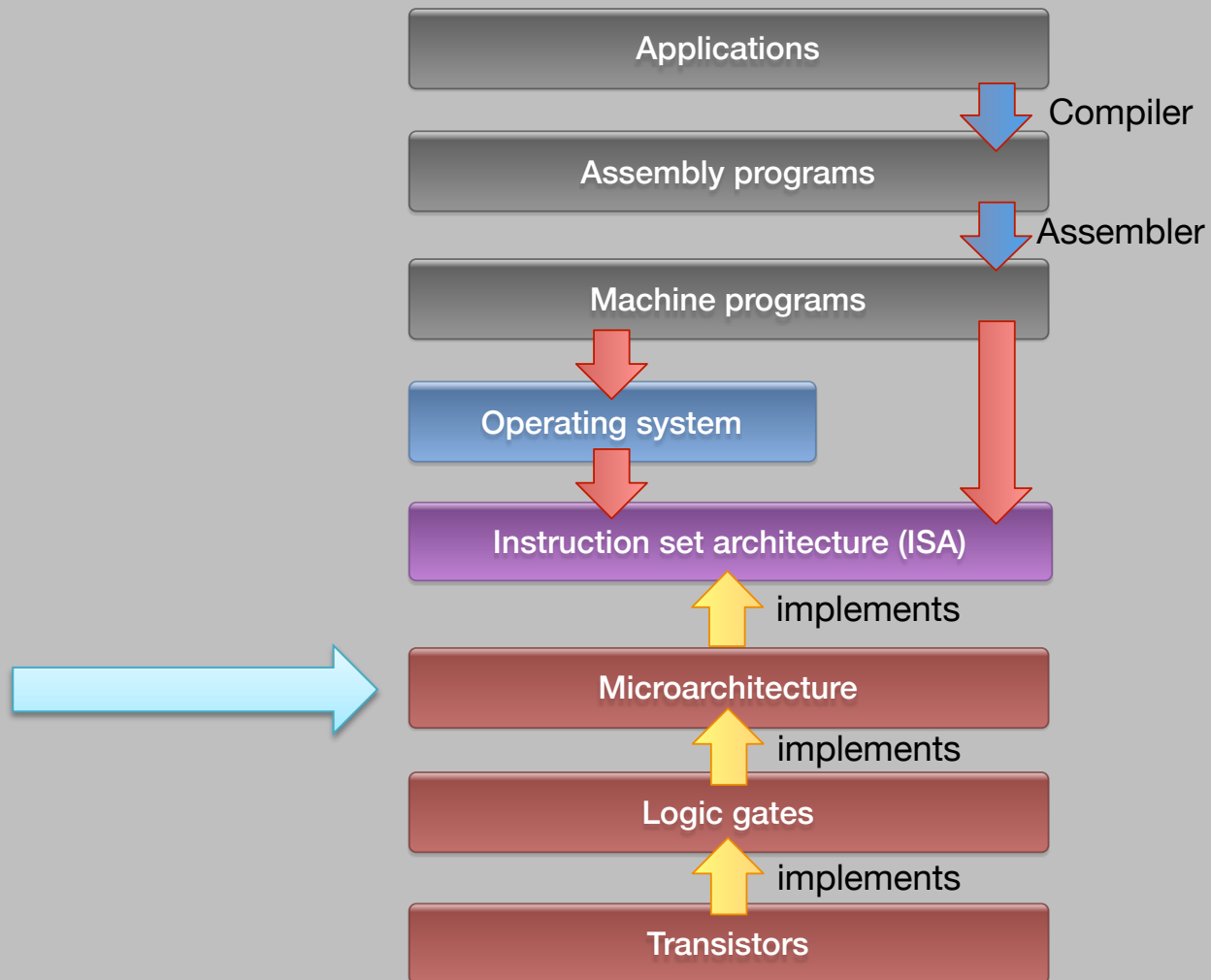


Microarchitecture at the Example of MIPS

Becker/Molitor, Chapter 10 treats a similar but not identical system.
We follow the American book "Digital design and computer architecture"
by Harris and Harris, 2013.
Here, Chapters 6 and 7 are particularly relevant.

Jan Reineke
Universität des Saarlandes

Abstraction layers in computer systems



Overview: Architecture vs Microarchitecture

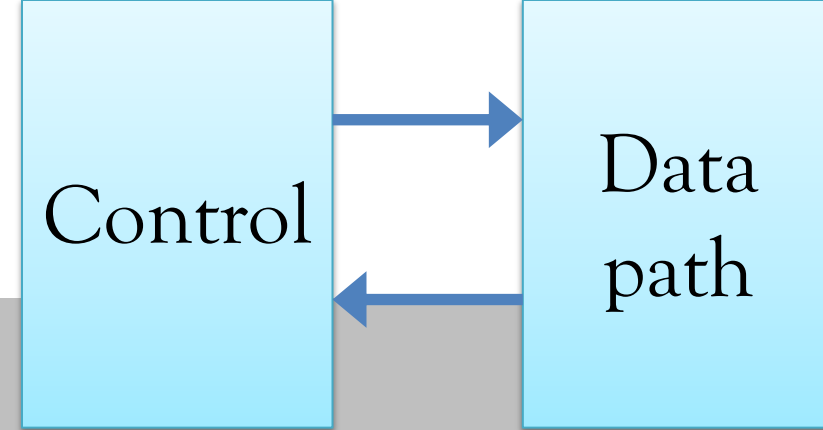
Microarchitecture

= concrete implementation of an instruction set in hardware

= „**How**“ a computer works

For example: Intel Skylake, AMD Zen 3 (both x86), Apple M1 (ARM)

High-level Structure



Datapath:

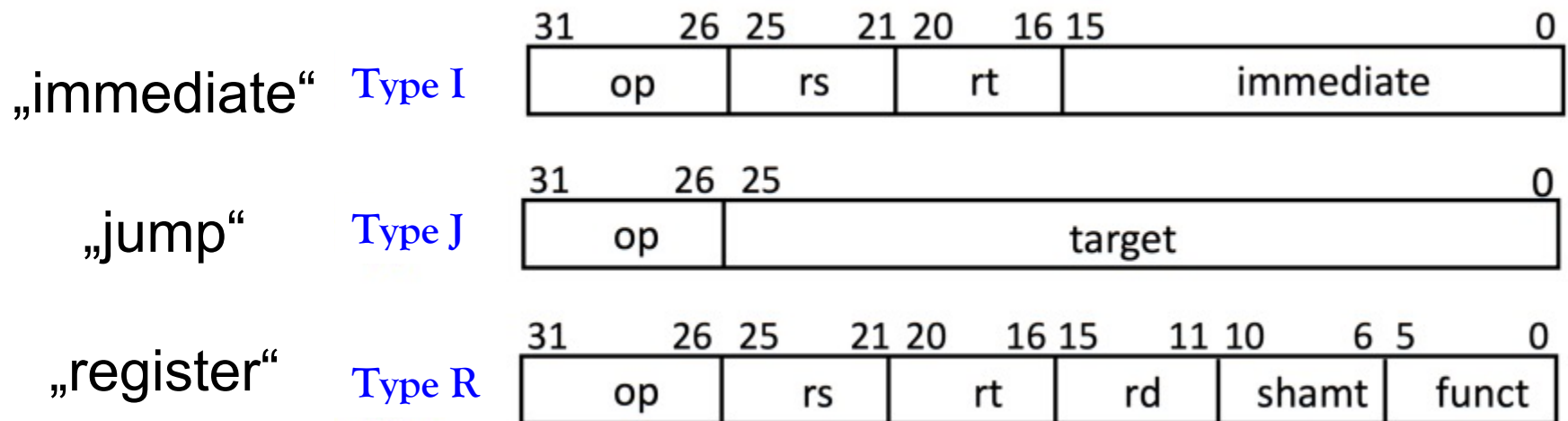
- Processing and transport of instructions and data
- Must support all operations and data transports
- Many options of differing complexity and speed are possible

Control:

- Processing and transport of control signals
- Many options:
 - Purely combinatorial control (here)
 - State-based control (e.g. with Pipelining, more about this later)
 - Microprogrammed (in current Intel and AMD processors)

Instruction encoding

- **Instruction encoding** refers to the encoding of instructions in machine words.
- MIPS uses a fixed-length 32-bit encoding of all instructions (this is unlike, e.g. x86)
- We distinguish three types I, J and R:



Instruction encoding

Abbreviation	Meaning
I	immediate
J	jump
R	register
op	6-bit encoding of the operation
rs	5-bit encoding of a source register
rt	5-bit encoding of a source or target register
immediate	16-bit immediate value
target	26-bit jump target
rd	5-bit encoding of the target register
shamt	5-bit encoding of “shift amount”
funct	6-bit encoding of the function

The MIPS subset

- R-type instructions:

add	add rd, rs, rt
subtract	sub rd, rs, rt
AND	and rd, rs, rt
OR	or rd, rs, rt
set less than	slt rd, rs, rt

op	code				Function	
----	------	--	--	--	----------	--

000000	rs	rt	rd	00000	100000
000000	rs	rt	rd	00000	100010
000000	rs	rt	rd	00000	100100
000000	rs	rt	rd	00000	100101
000000	rs	rt	rd	00000	101010

- Memory instructions:

load word	lw rt, imm(rs)
store word	sw rt, imm(rs)

100011	rs	rt	imm
101011	rs	rt	imm

- Jump and branch instructions:

branch on equal	beq rs, rt, imm
jump	j target

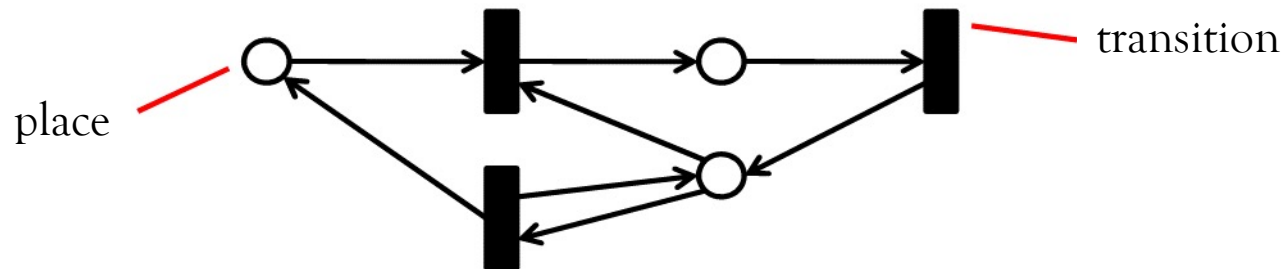
000100	rs	rt	imm
000010	target		

Petri nets

We are describing the refinement of the instructions via **Petri nets**.
Petri nets are a standard notation to represent parallel and distributed processes.

Static representation:

- Petri nets consist of **places** and **transitions**, which are connected via edges.
- Transitions can be associated with **conditions** and **operations**.
- A Petri net graph must be bipartite, i.e., no two places may be directly connected. The same holds for transitions.

