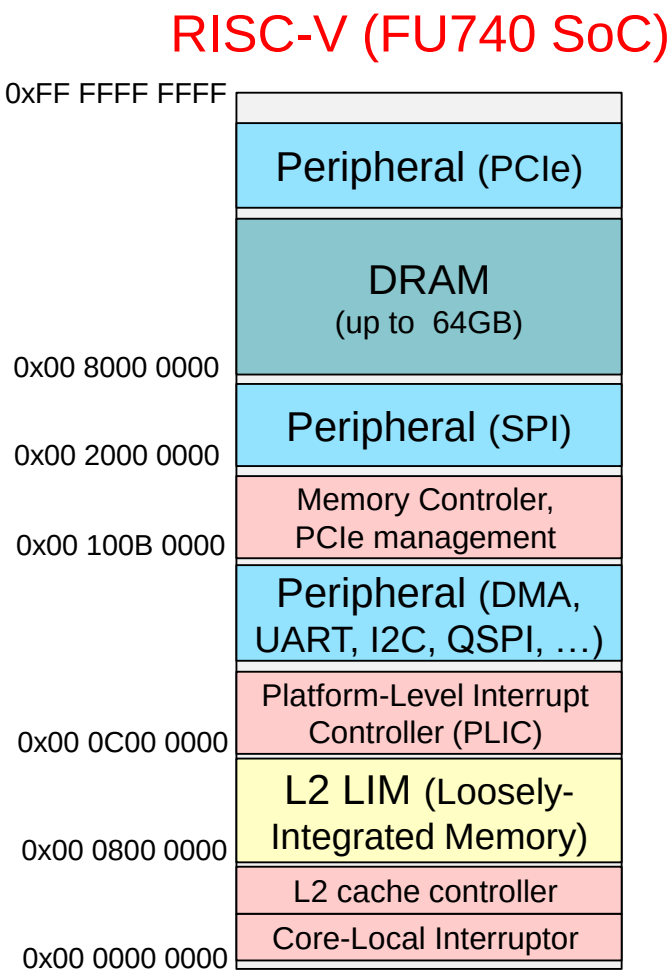
Basically, there are two approaches to CPU-I/O communication:

- **Memory-mapped I/O**:
  - A part of main memory address space is dedicated to the I/O devices. R/W from/to these addresses are interpreted as commands or data transfers from/to these devices. The memory system ignores these operations since it knows the I/O address range. The I/O driver however detects these ops and reacts accordingly.
  - In our case study, a **centralized approach** was used when an address decoder controls to the main memory and all I/O devices.
  - In practice, an **autonomous approach** is common where each I/O device has its own special registers with device addresses, initialized during the boot process. Each I/O snoops its own addresses.
- **Port-mapped I/O**:
  - A dedicated separated I/O address space is used.
  - **Special I/O instructions** are needed (e.g., `in` and `out` in x86) to manage this space.

Partitioning of physical memory address space I

An example of partitioning of physical memory address space for SoC (System on Chip) FU740 containing a 64-bit 5-core RISC-V CPU [3]:

