Chapter 7

MICROARCHITECTURE

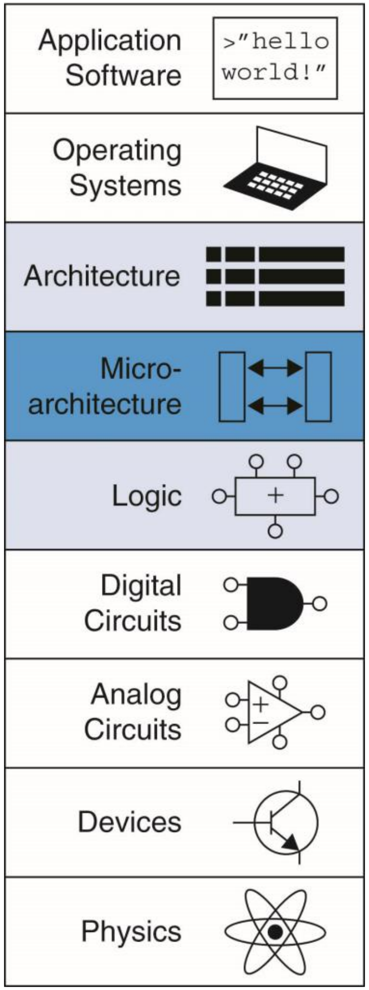
***Digital Design and Computer Architecture*, 2nd Edition**

David Money Harris and Sarah L. Harris

ELSEVIER

Chapter 7 :: Topics

- **Introduction**
- **Performance Analysis**
- **Single-Cycle Processor**
- **Pipelined Processor**
- **Exceptions**
- **Advanced Microarchitecture**



MICROARCHITECTURE

Introduction

- **Microarchitecture:** the implementation of an architecture in hardware
- Processor:
  - **Datapath:** functional blocks
  - **Control:** control signals

Application Software	programs
Operating Systems	device drivers
Architecture	instructions registers
Micro-architecture	datapaths controllers
Logic	adders memories
Digital Circuits	AND gates NOT gates
Analog Circuits	amplifiers filters
Devices	transistors diodes
Physics	electrons

ELSEVIER

Microarchitecture

- Multiple implementations for a single architecture:
  - **Single-cycle:** Each instruction executes in a single cycle
  - **Multicycle:** Instructions are broken into series of shorter steps Each instruction executes in n cycles, where n varys according to the instr.
  - **Pipelined:** Each instruction broken up into series of steps & multiple instructions execute at once (Note: AMD and Intel pipelines are different, for the same IA-32 architecture (a.k.a. x86 ISA)

Processor Performance

- Program execution time

Execution Time = (#instructions)(cycles/instruction)(seconds/cycle)

- Definitions:
  - IC: Instruction Count (= #instructions)
  - CPI: Cycles/Instruction
  - clock period: seconds/cycle
  - IPC: Instructions/Cycle (= 1/CPI)
- Challenge is to satisfy constraints of:
  - Cost
  - Power
  - Performance

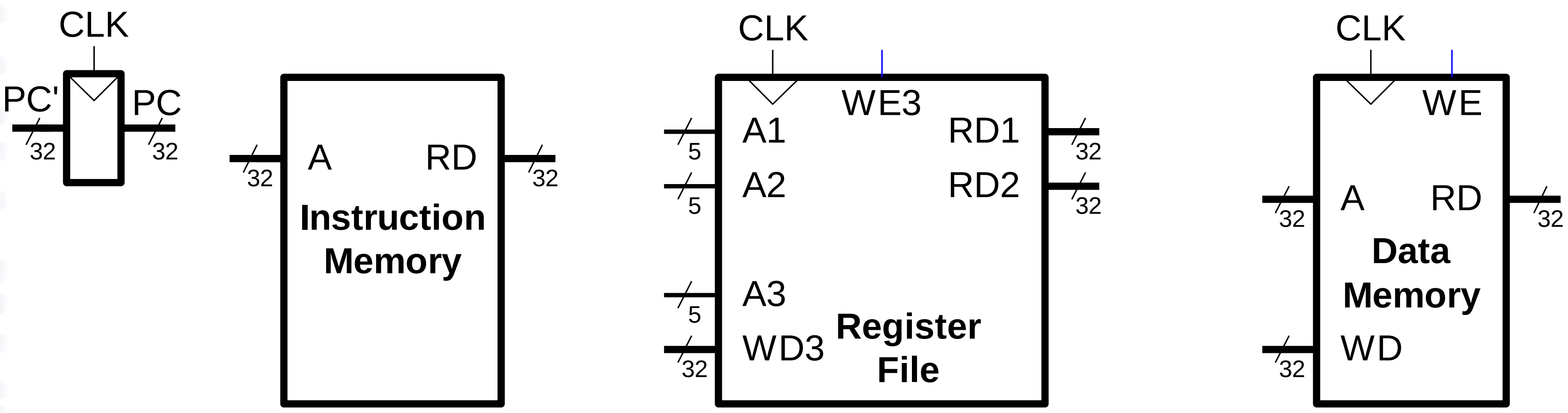
MIPS Processor

- Consider subset of MIPS instructions:
  - R-type instructions: `and`, `or`, `add`, `sub`, `slt`
  - Memory instructions: `lw`, `sw`
  - Branch instructions: `beq`

Architectural State

- Determines everything about a processor:
  - PC and special registers
  - Register File
  - Memory

MIPS State Elements



Plus the HI and LO registers

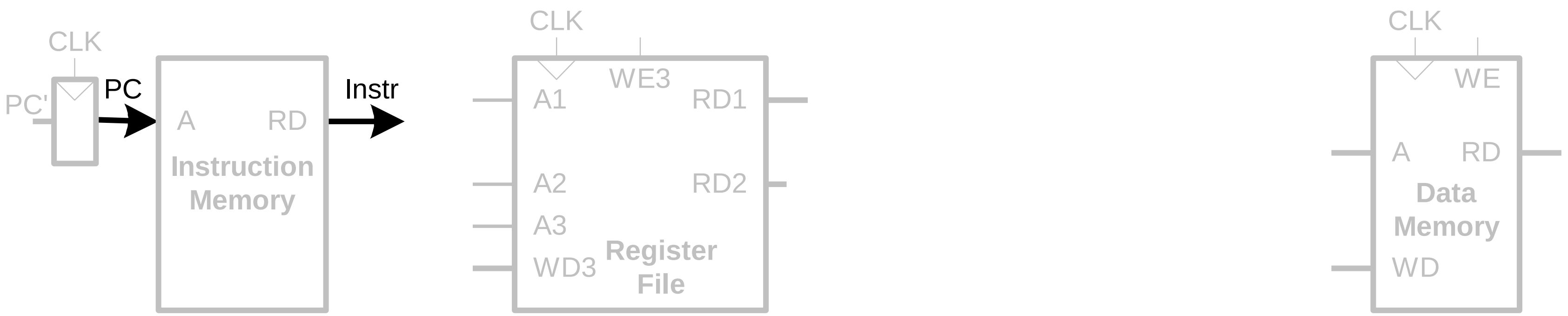
Single-Cycle MIPS Processor

- Datapath—design it 1st, to make the instruction actions possible
- Control—design it 2nd, to make them happen

Single-Cycle Datapath: `lw` fetch

STEP 1: Fetch instruction

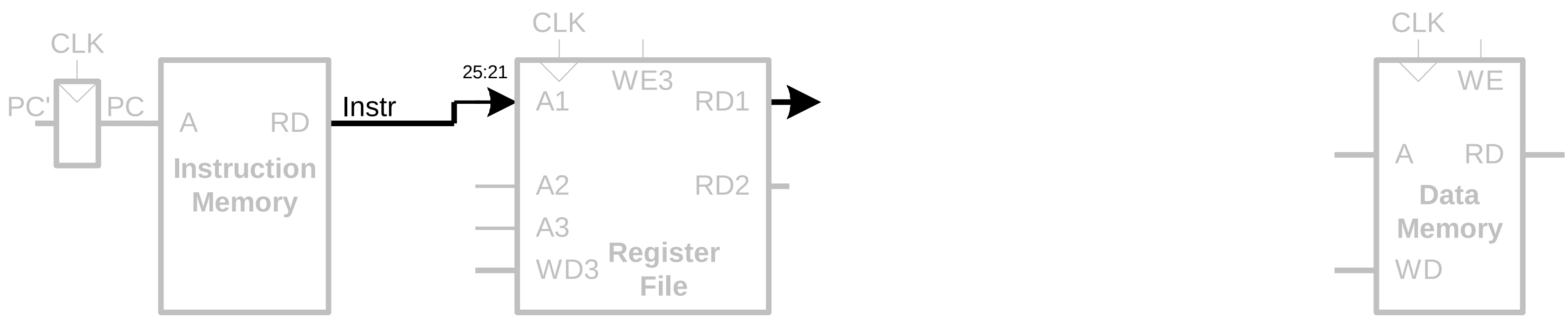
IM[PC]



Single-Cycle Datapath: `lw` Register Read

STEP 2: Read source operands from RF

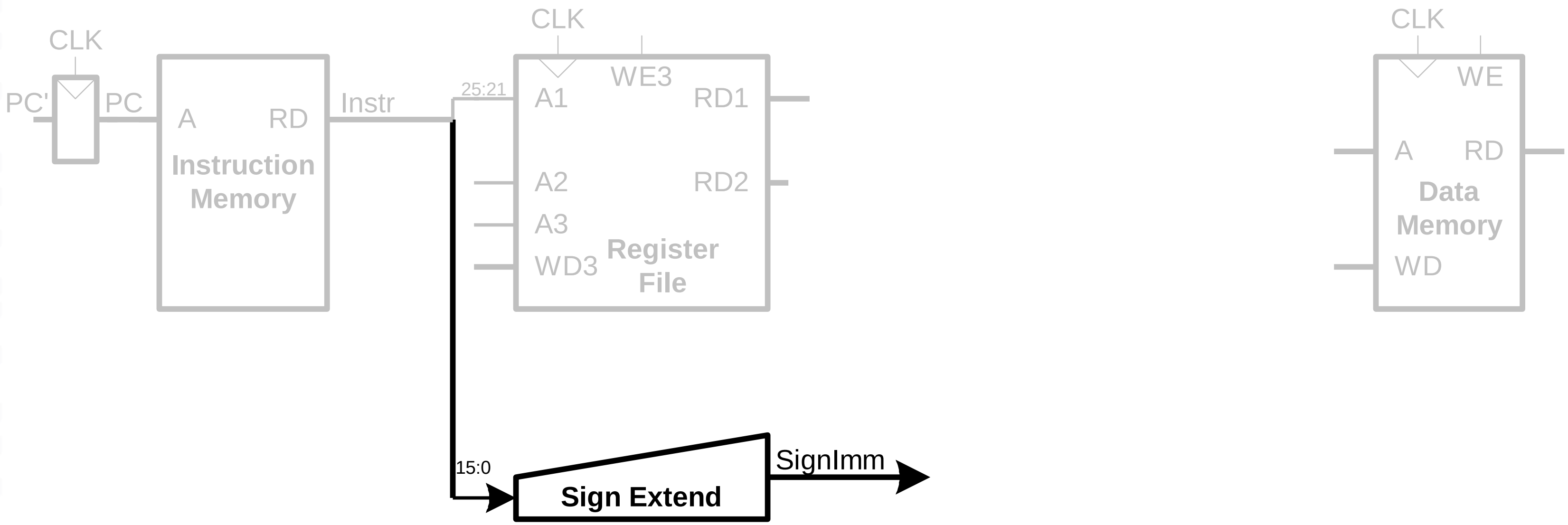
RF[rs]   or RF[Instr(25:21)]