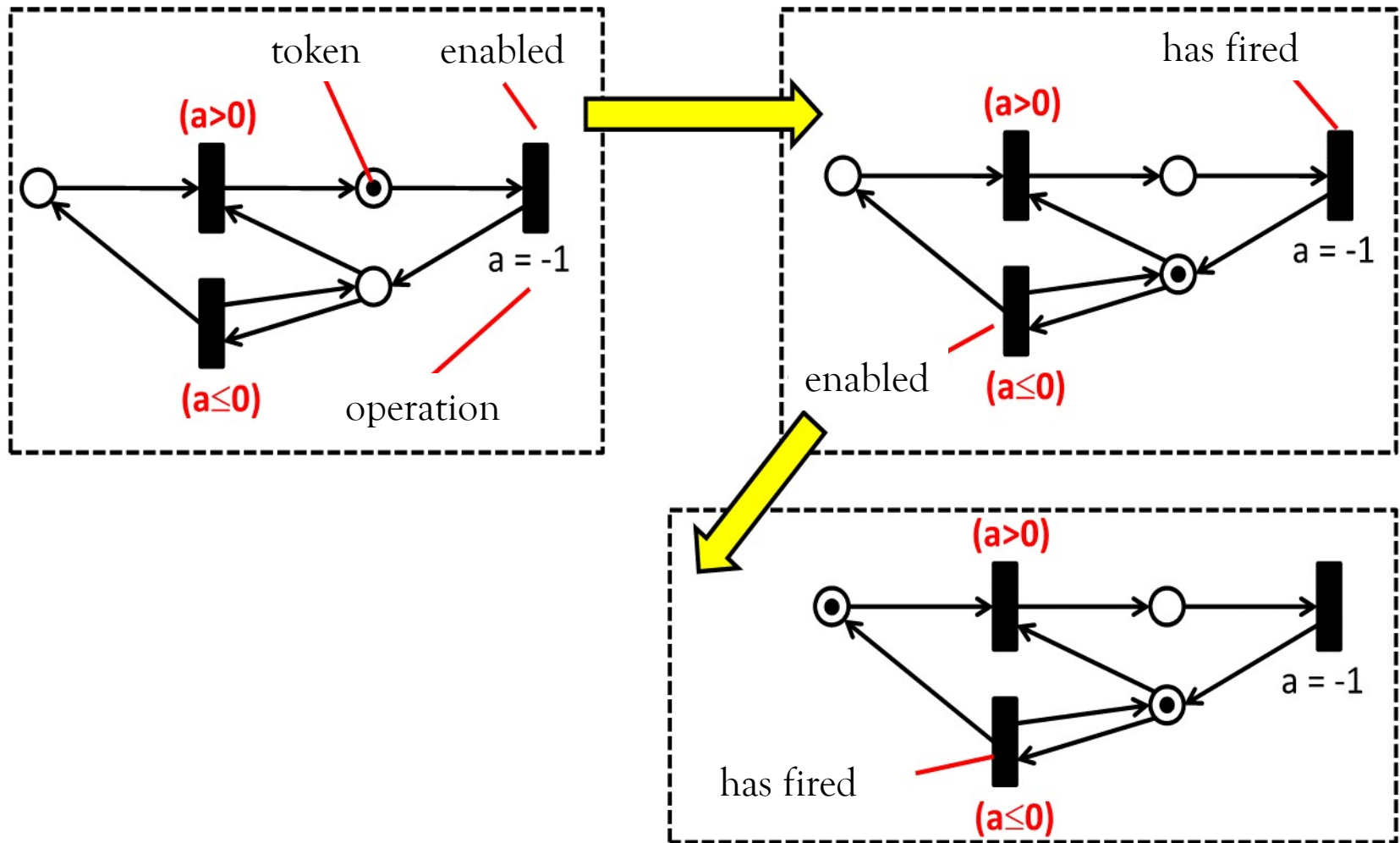


Petri nets

Dynamic representation:

- Places are associated with **tokens**.
- **Tokens** are “transported” across transitions according to the following rules:
 - A transition is “**enabled**” if
 - (a) the condition associated with the transition is satisfied, and
 - (b) every input place of the transition contains at least one token.
 - An enabled transition may “**fire**”.
Then, one token is removed from each input place,
and one token is added to each output place.
And the operation associated with the transition is executed.

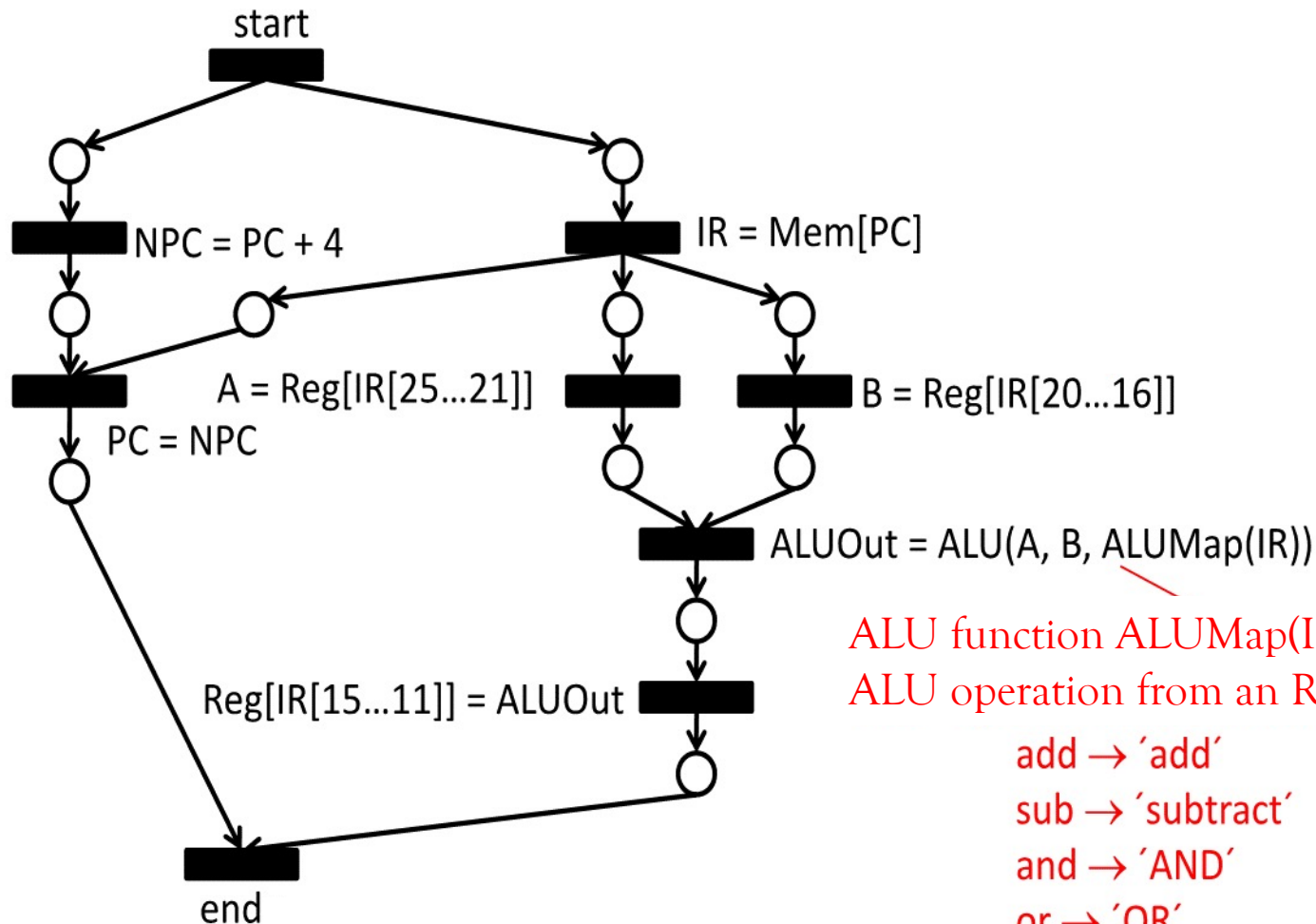
Petri nets: Example



Modeling the MIPS instructions

- **Temporary variables** to store 32-bit data and instructions:
A, B, ALUOut, Target,
PC (program counter), NPC (next program counter), IR (instruction)
- **Internal registers** of the processor: $\text{Reg}[i], 0 \leq i < 32$
- **Memory** at the address i: $\text{Mem}[i]$
- **Shifting** a word by 2 bit to the left:
 $\text{'01110'} \ll 2 = \text{'0111000'}$
- **Concatenation** of bits: $\text{'001'} \llcorner \text{'110'} = \text{'001110'}$
- **Extension** of a halfword into a word with/without sign extension:
 $\text{SignExt}(), \text{ZeroExt}()$
- **Arithmetic operation** where op is from {'add', 'subtract', 'AND', 'OR', 'setOnLessThan'}: $\text{ALU}(a, b, \text{op})$

Example: R-type instructions



ALU function **ALUMap(IR)** determines the ALU operation from an R-type instruction:

add → 'add'

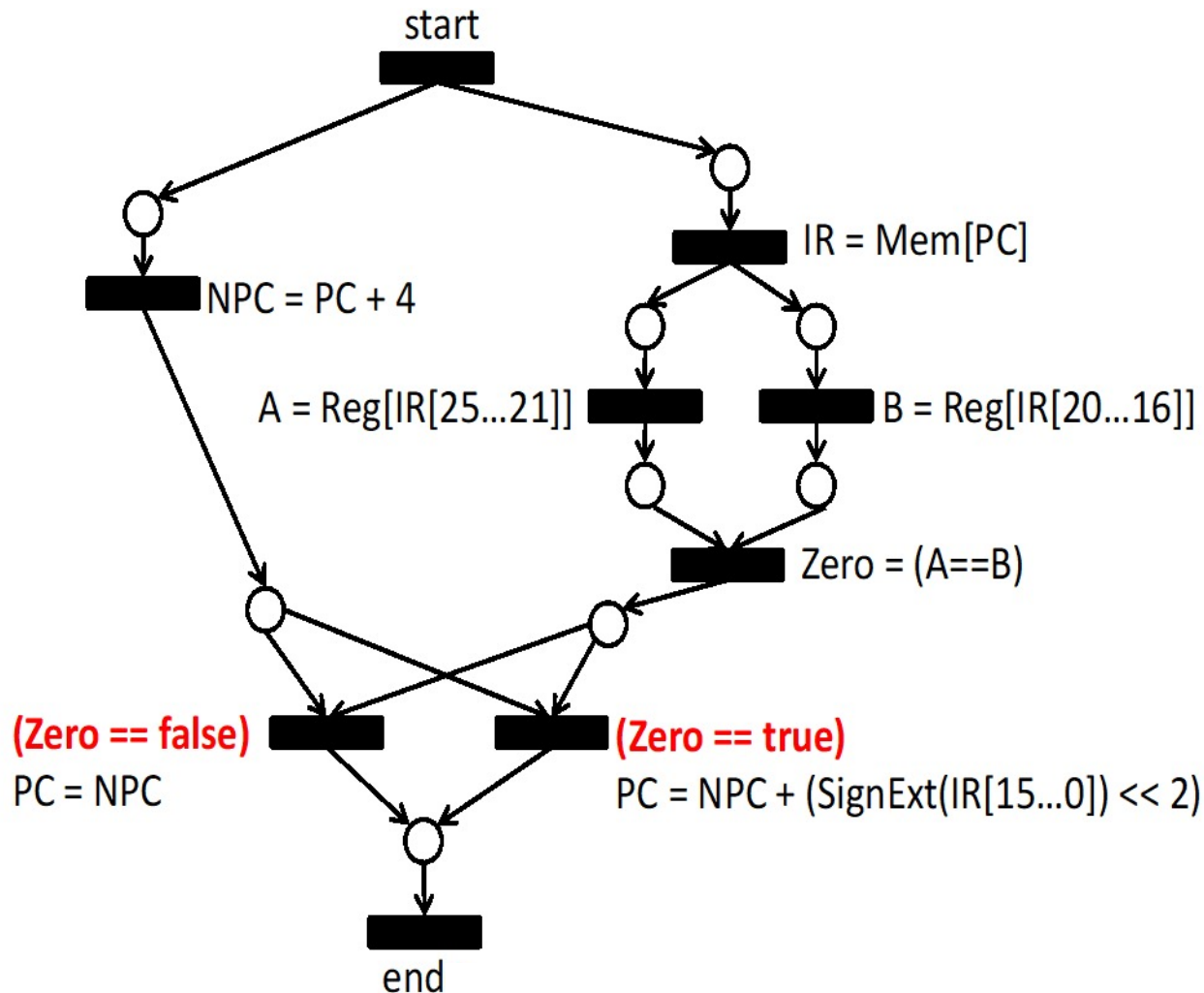
sub → 'subtract'

and → 'AND'

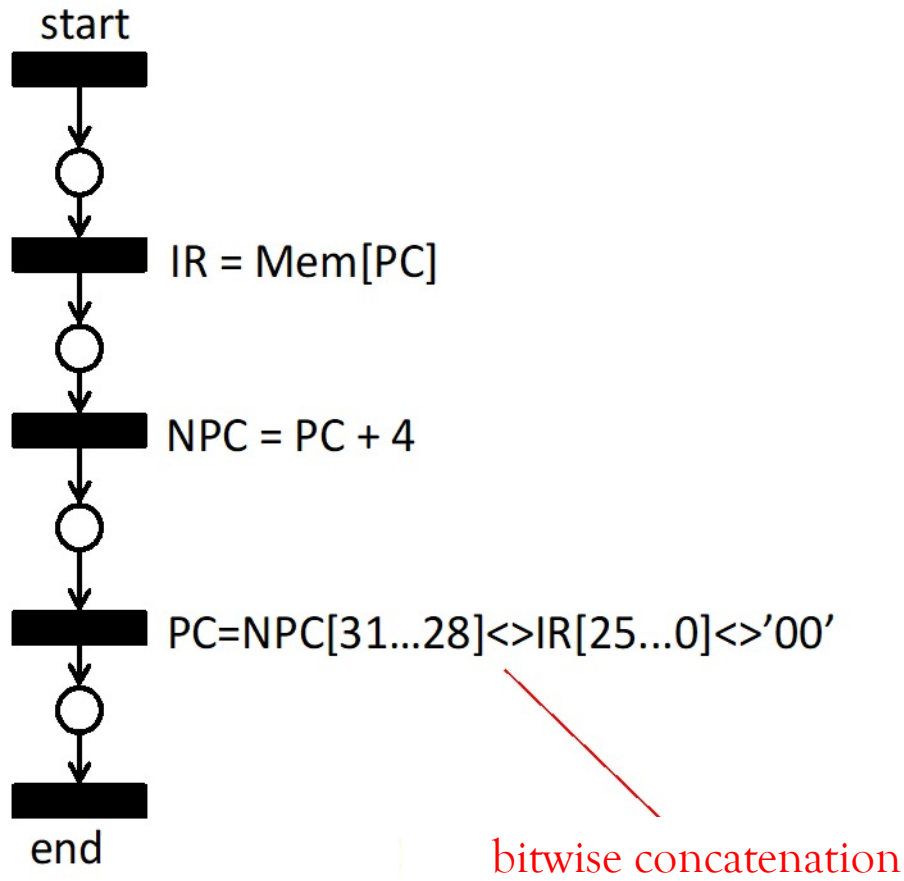
or → 'OR'

slt → 'setOnLessThan'