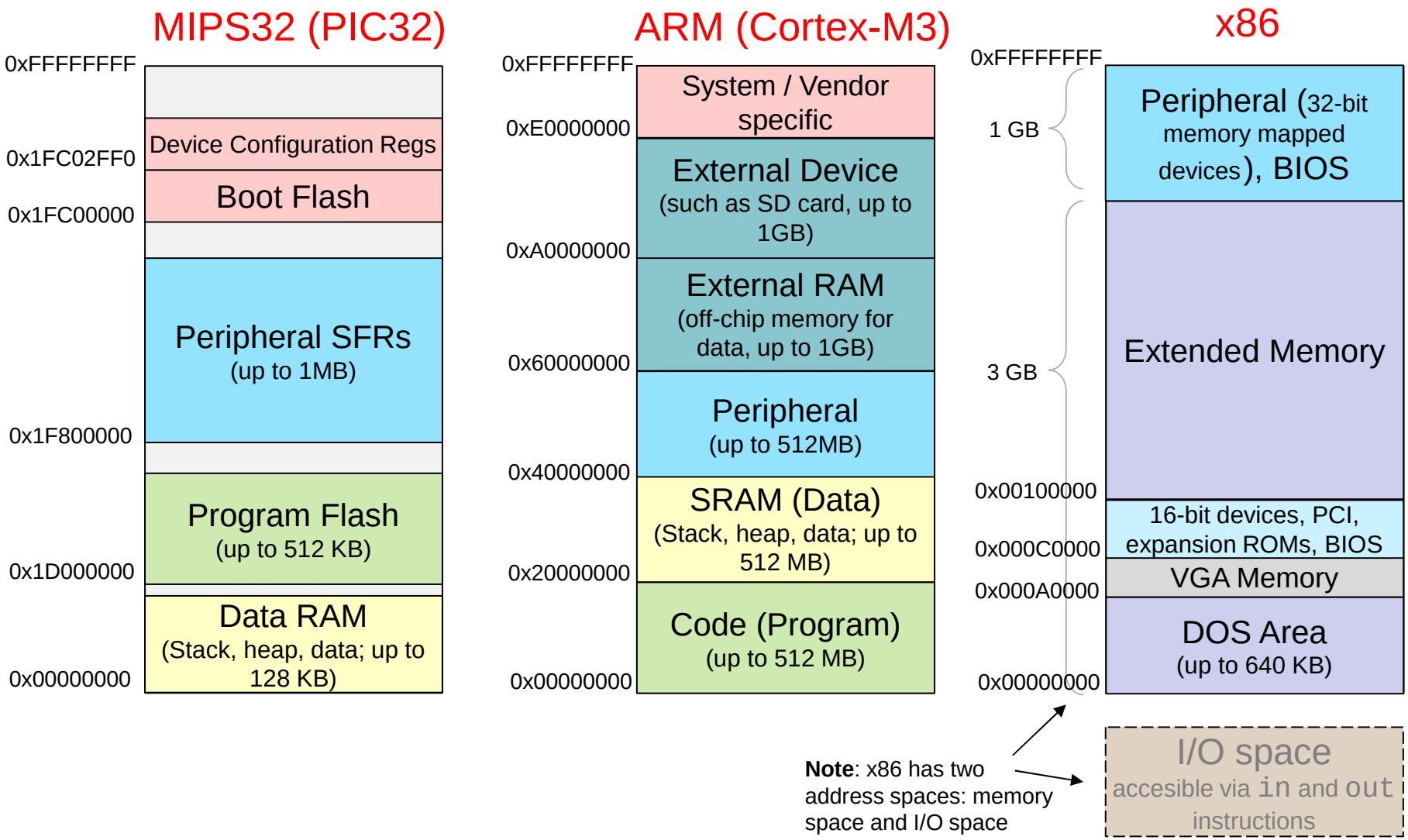




[3] 1xRV64IMAC, 4xRV64IMAFDC

Partitioning of physical memory address space II

Partitioning of physical memory address space differs among architectures and vendors:



References

(1) J. L. Hennessy, D. A. Patterson: Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 5th Edition. 2011.

(2) D. Paterson, J. Hennessy: Computer Organization and Design, The HW/SW Interface. Elsevier, ISBN: 978-0-12-370606-5

(3) The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213, RISC-V Foundation, December 2019.

(4) D. Harris, S. Harris: Digital Design and Computer Architecture, 2nd Edition. Morgan Kaufmann.

(5) J. Franco: What is Microprogramming and Why Should We Know About it? http://gauss.ececs.uc.edu/Courses/c4029/exams/Spring2013/Review/microcod

(6) Cortex-M3 Technical Reference Manual: https://developer.arm.com/documentation/ddi0337/e/memory-map/about-the-memory-map

(7) PIC32 Family Reference Manual: Section 3. Memory Organization. http://ww1.microchip.com/downloads/en/devicedoc/60001115h.pdf

(8) SiFive U74 Core Complex Manual, 21G2.01.00.

(9) SiFive FU740-C000 Manual, v1p2.