Single-Cycle Datapath: `lw` Immediate

STEP 3: Sign-extend the immediate

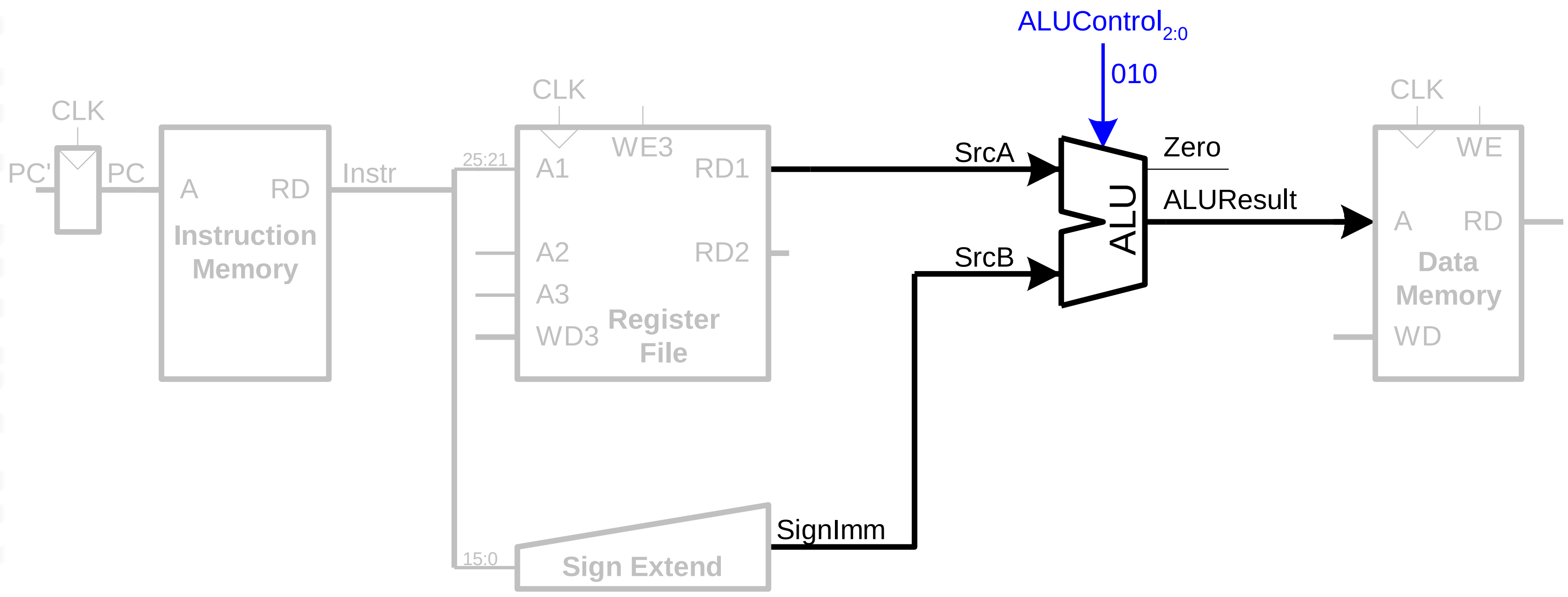
SignExt(immed)



Single-Cycle Datapath: `lw` address

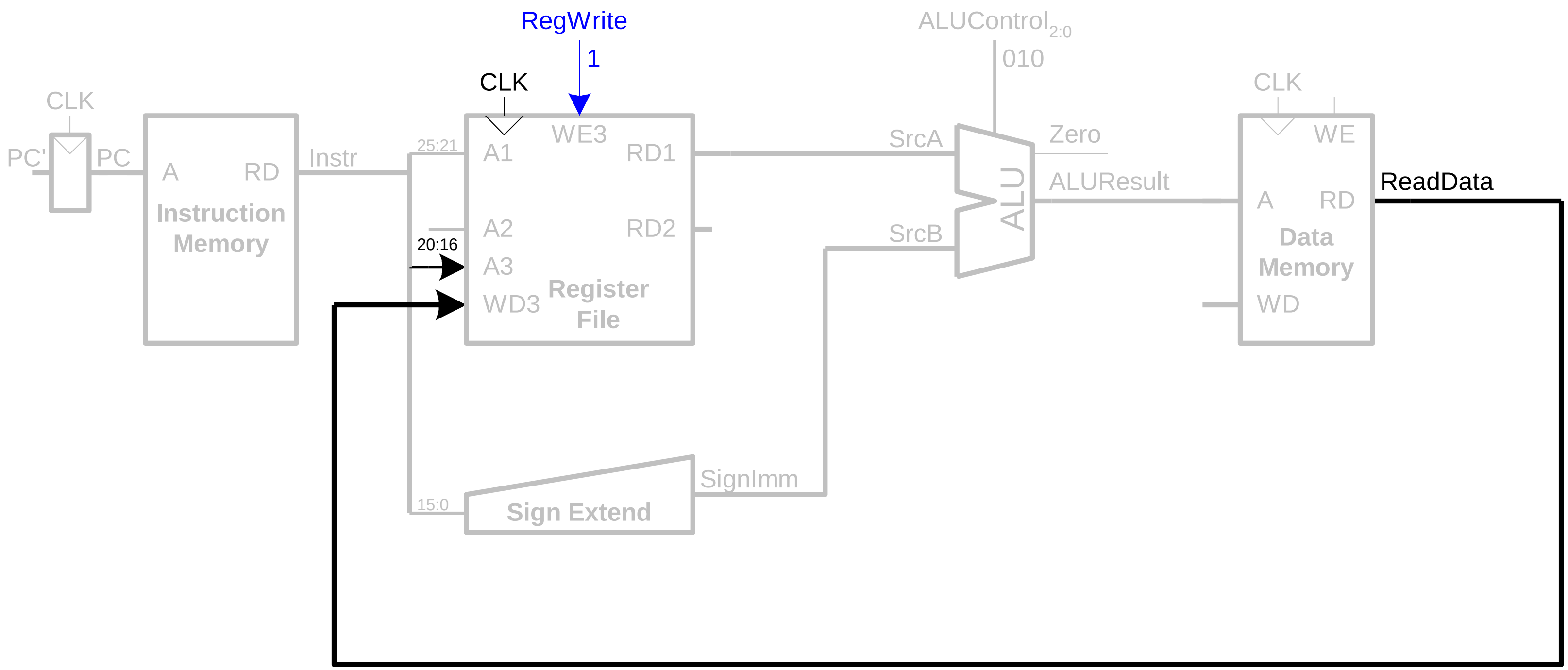
STEP 4: Compute the memory address

$$addr = RF[rs] + SignExt(immed)$$



Single-Cycle Datapath: `lw` Memory Read

- **STEP 5:** Read data from memory and write it back to register file: RF[rt] ← DM[addr]