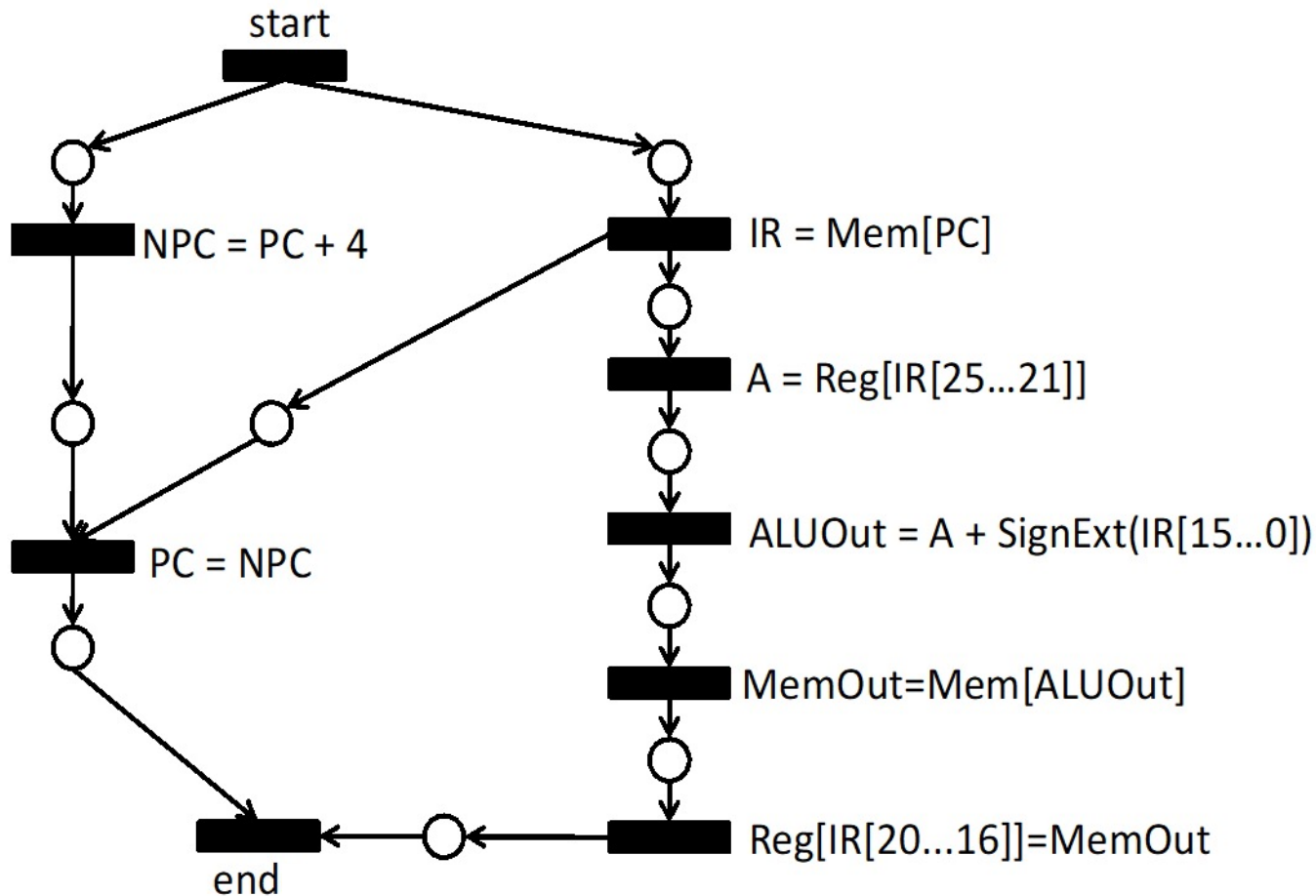
Example: beq instruction



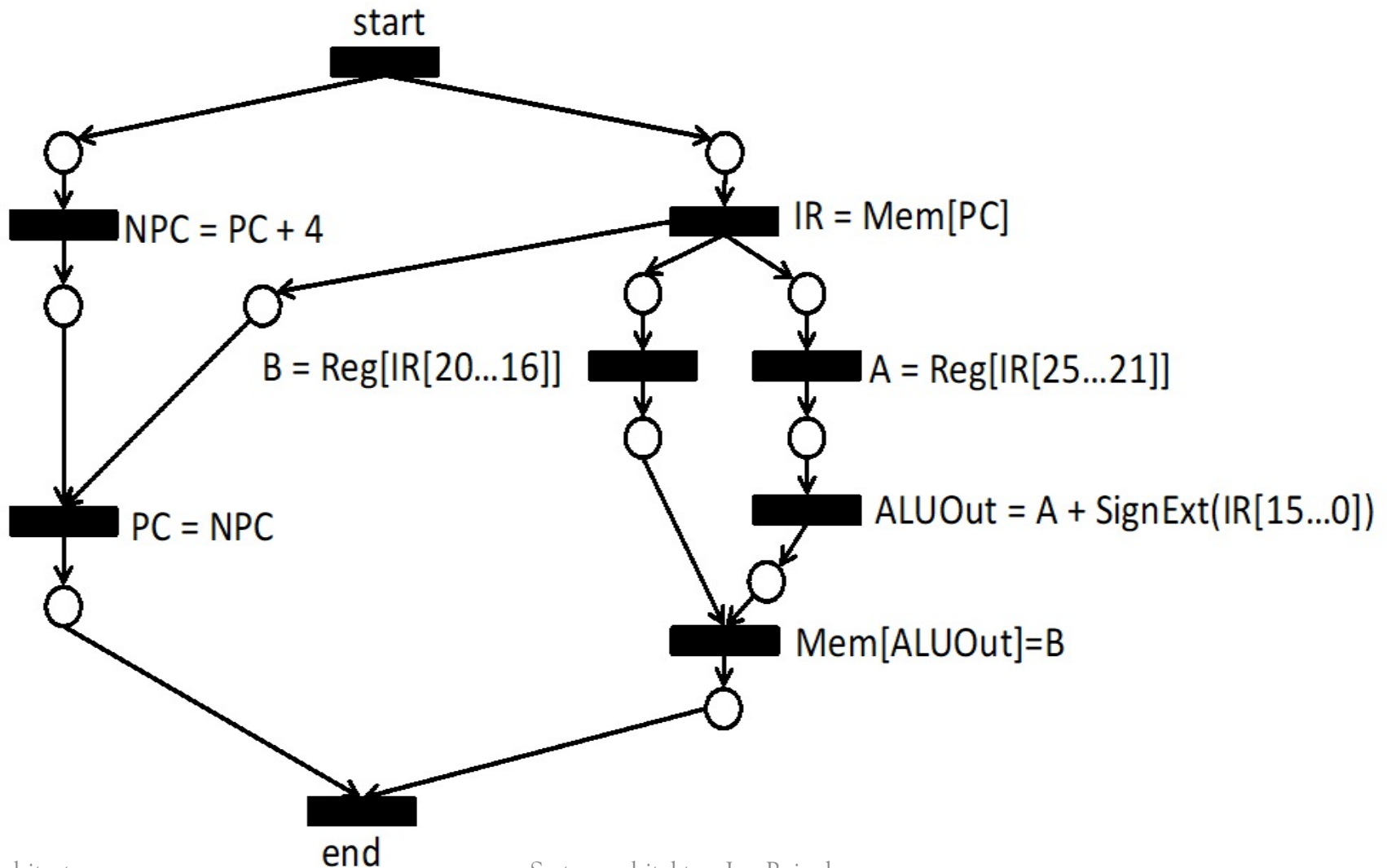
Example: j instruction



Example: lw instruction



Example: sw instruction

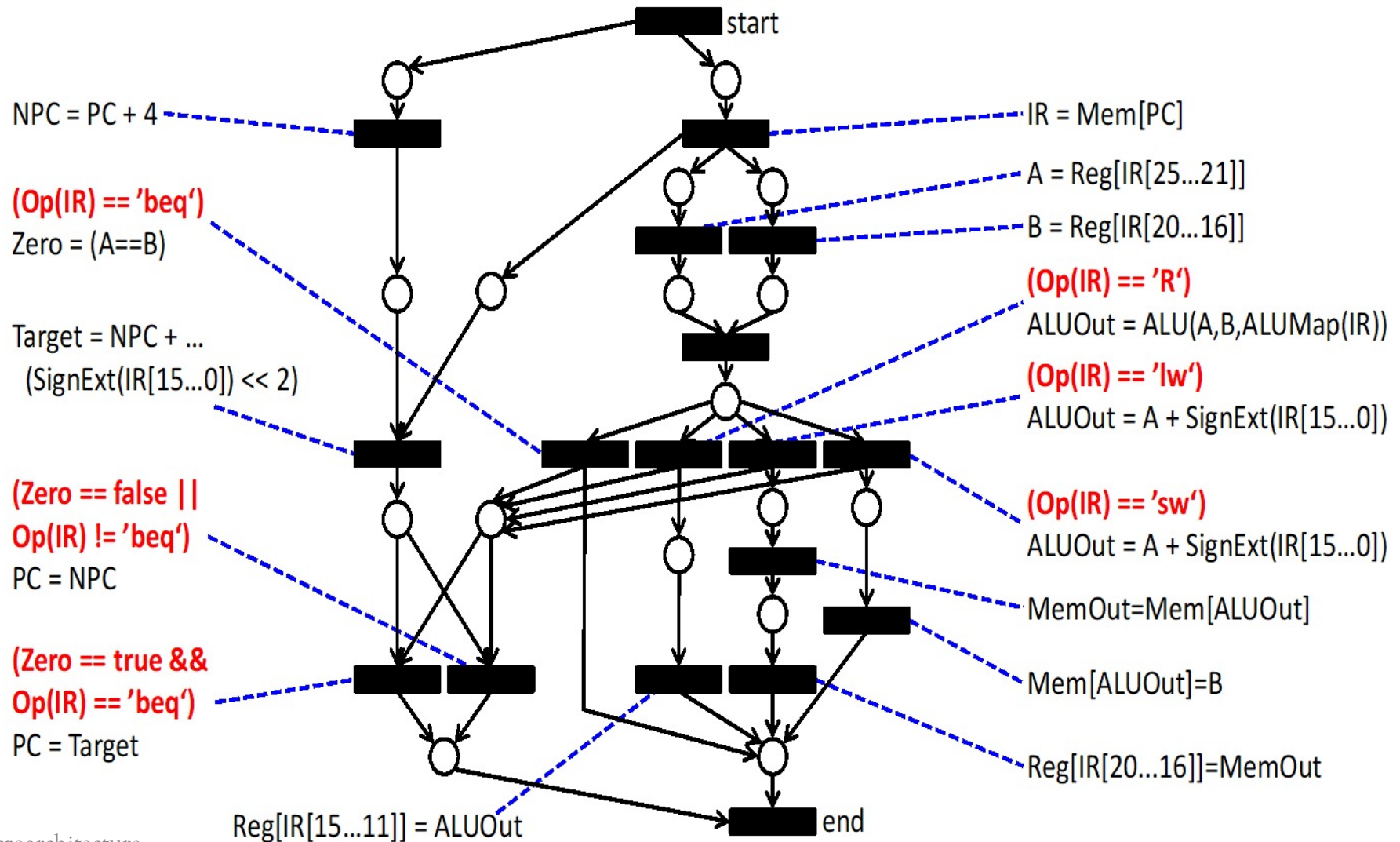


Modeling the MIPS subset

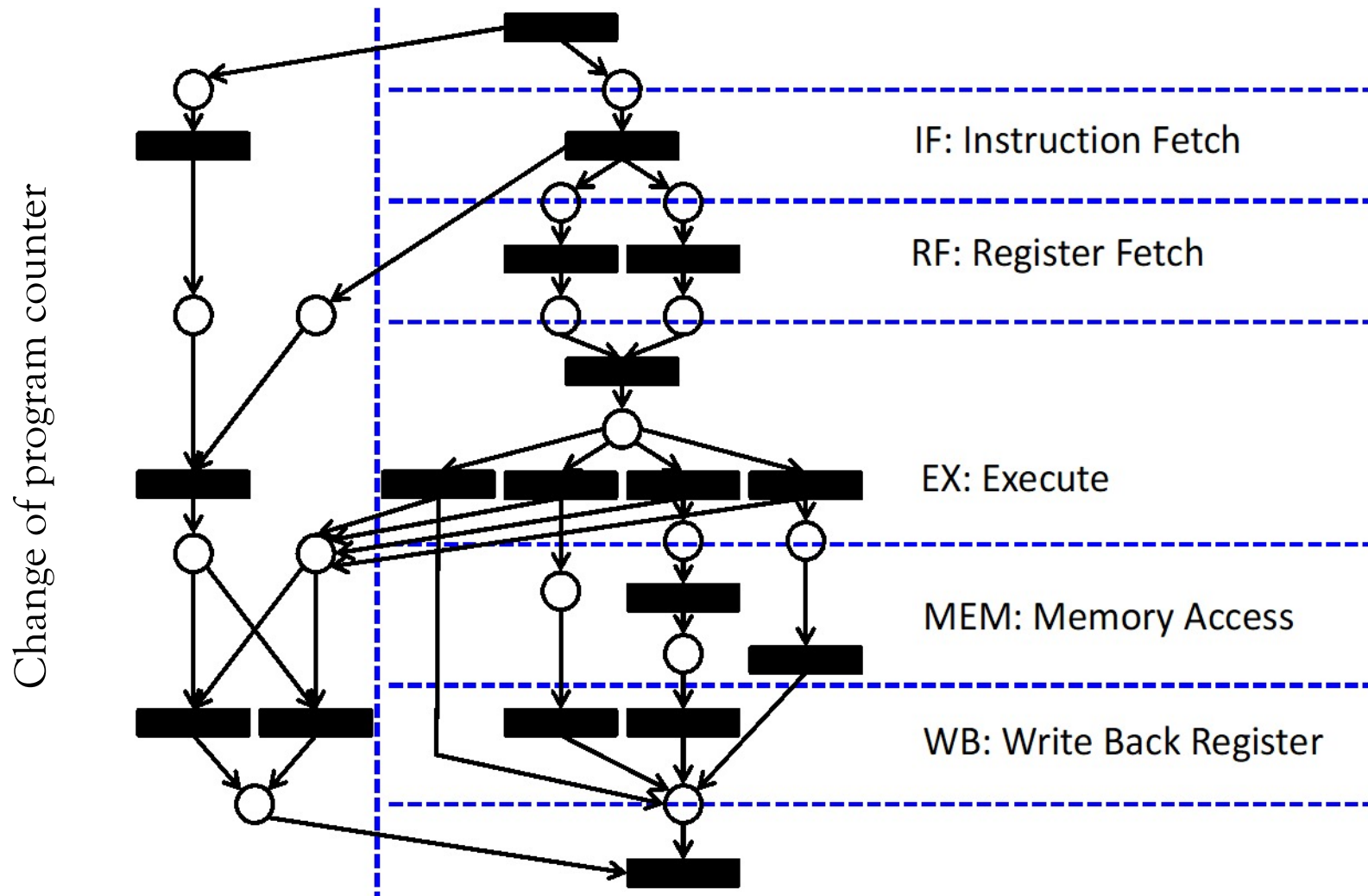
Unified Petri net:

- Merging the Petri nets of all instructions
- Definition of an additional temporary variable 'Target'.
- Connecting 'start' and 'end' for cyclic operation
- $Op[IR]$ from $\{'R', 'lw', 'sw', 'beq', 'j'\}$ defines the type of instruction and is determined from $IR[31..26]$.

MIPS refinement (without j instruction)

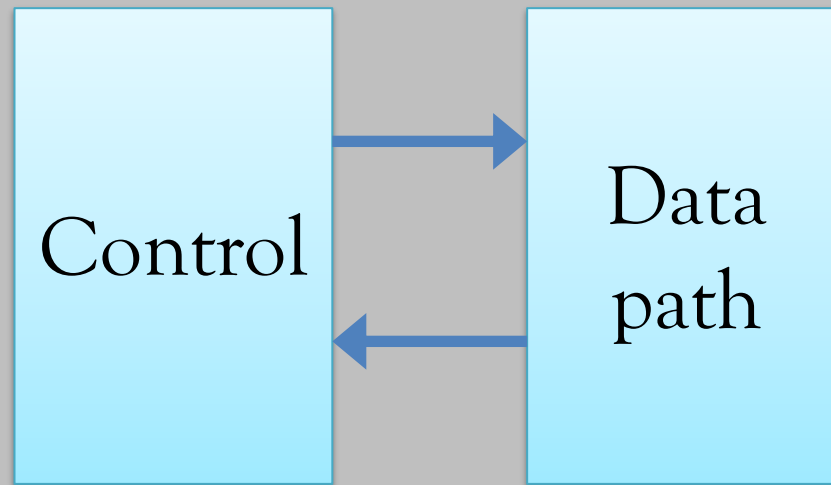