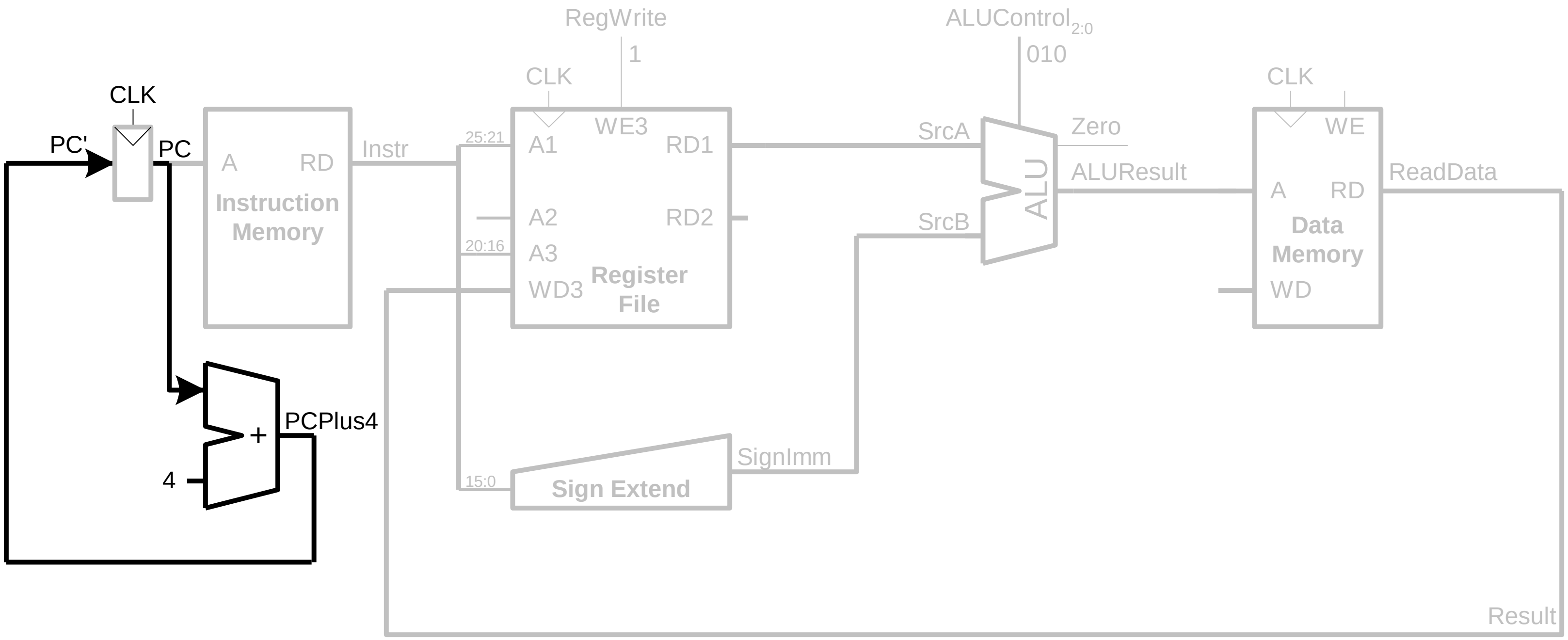


ELSEVIER

Single-Cycle Datapath: `lw` PC Increment

STEP 6: Determine address of next instruction

PC ← PC + 4



Full RTL Expression for `lw`

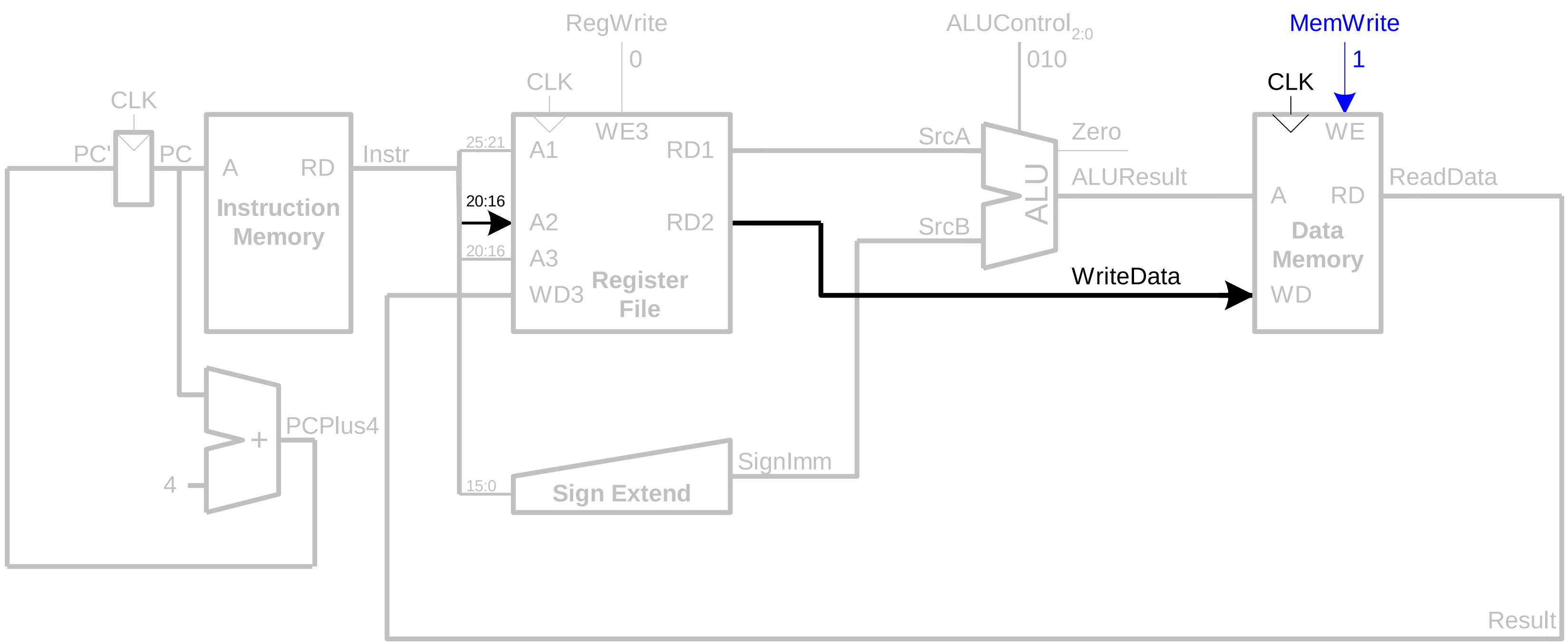
IM[PC]

RF[rt] ← DM[RF[rs] + SignExt(immed)]

PC ← PC + 4

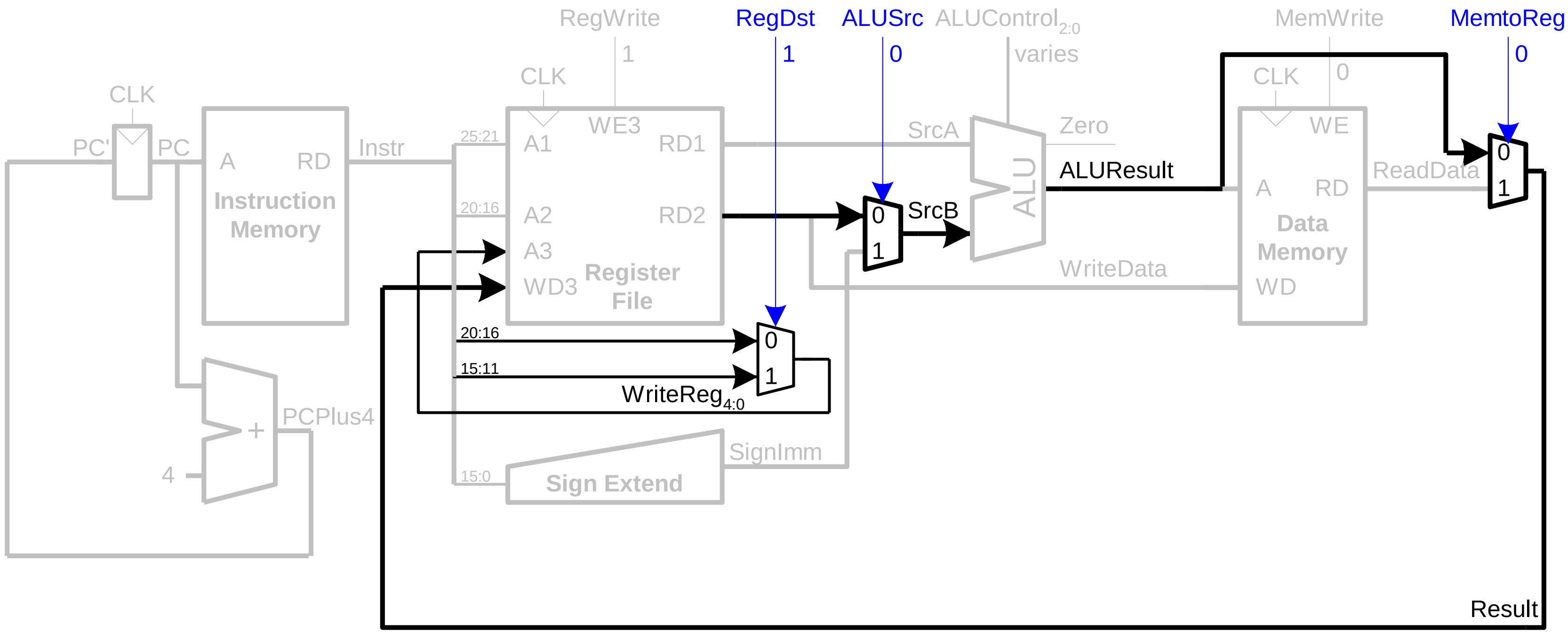
Single-Cycle Datapath: `sw`

Write data in `rt` to memory: DM[addr]←RF[rt]



Single-Cycle Datapath: R-Type

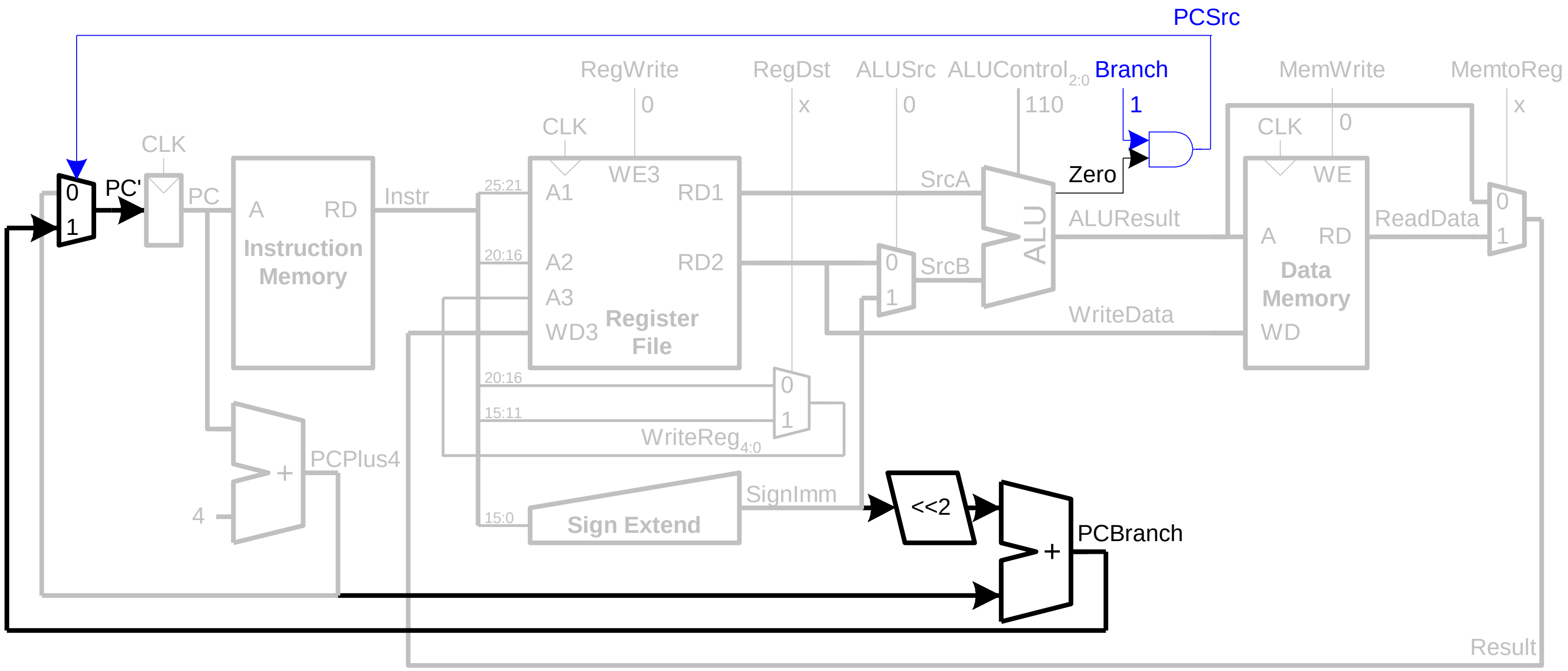
- Read from `rs` and `rt`
- Write *ALUResult* to register file
- Write to `rd` (instead of `rt`)   RF[rd] ← RF[rs] op RF[rt]



Single-Cycle Datapath: beq

- Determine whether values in `rs` and `rt` are equal
- Calculate branch target address:

  BTA = PC + 4 + SignExt(immed)<< 2   # <<2 = 4x



RTL Expression for `beq`

IM[PC]

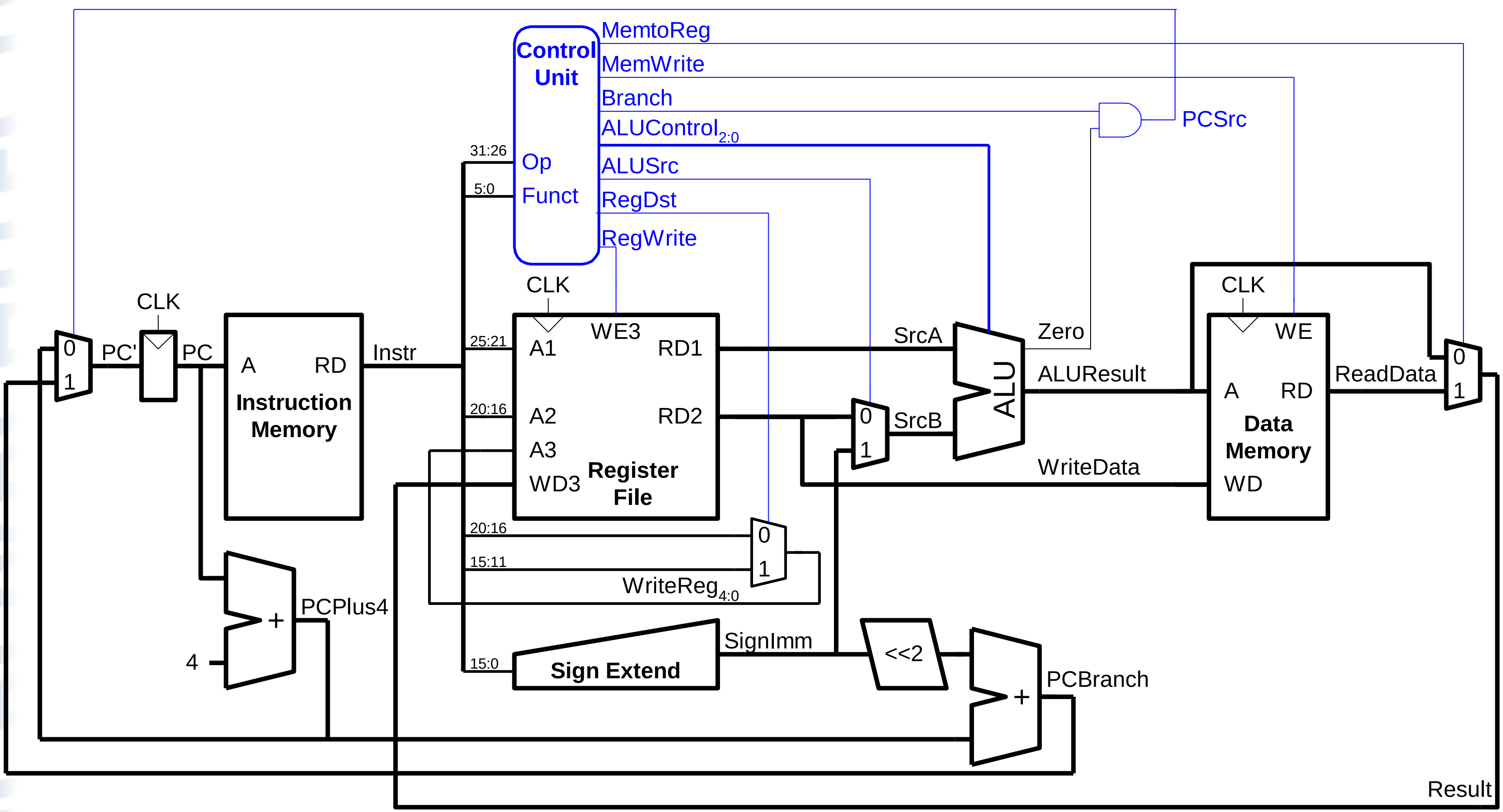
if (RF[rs] - RF[rt] == 0)