High-level structure



Single-cycle implementation

All operations are executed in a single cycle.



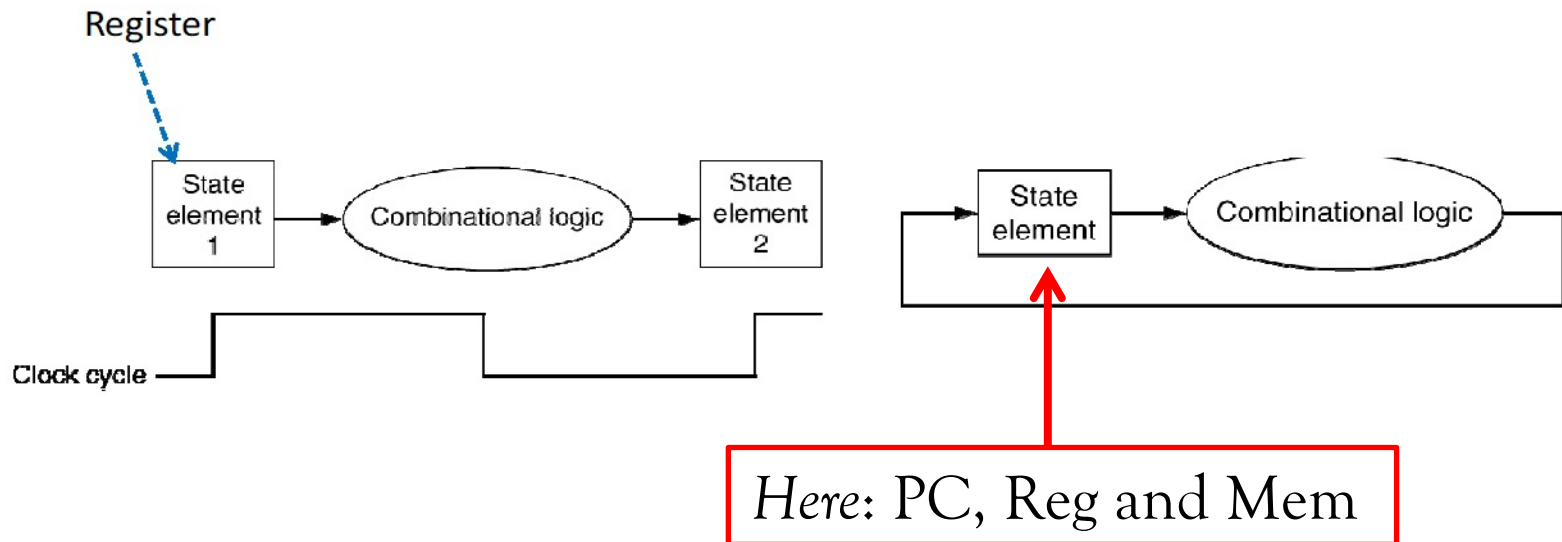
Combinatorial circuit;
implements the **structure** of the
Petri net.

Implements all **operations** that are
associated with the transitions of
the Petri net.

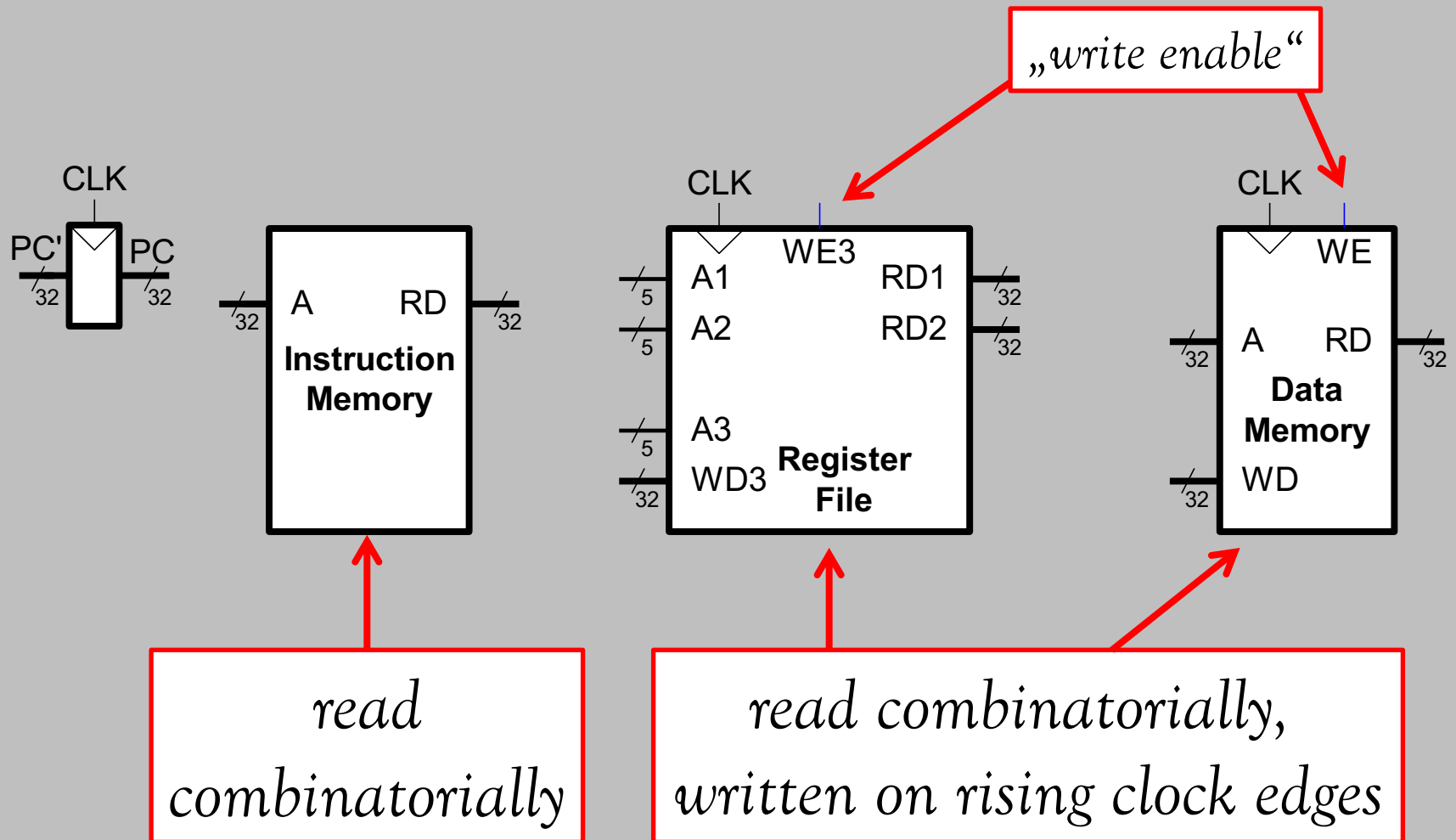
Synchronous sequential circuit design

Single-cycle system:

- Combinatorial logic processes data between the rising clock edges
- The *longest delay* between register outputs and register inputs determines the *minimal clock period*.
- Synchronous → changes to memory elements are synchronized by a clock signal