  PC $\leftarrow$ BTA

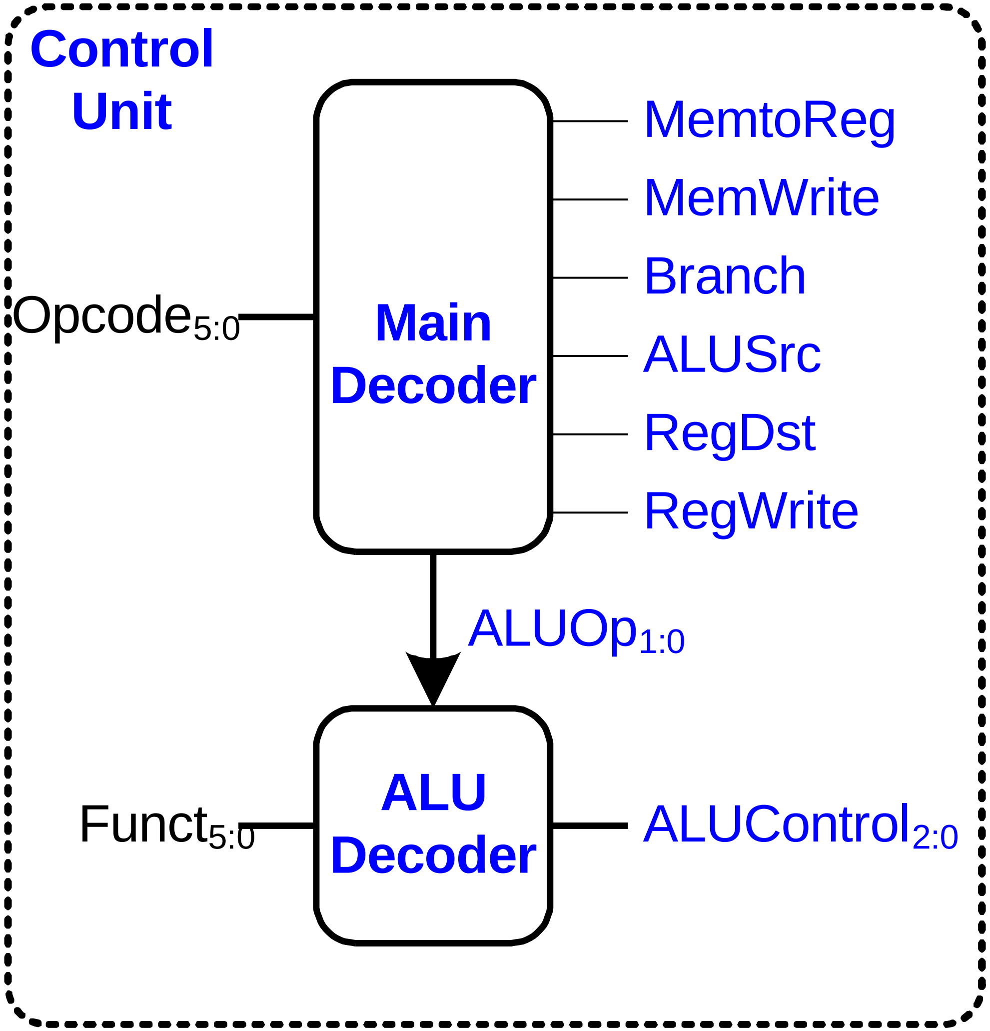
else

  PC $\leftarrow$ PC + 4

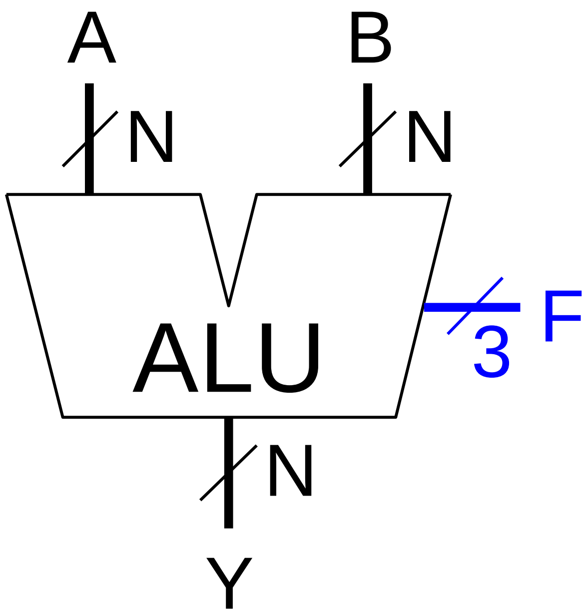
Single-Cycle Processor



Single-Cycle Control

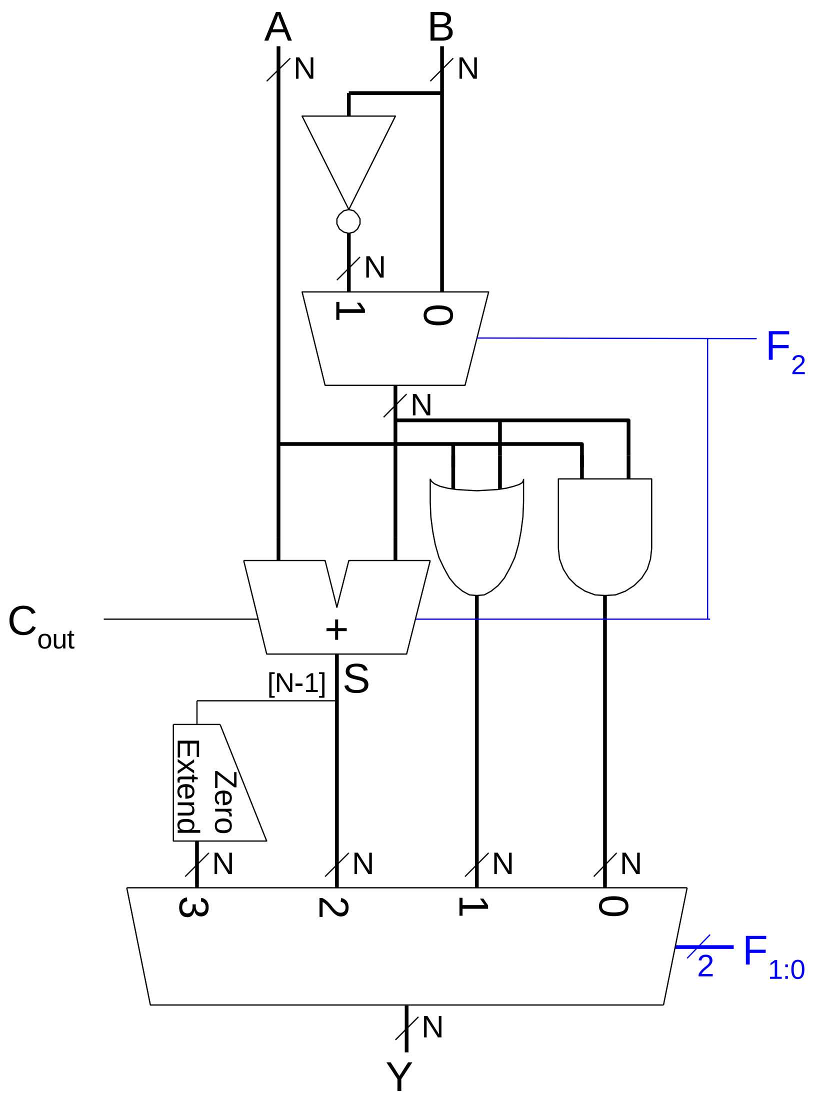


Review: ALU



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

Review: ALU



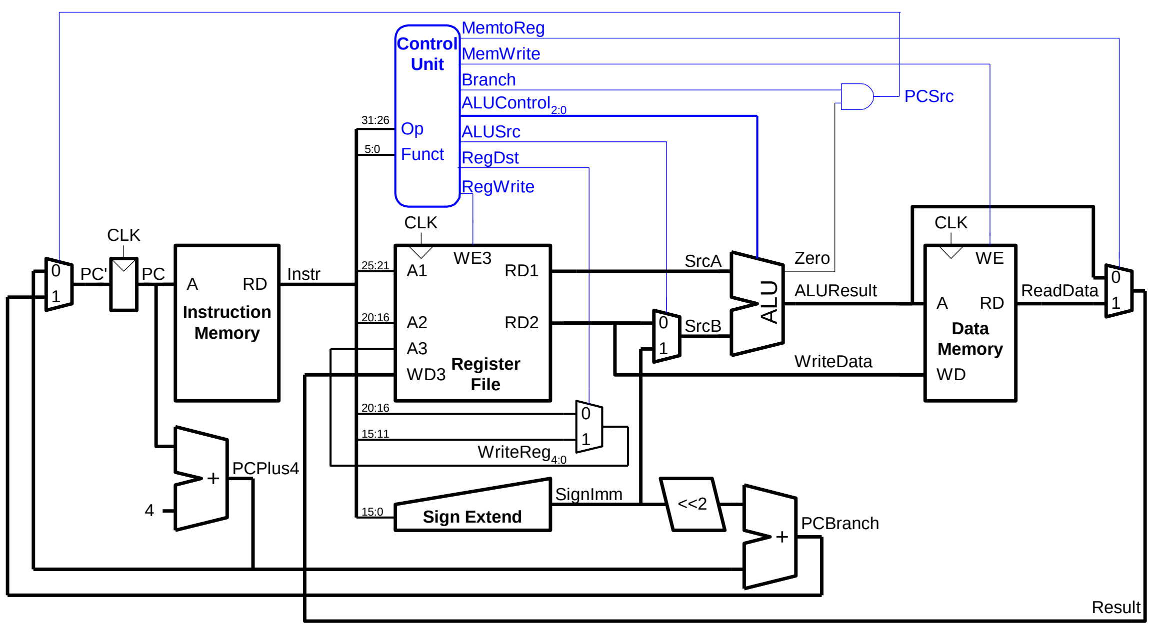
Control Unit: ALU Decoder

$ALUOp_{1:0}$	Meaning
00	Add (for lw, sw)
01	Subtract (for beq)
10	Look at funct (R-type)
11	Not Used

$ALUOp_{1:0}$	funct	$ALUControl_{2:0}$
00	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (`add`)	010 (Add)
1X	100010 (`sub`)	110 (Subtract)
1X	100100 (`and`)	000 (And)
1X	100101 (`or`)	001 (Or)
1X	101010 (`slt`)	111 (SLT)

Control Unit: Main Decoder

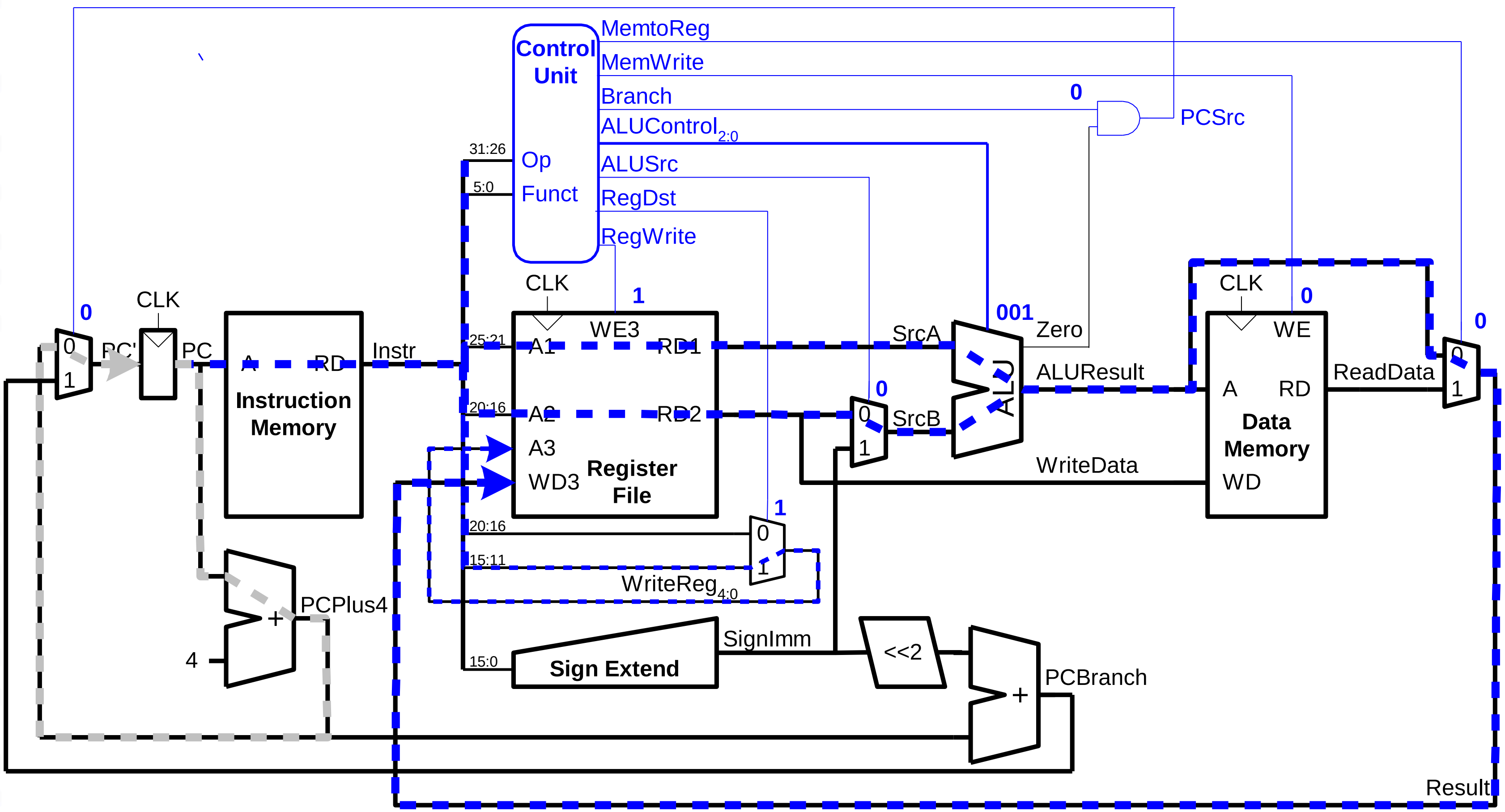
Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000							
`lw`	100011							
`sw`	101011							
`beq`	000100							