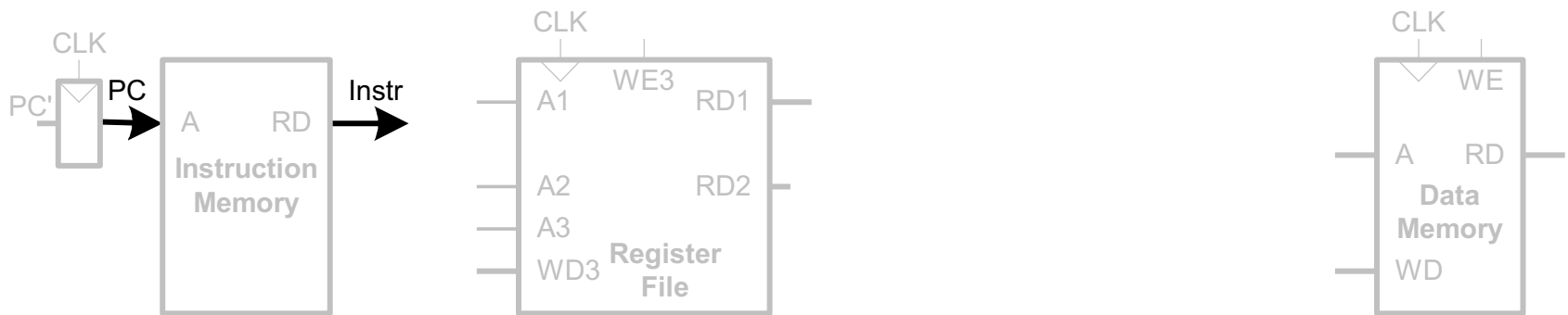


Memories in the MIPS datapath



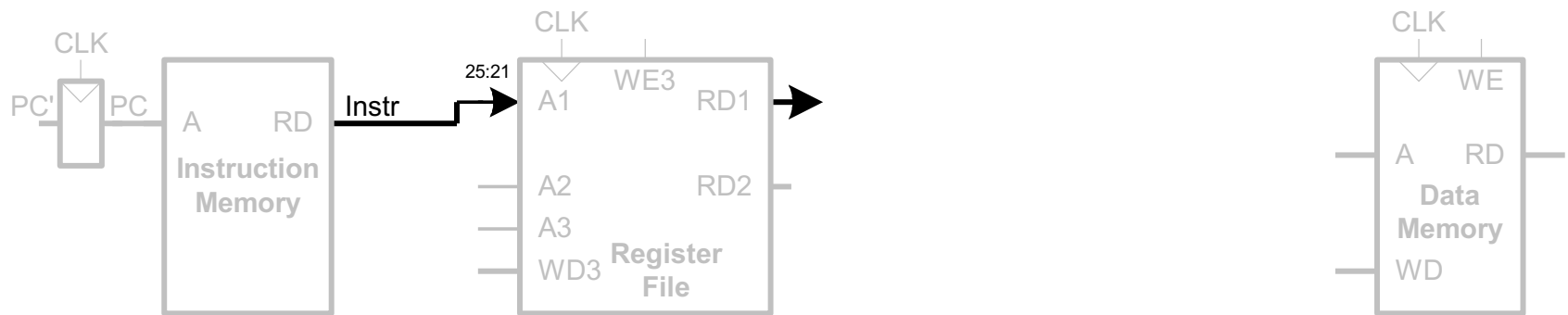
Single-cycle datapath: lw instruction

- Step 1: „Instruction fetch“



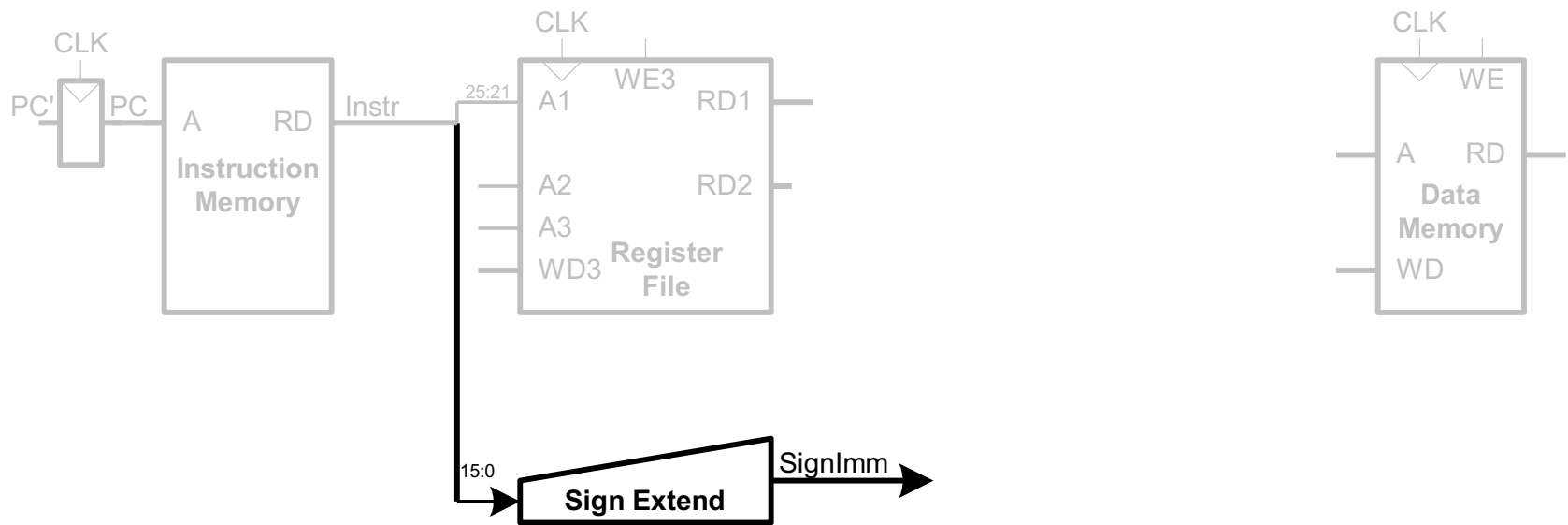
Single-cycle datapath: lw instruction

- Step 2: „Register fetch“



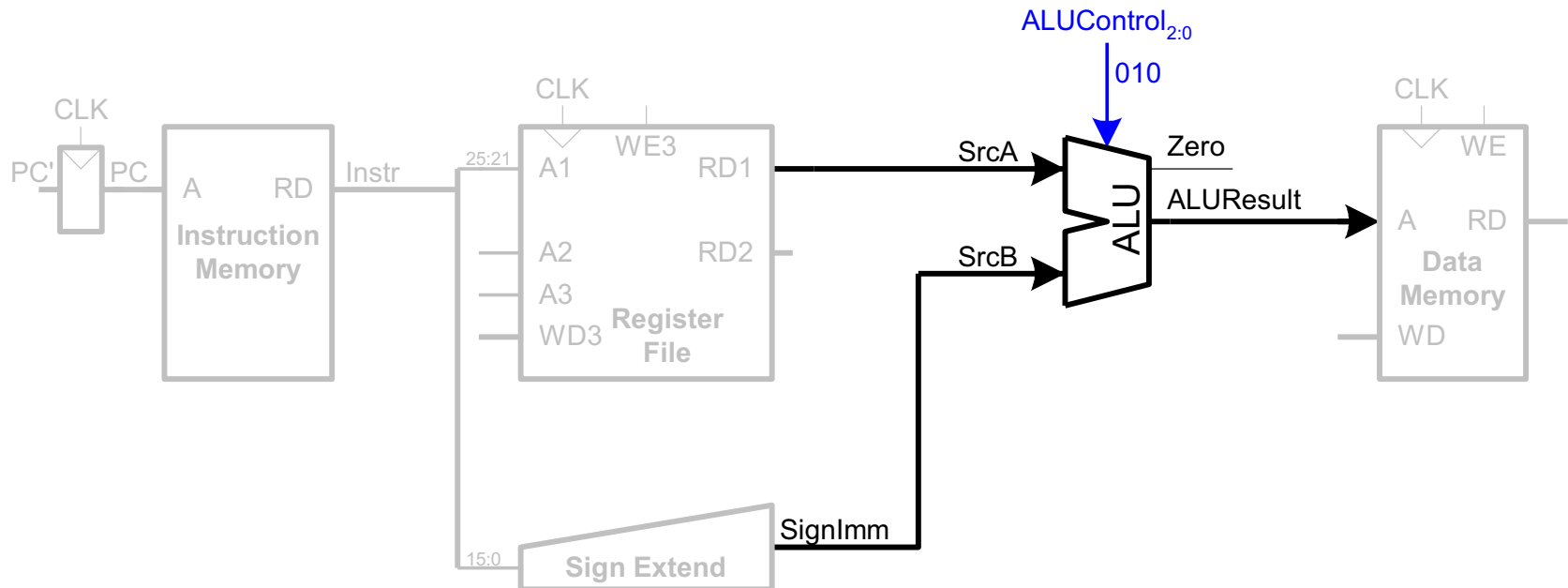
Single-cycle datapath: lw instruction

- Step 3: Sign extension



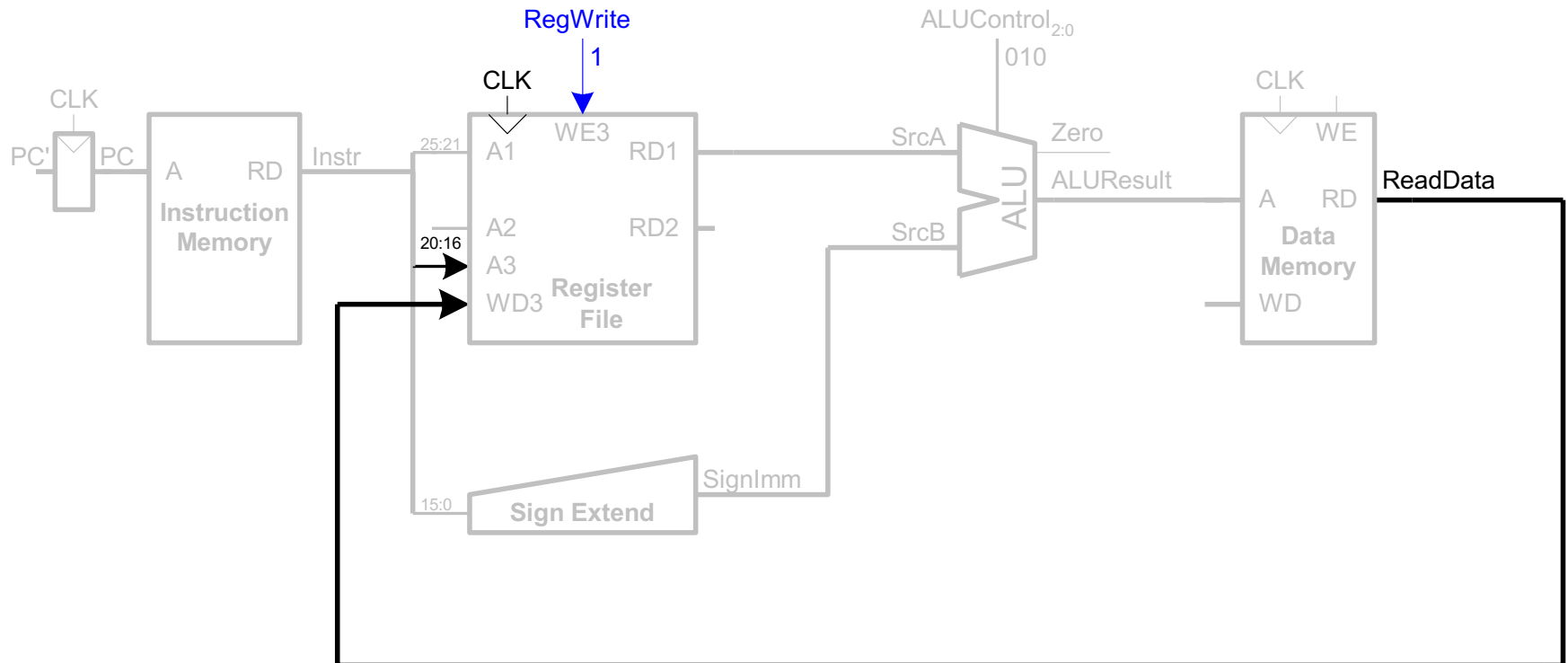
Single-cycle datapath: lw instruction

- Step 4: Compute memory address



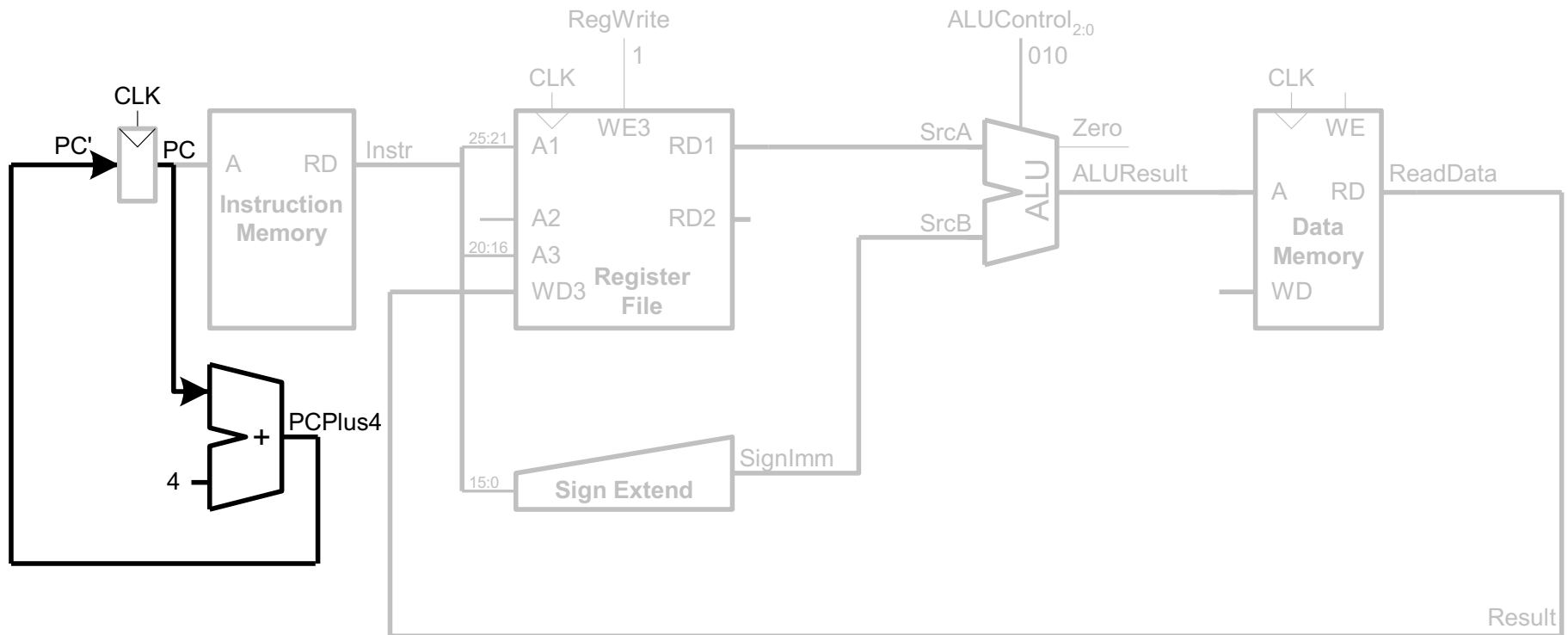
Single-cycle datapath: lw instruction