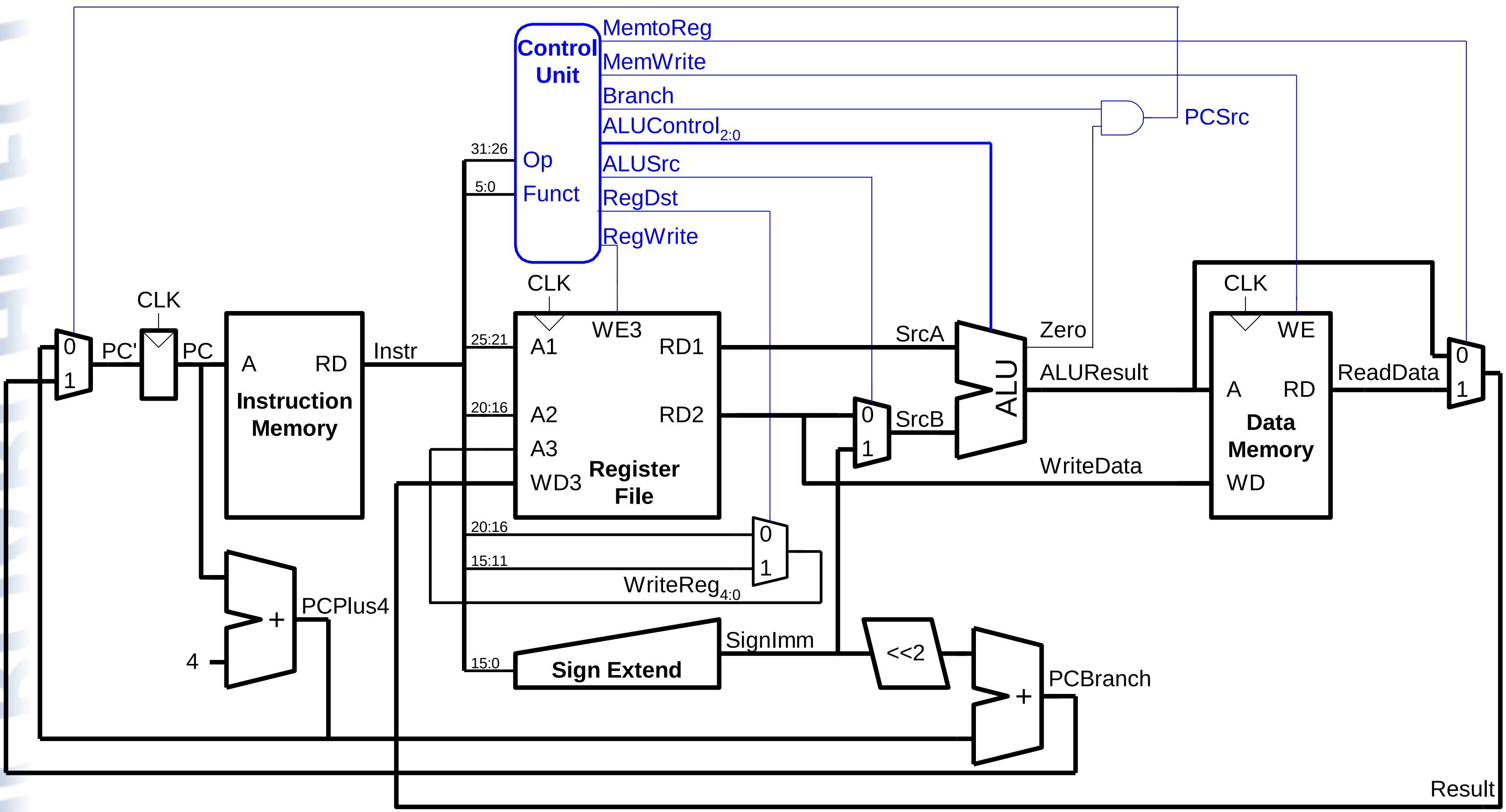
Control Unit: Main Decoder

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

Single-Cycle Datapath: `or`



Extended Functionality: `addi`



No change to datapath

ELSEVIER

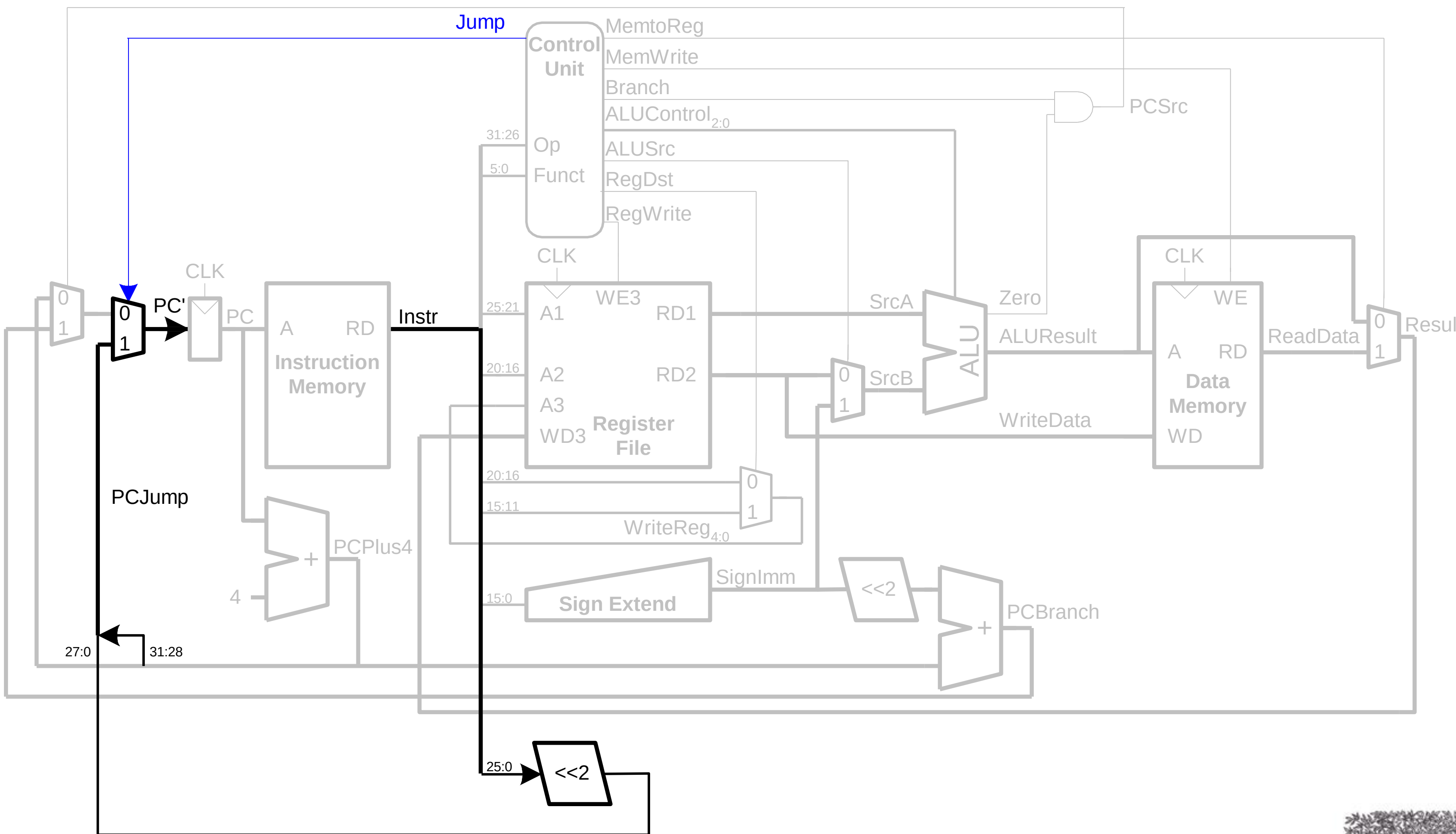
Main Decoder table: `addi`

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
`addi`	**001000**							

Main Decoder table: addi

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
`addi`	**001000**	**1**	**0**	**1**	**0**	**0**	**0**	**00**

Extended Functionality: j



Main Decoder table: 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**								

ELSEVIER

MICROARCHITECTURE

Main Decoder table: `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

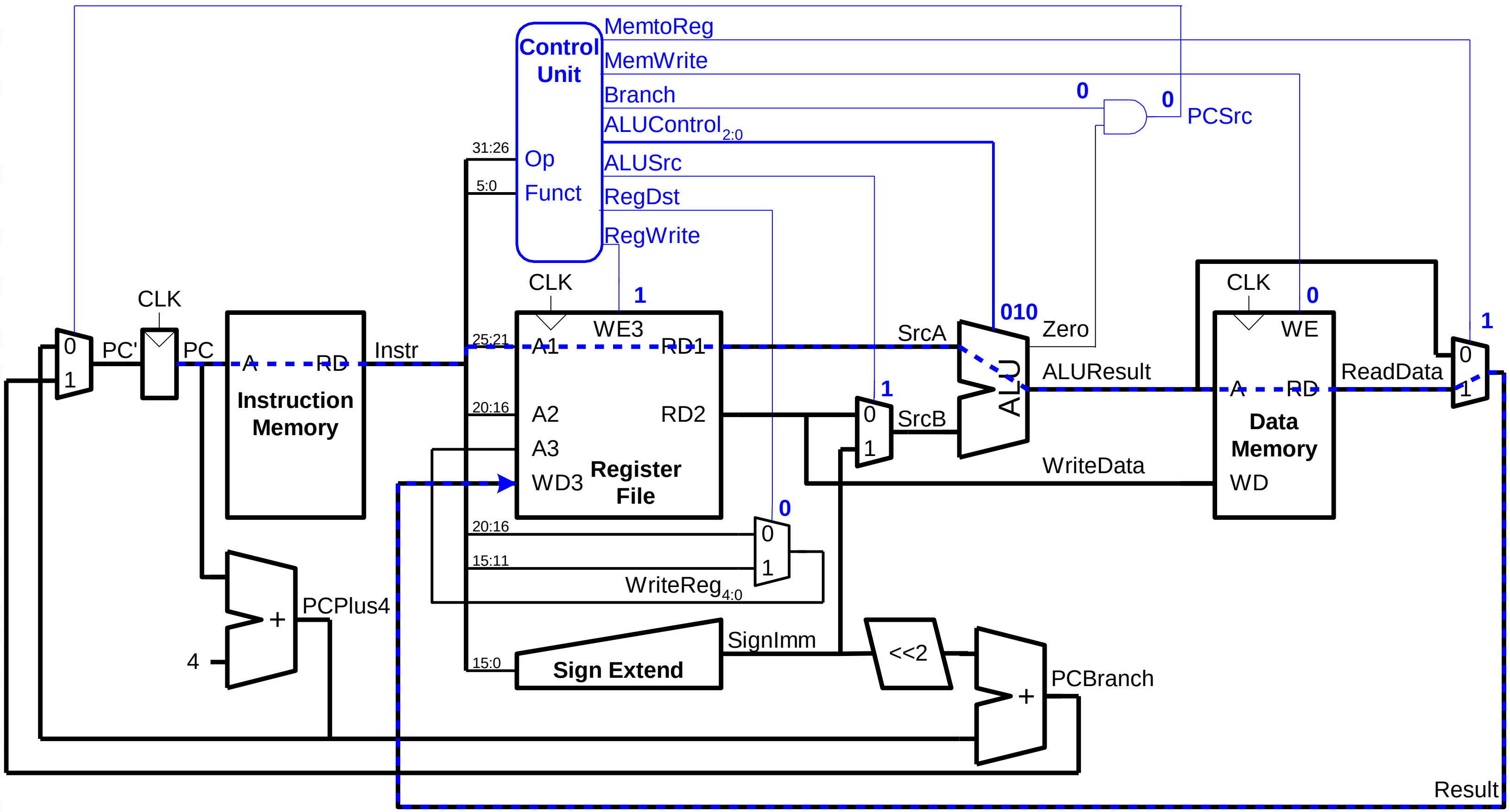
Review: Processor Performance

Program Execution Time

= (#instructions)(cycles/instruction)(seconds/cycle)

= IC x CPI x T_C

Single-Cycle Performance



T_C **limited by critical path (`lw`)**

MICROARCHITECTURE

ELSEVIER

Single-Cycle Performance

- Single-cycle critical path:

$$T_c = t_{pcq\_PC} + t_{\text{mem}} + \max(t_{RF\text{read}}, t_{sext} + t_{\text{mux}}) + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}}$$

- Typically, limiting paths are:
  - memory, ALU, register file
  - $T_c = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}}$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$$T_c = ?$$

ELSEVIER

MICROARCHITECTURE

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\text{read}}$	150
Register file setup	$t_{RF\text{setup}}$	20

$$T_c = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}}$$

$$= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps}$$

$$= 925 \text{ ps} \qquad [f_{\text{clk}} = 1/0.925 \text{ GHz} = 1.08 \text{ GHz}]$$

Single-Cycle Performance Example

Program with IC = 100 billion instructions:

Execution Time = IC x CPI x T_C

$= (100 \times 10^9)(1)(925 \times 10^{-12}\ \text{s})$

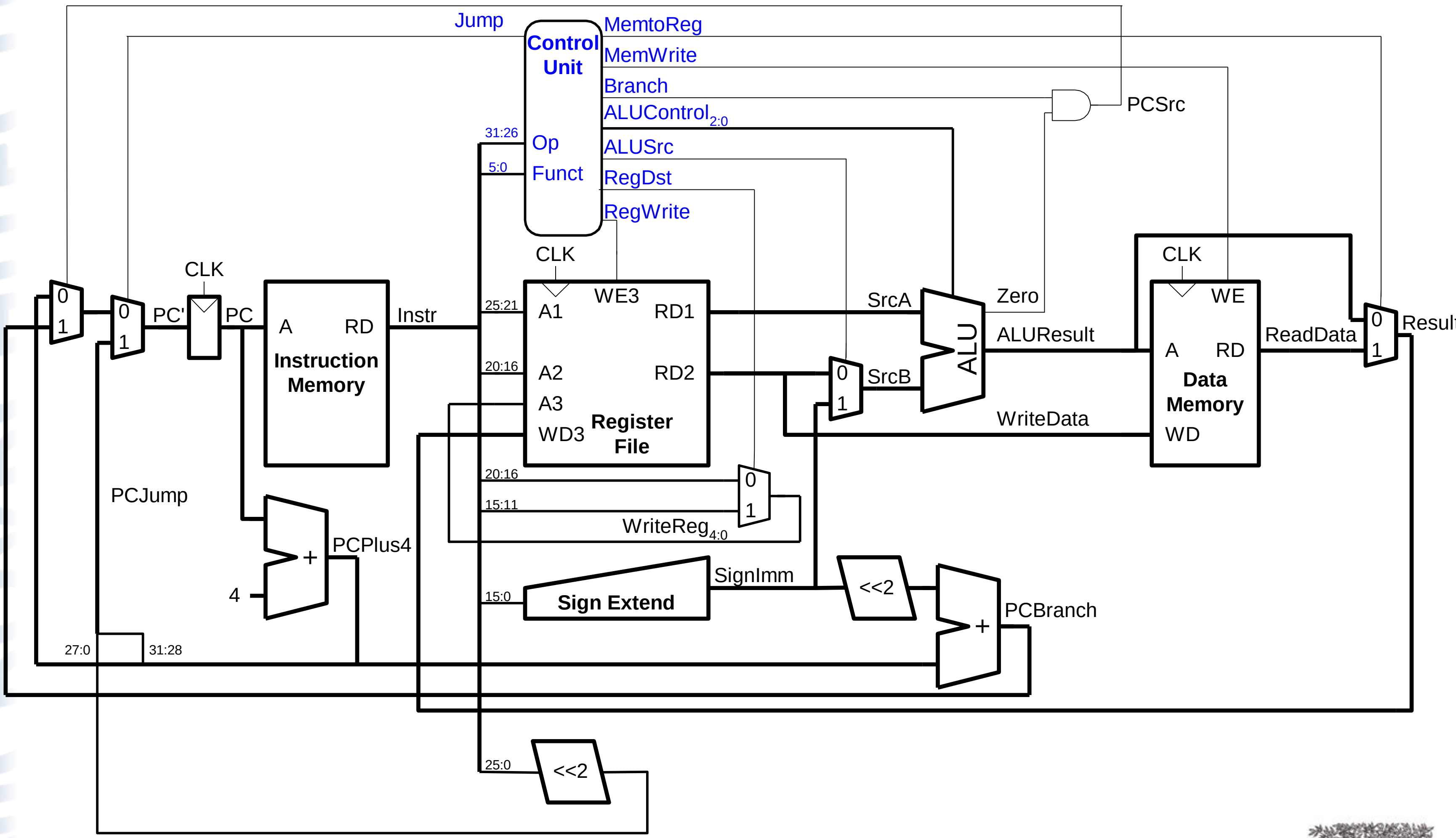
= **92.5 seconds**

Evaluation of Single-Cycle Processor

Pros and cons of single-cycle implementation:

+ simple design
+ 1 cycle per every instruction
- slow cycle time
  limited by longest instruction (`lw`)
- HW: 2 adders + ALU; 2 memories

Review: Single-Cycle Processor



System Verilog Model

```
//--------------------------------------------------
//  by David Harris, in Chapter 7 of DDCA 2nd ed. textbook
//  Top level system of Fig 7.59 including MIPS and memories
//--------------------------------------------------

module top (input   logic        clk, reset,
            output  logic[31:0] writedata, dataadr,
            output  logic        memwrite);

logic [31:0] pc, instr, readdata;

// instantiate processor and memories
mips mips (clk, reset, pc, instr, memwrite, dataadr, writedata, readdata);
imem imem (pc[7:2], instr);
dmem dmem (clk, memwrite, dataadr, writedata, readdata);

endmodule
```

MICROARCHITECTURE

ELSEVIER

Sys. Verilog Model of Data Memory

```
//---------------------------------------------
//  External data memory used by MIPS single-cycle processor
//---------------------------------------------
module dmem (input  logic            clk, we,
             input  logic[31:0]      a, wd,
             output logic[31:0]      rd);

logic  [31:0] RAM[63:0];

assign rd = RAM[a[31:2]];    // word-aligned  read

always_ff @(posedge clk)
   if (we)
     RAM[a[31:2]] <= wd;      // word-aligned write
endmodule
```

Sys. Verilog Model of Instr. Memory

```
module imem (input logic[5:0] addr, output reg [31:0] instr);

// imem is modeled as a lookup table, a stored-program byte-addressable ROM
        always_comb
          case ({addr,2'b00})
//                address          instruction
//                ---------        --------------
                  8'h00: instr = 32'h20020005;
                  8'h04: instr = 32'h2003000c;
                  8'h08: instr = 32'h2067fff7;
                  8'h0c: instr = 32'h00e22025;
                  8'h10: instr = 32'h00642824;
                  8'h14: instr = 32'h00a42820;
                  8'h18: instr = 32'h10a7000a;
                  8'h1c: instr = 32'h0064202a;
                  8'h20: instr = 32'h10800001;
                default: instr = {32{1'bx}};     // unknown instruction
          endcase
endmodule
```

MICROARCHITECTURE

ELSEVIER

Alternate Model of Instr. Memory

```
module imem (input    logic[5:0] addr,
                      output logic[31:0] instr);

logic  [31:0] RAM[63:0];

// imem is RAM, loaded from memfile.dat file with hex values at startup
  initial
    begin
      $readmemh("memfile.dat", RAM);
    end

assign instr = RAM[addr];   // instr at RAM[addr] is read out

endmodule

// imem can be created with CoreGen for Xilinx synthesis
```

Sys. Verilog Model of MIPS processor

```
// single-cycle MIPS processor
module mips (input  logic           clk, reset,
             output logic[31:0]  pc,
             input  logic[31:0]   instr,
             output logic          memwrite,
             output logic[31:0]  aluout, writedata,
             input  logic[31:0]   readdata);

  logic        memtoreg, pcsrc, zero, alusrc, regdst, regwrite, jump;
  logic [2:0]  alucontrol;

  controller c (instr[31:26], instr[5:0], zero, memtoreg, memwrite, pcsrc,
                alusrc, regdst, regwrite, jump, alucontrol);

  datapath dp (clk, reset, memtoreg, pcsrc, alusrc, regdst, regwrite, jump,
               alucontrol, zero, pc, instr, aluout, writedata, readdata);

endmodule
```

Sys. Verilog Model of Controller

```
module controller ( input  logic[5:0] op, funct,
                    input   logic       zero,
                    output logic        memtoreg, memwrite,
                    output logic        pcsrc, alusrc,
                    output  logic       regdst, regwrite,
                    output  logic       jump,
                    output logic[2:0] alucontrol);

  logic [1:0] aluop;
  logic        branch;

  maindec md (op, memtoreg, memwrite, branch, alusrc, regdst, regwrite,
                      jump, aluop);

  aludec  ad (funct, aluop, alucontrol);

  assign pcsrc = branch & zero;

endmodule
```


ELSEVIER

Sys. Verilog Model of Main Decoder

```
module maindec (input  logic[5:0] op,
                                output logic memtoreg, memwrite, branch,
                                output logic alusrc, regdst, regwrite, jump,
                                output logic[1:0] aluop );
  logic [8:0] controls;

  assign {regwrite, regdst, alusrc, branch, memwrite,
          memtoreg,  aluop, jump} = controls;

 always_comb
   case(op)
     6'b000000: controls  = 9'b110000010; //Rtype
     6'b100011: controls  = 9'b101001000; //LW
     6'b101011: controls  = 9'b001010000; //SW
     6'b000100: controls  = 9'b000100001; //BEQ
     6'b001000: controls  = 9'b101000000; //ADDI
     6'b000010: controls  = 9'b000000100; //J
     default:      controls  = 9'bxxxxxxxxx;    //illegal op
   endcase
endmodule
```

Sys. Verilog Model of ALU Decoder

```
module aludec (input    logic[5:0] funct,
               input    logic[1:0] aluop,
               output logic [2:0] alucontrol);
  always_comb
    case(aluop)
      2'b00: alucontrol  = 3'b010;  // add  (for lw/sw/addi)
      2'b01: alucontrol  = 3'b110;  // sub   (for beq)
      default: case(funct)          // R-TYPE instructions
          6'b100000: alucontrol  = 3'b010; // ADD
          6'b100010: alucontrol  = 3'b110; // SUB
          6'b100100: alucontrol  = 3'b000; // AND
          6'b100101: alucontrol  = 3'b001; // OR
          6'b101010: alucontrol  = 3'b111; // SLT
          default:      alucontrol  = 3'bxxx; // ???
        endcase
    endcase
endmodule
```

Sys. Verilog Model of Datapath

```
module datapath (input  logic clk, reset, memtoreg, pcsrc, alusrc, regdst,
                 input logic regwrite, jump, input  logic[2:0]  alucontrol,
                 output  logic zero, output logic[31:0] pc,
                input  logic[31:0] instr,
                 output logic[31:0] aluout, writedata,
                input  logic[31:0] readdata);
  logic [4:0]  writereg;
  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  logic [31:0] signimm, signimmsh, srca, srcb, result;

  // next PC logic
  flopr #(32) pcreg(clk, reset, pcnext, pc);
  adder       pcadd1(pc, 32'b100, pcplus4);
  sl2         immsh(signimm, signimmsh);
  adder       pcadd2(pcplus4, signimmsh, pcbranch);
  mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc,
                      pcnextbr);
  mux2 #(32)  pcmux(pcnextbr, {pcplus4[31:28],
                    instr[25:0], 2'b00}, jump, pcnext);
```

Sys. Verilog Model of Datapath

```
// register file logic
  regfile     rf (clk, regwrite, instr[25:21], instr[20:16], writereg,
                  result, srca, writedata);

  mux2 #(5)   wrmux (instr[20:16], instr[15:11], regdst, writereg);
  mux2 #(32)  resmux (aluout, readdata, memtoreg, result);
  signext     se (instr[15:0], signimm);

 // ALU logic
  mux2 #(32)  srcbmux (writedata, signimm, alusrc, srcb);
  alu         alu (srca, srcb, alucontrol, aluout, zero);

endmodule
```