- Step 5: Load data from memory and write it back into the register file



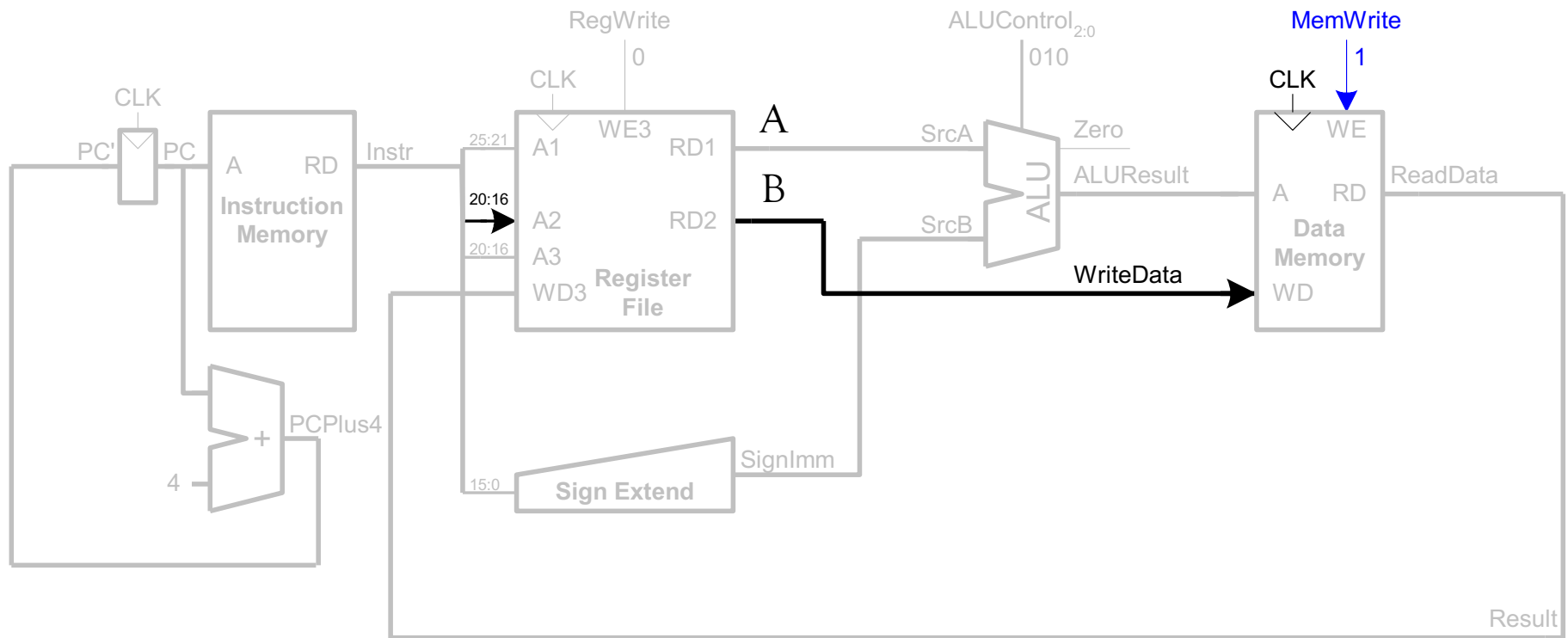
Single-cycle datapath: lw instruction

- Step 6: Determine address of next instruction



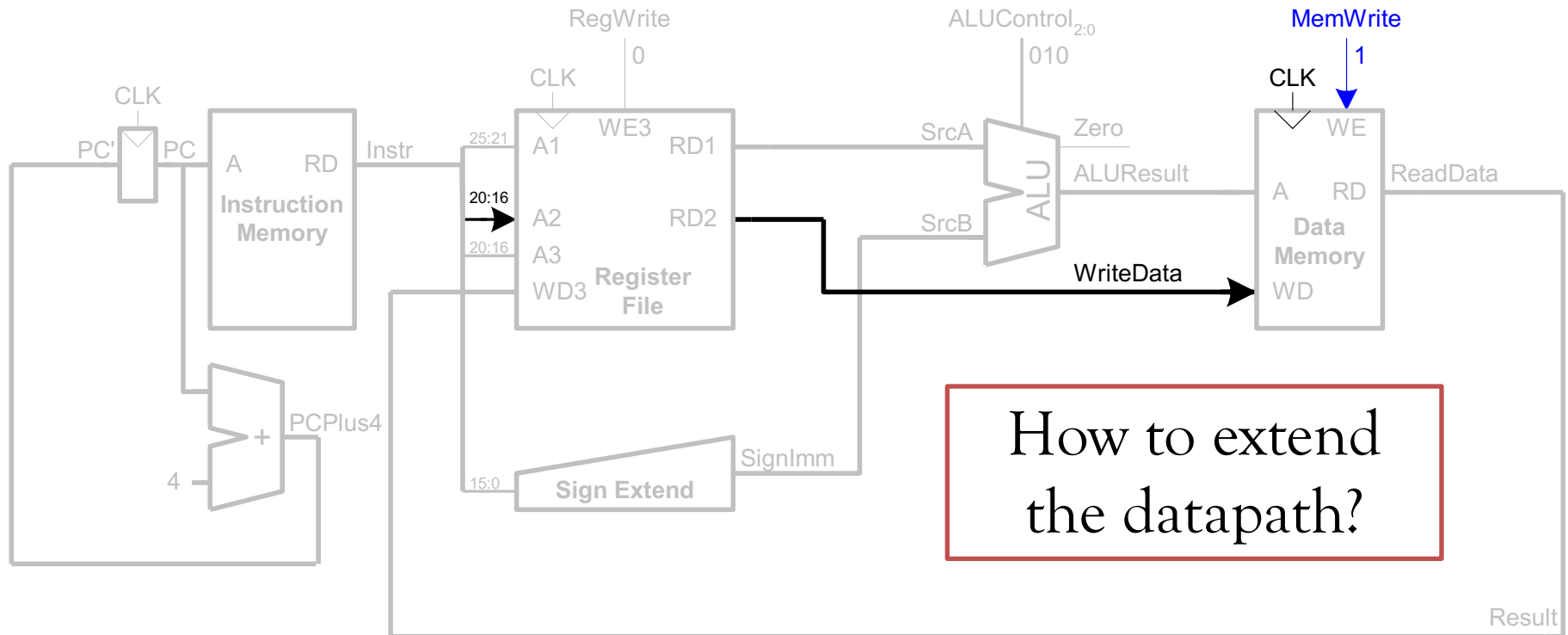
Single-cycle datapath: sw instruction

- Store data from $\text{Reg}[rt]$ into memory:



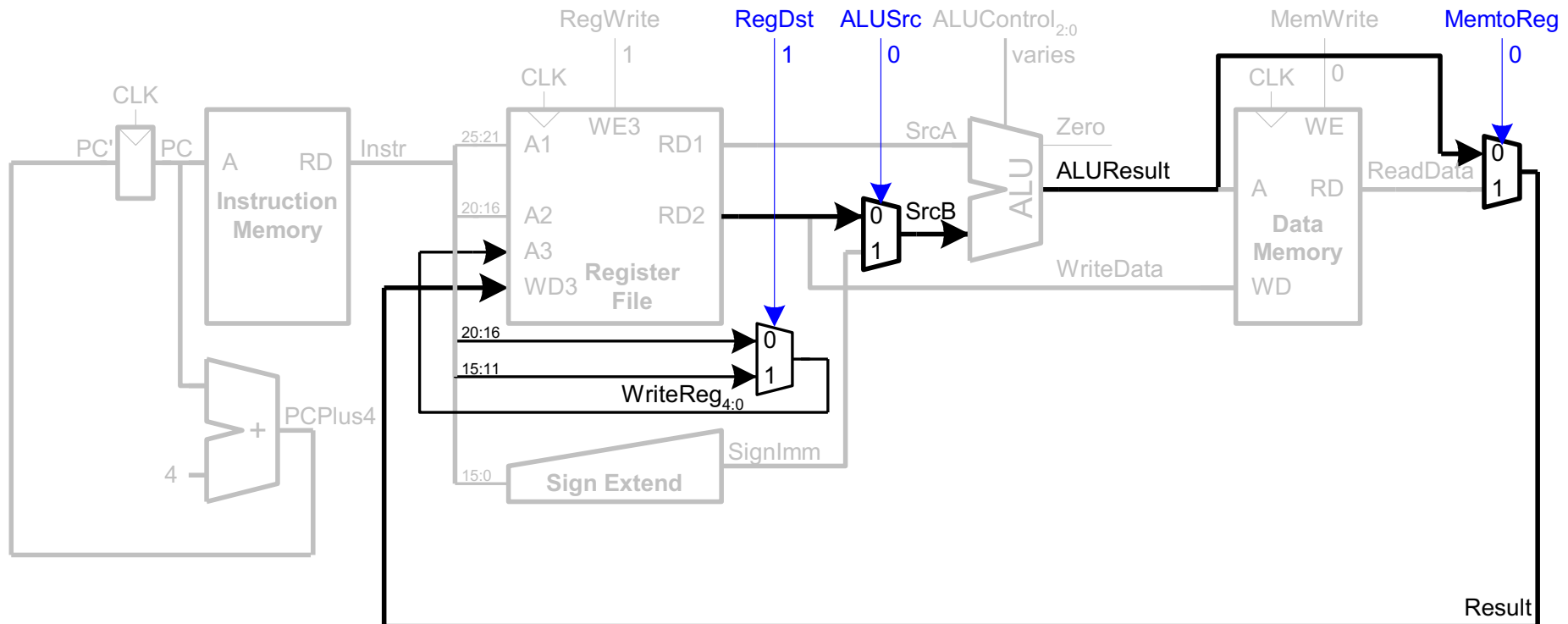
Single-cycle datapath: R-type instructions

- Read operands from `rs` and `rt`
- Write `ALUResult` into register file
- Target register: `rd` (instead of `rt` in case of `load word`)



Single-cycle datapath: R-type instructions

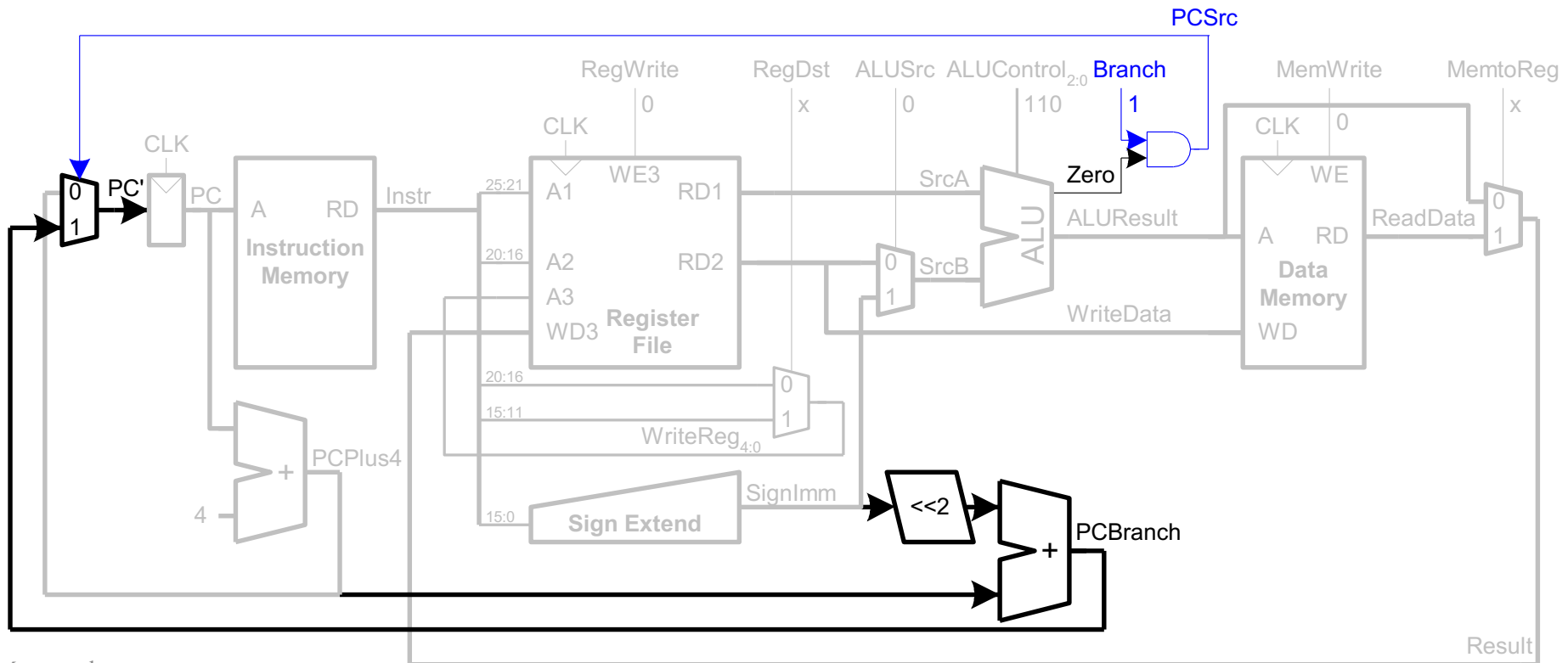
- Read operands from `rs` and `rt`
- Write `ALUResult` into register file
- Target register: `rd` (instead of `rt` in case of `load word`)



Single-cycle datapath : beq instruction

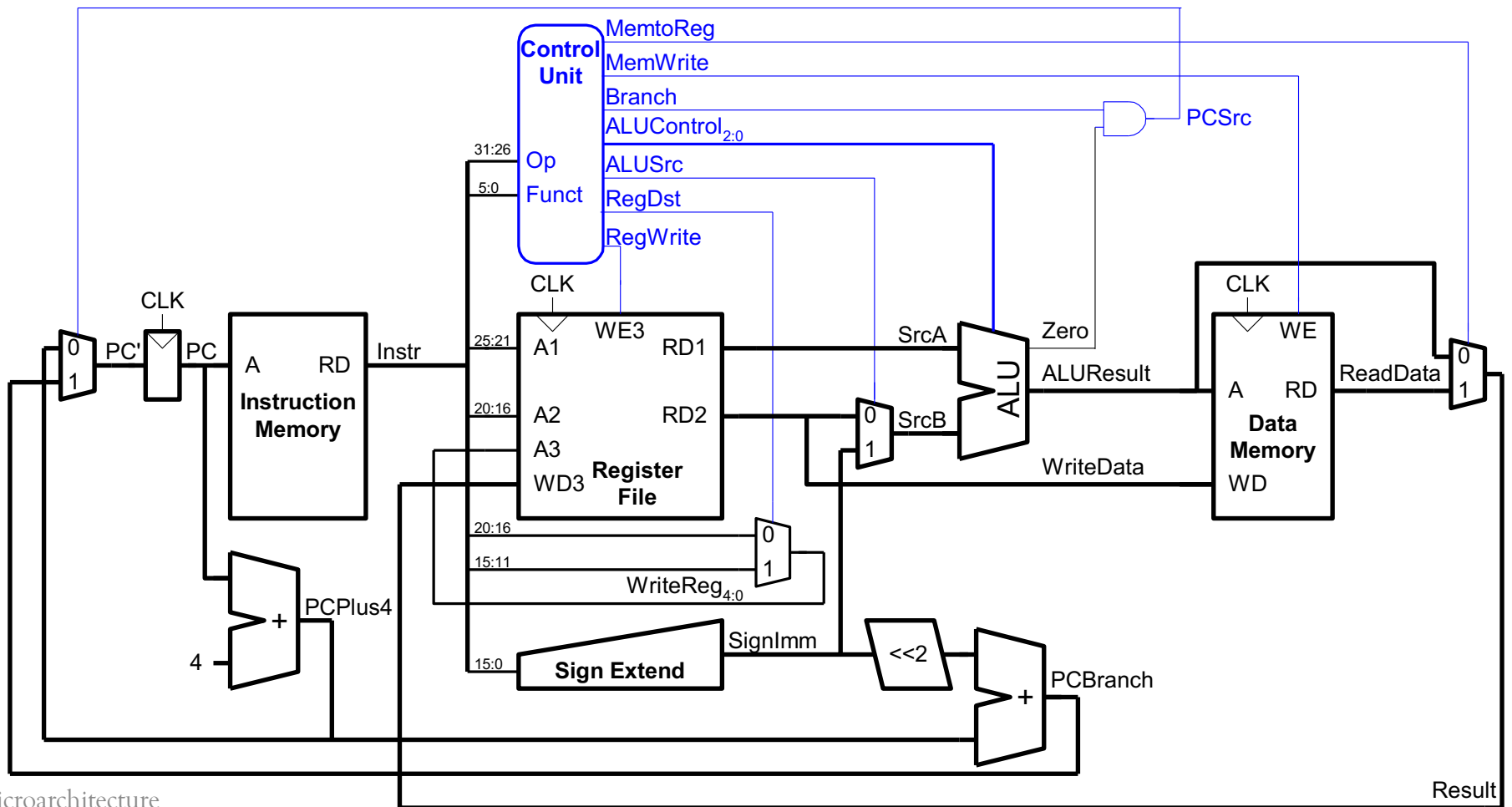
- Determine whether `rs` is equal to `rt`.
- Compute branch target address:

$$\text{BTA} = \text{SignExt}(\text{IR}[15..0]) \ll 2 + (\text{PC} + 4)$$

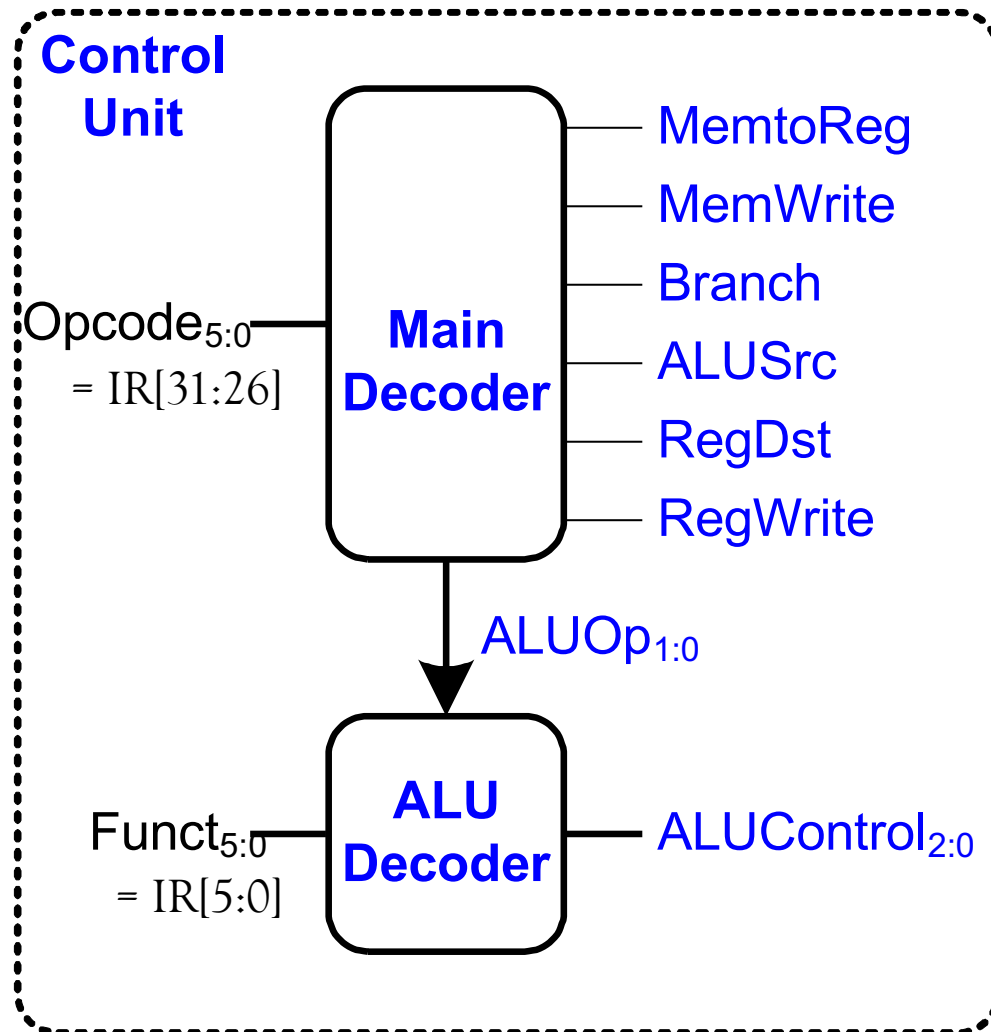


Single-cycle processor

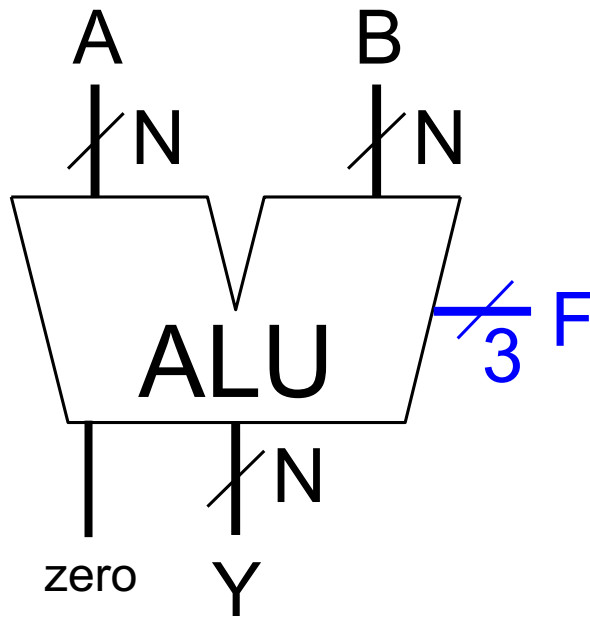
Here: with combinatorial control



Single-cycle control unit



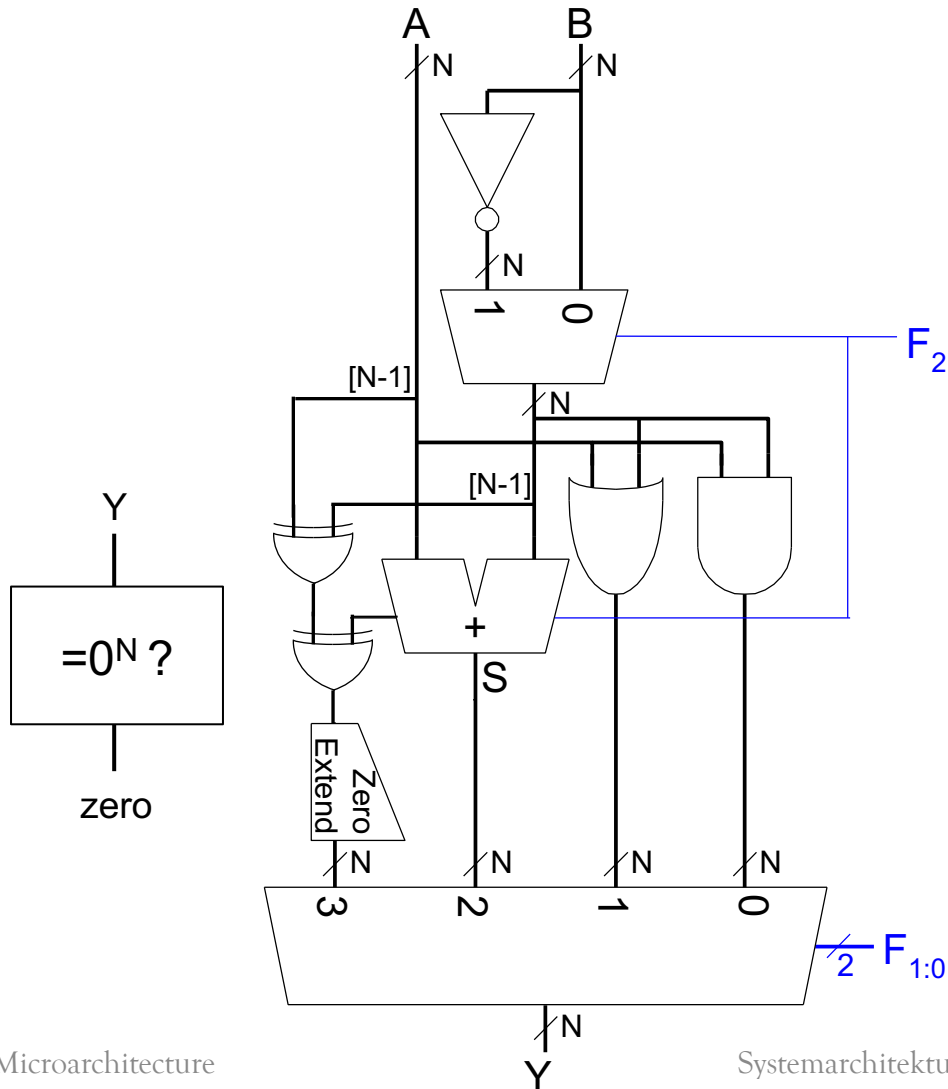
Assumptions about the ALU



$F_{2:0}$	Function
000	$A \& B$
001	$A B$
010	$A + B$
011	not used
100	$A \& \sim B$
101	$A \sim B$
110	$A - B$
111	SLT

“set less than”

ALU implementation



$F_{2:0}$	Function
000	A & B
001	A B
010	A + B
011	not used
100	A & \sim B
101	A \sim B
110	A - B
111	SLT

Control unit: ALU decoder

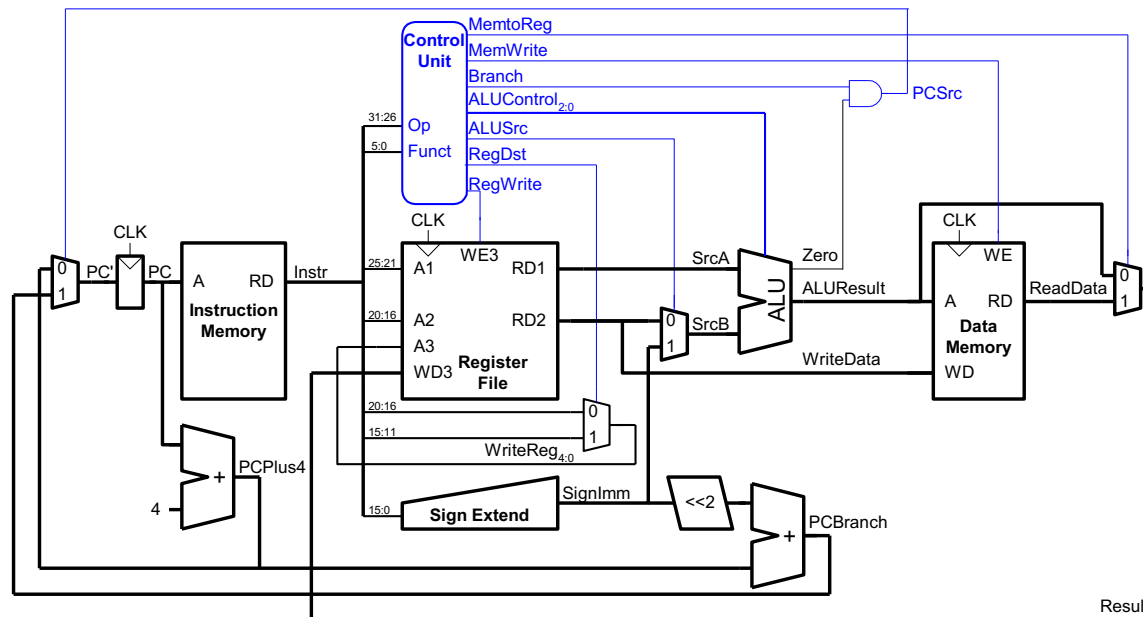
ALUOp _{1:0}	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used

X = Don't Care

ALUOp _{1:0}	Funct	ALUControl _{2:0}
00	X	010 (Add)
01	X	110 (Subtract)
10	100000 (add)	010 (Add)
10	100010 (sub)	110 (Subtract)
10	100100 (and)	000 (And)
10	100101 (or)	001 (Or)
10	101010 (slt)	111 (SLT)
11	X	X

Control unit: Main decoder

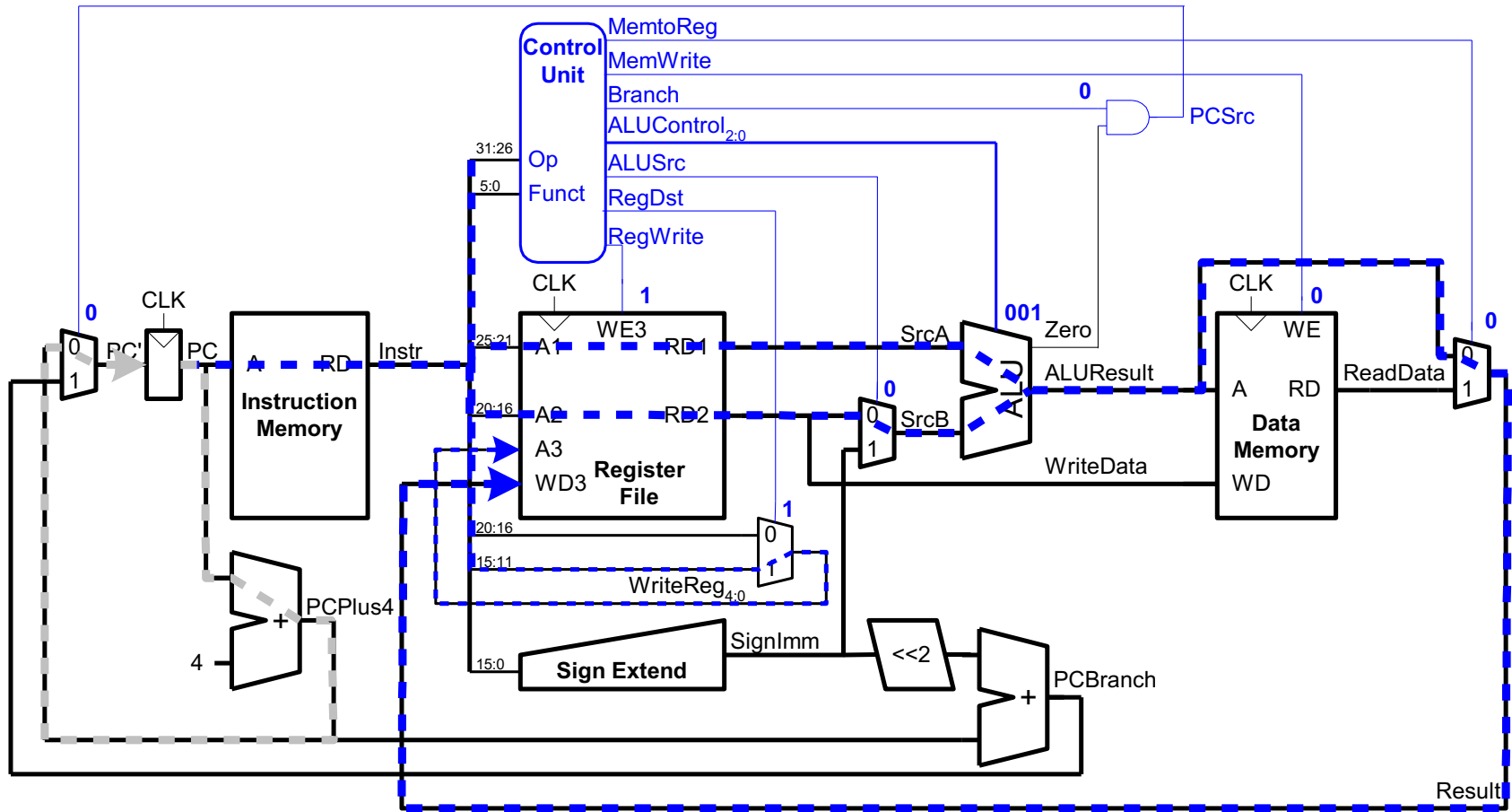
Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
R-type	000000							
lw	100011							
sw	101011							
beq	000100							



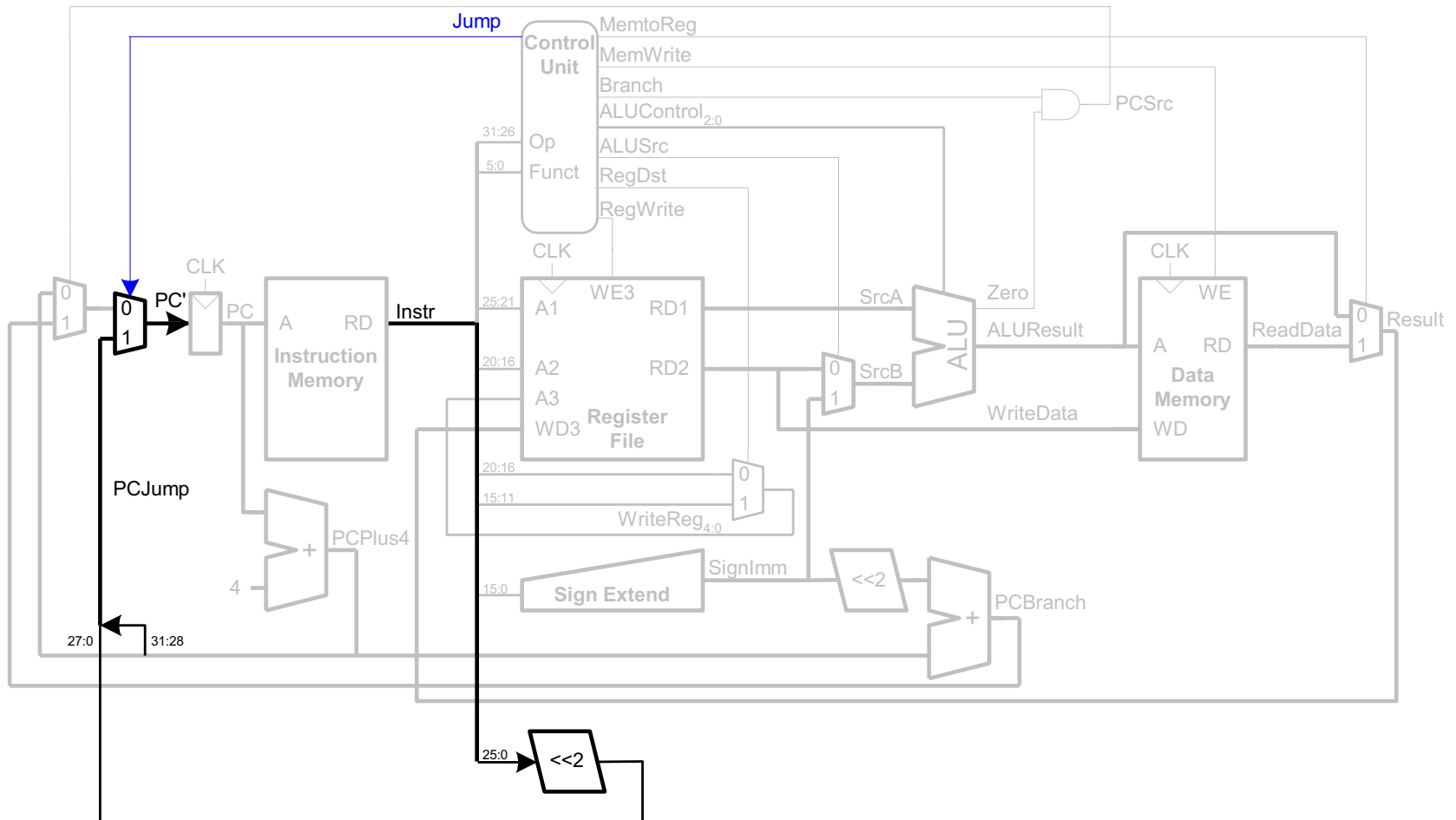
Control unit: Main decoder

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

Example: Single-cycle processor: or



Single-cycle datapath: Extension for j



Control unit: Main decoder:

Extension for j

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
j	000010	0	X	X	X	0	X	XX	1

Conclusions

- **Petri nets** to describe the interpretation of MIPS instructions
 - **Datapath** corresponds to *components* that implement the transitions of the Petri net
 - **Control** derived from *structure* and *conditions* of the Petri net
- Control unit can be divided into **main decoder** and **ALU decoder**
- In the single-cycle processor the control is *combinatorial*