Sys. Verilog Model of Register File

```
module regfile (input    logic clk, we3,
                input    logic[4:0]  ra1, ra2, wa3,
                input    logic[31:0] wd3,
                output logic[31:0] rd1, rd2);

  logic [31:0] rf [31:0];

  // three ported register file: read two ports combinationally
  // write third port on rising edge of clock. Register0 hardwired to 0.

  always_ff
    if (we3)
       rf [wa3] <= wd3;

  assign rd1 = (ra1 != 0) ? rf [ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ ra2] : 0;

endmodule
```

Sys. Verilog Models of Other Parts

```
module adder (input    logic[31:0] a, b,
                       output  logic[31:0] y);
   assign y = a + b;
endmodule

module sl2 (input   logic[31:0] a,
                  output logic[31:0] y);
// shift left by 2
    assign y = {a[29:0], 2'b00};
endmodule

module signext (input    logic[15:0] a,
                         output logic[31:0] y);

  assign y = {{16{a[15]}}, a};
endmodule
```

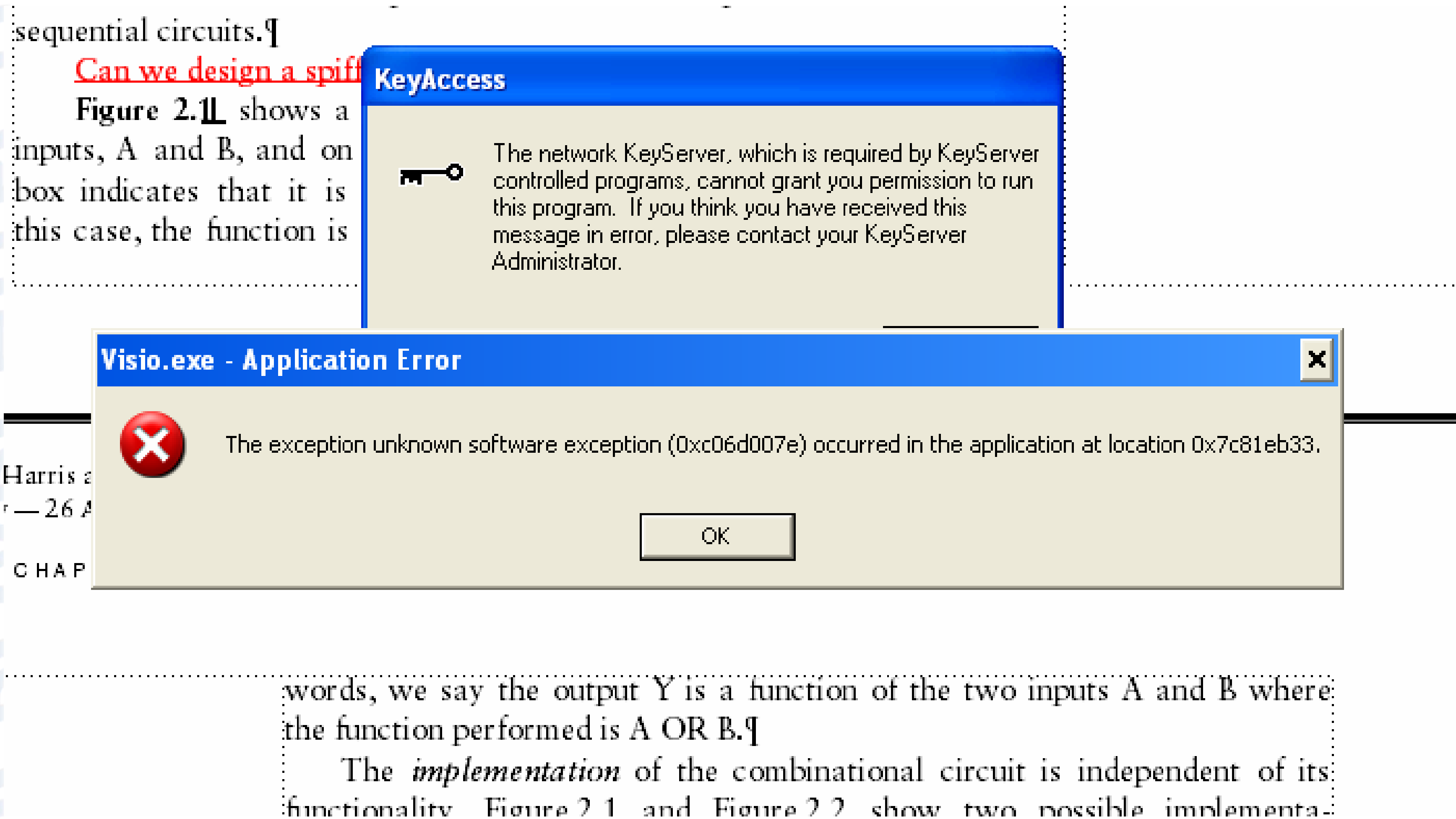
Sys. Verilog for Parameterized Parts

```
module flopr #(parameter WIDTH = 8)
                           (input logic clk, reset,
                            input logic[WIDTH-1:0] d,
                            output logic[WIDTH-1:0] q);

  always_ff@(posedge clk, posedge reset)
    if (reset) q <= 0;
    else         q <= d;
endmodule


module mux2 #(parameter WIDTH = 8)
              (input   logic[WIDTH-1:0] d0, d1,
               input   logic s,
               output logic[WIDTH-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

Review: Exceptions

- Unscheduled function call to *exception handler*
- Caused by:
  - Hardware, also called an *interrupt*, e.g. keyboard
  - Software, also called *traps*, e.g. undefined instruction
- When exception occurs, the processor:
  - Records cause of exception (`Cause` register)
  - Jumps to exception handler (0x80000180)
  - Returns to program (`EPC` register)

Example Exception



ELSEVIER

Review: Exception Registers

- Not part of register file; in Coprocessor 0
  - `Cause`
    - Records cause of exception
    - Coprocessor 0 register 13
  - `EPC` (Exception PC)
    - Records PC where exception occurred
    - Coprocessor 0 register 14
- Move from Coprocessor 0
  - `mfc0 $t0, Cause (=mfc0 $t0,$13)`
  - Moves contents of `Cause` into `$t0`

mfc0

010000	00000	`$t0` (8)	`Cause` (13)	00000000000
31:26	25:21	20:16	15:11	10:0

Review: Exception Causes

Exception	Cause
Hardware Interrupt	0x00000000
System Call	0x00000020
Breakpoint / Divide by 0	0x00000024
Undefined Instruction	0x00000028
Arithmetic Overflow	0x00000030

Extend single-cycle MIPS processor to handle last two types of exceptions

Exception RTLs

Undefined Instruction

IM[PC]
. . . # problem in decoding (bad `op` or `func`)
Cause ← 40 # = 0x28
EPC ← PC
PC ← 0x80000180 #Exception handler address

Arithmetic Overflow

IM[PC]
. . . # ALU operation overflows
Cause ← 48 # = 0x30
EPC ← PC
PC ← 0x80000180 #Exception handler address

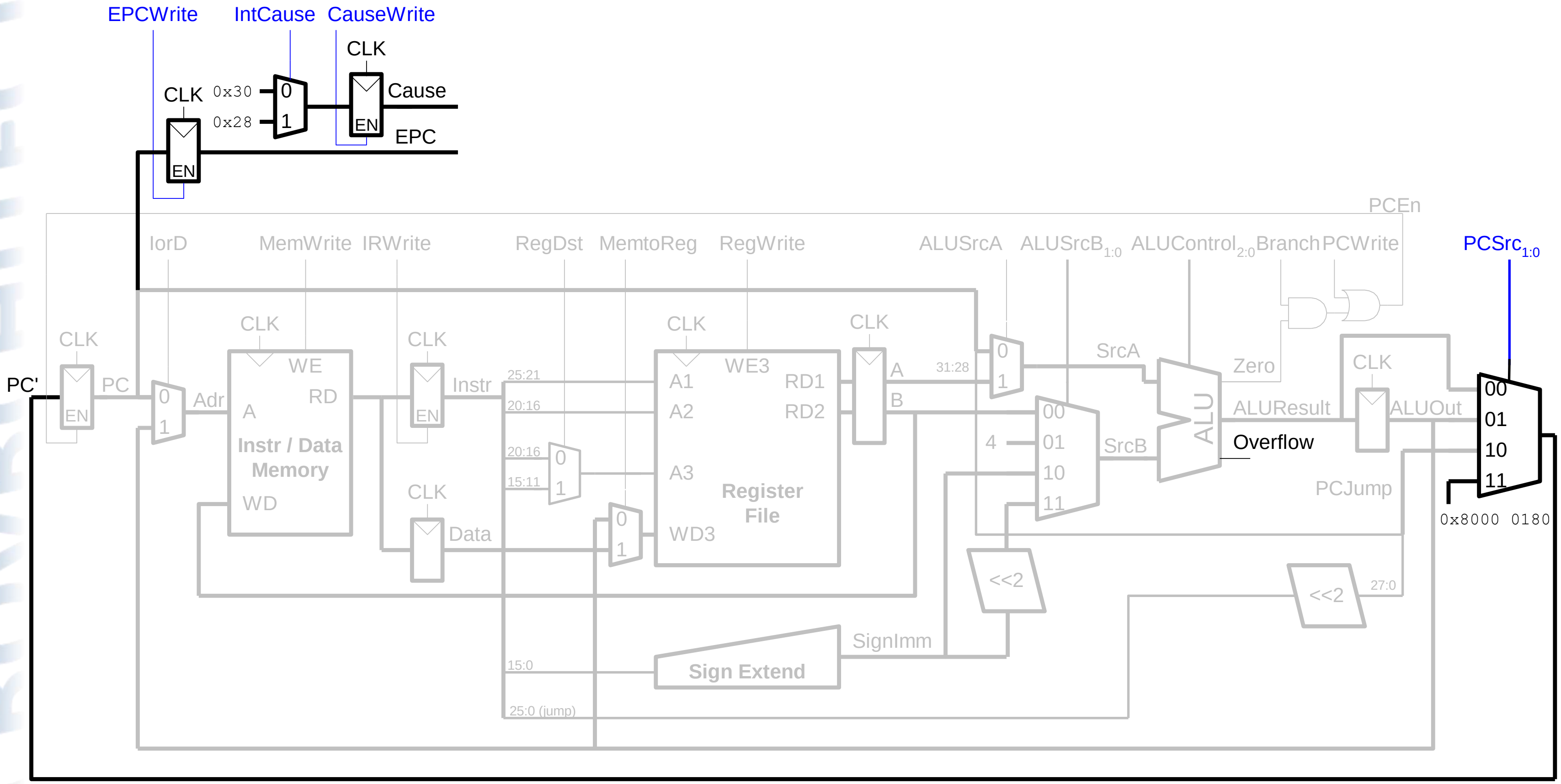
mfc0 instruction (e.g. `mfc0 $t1, $13`)

IM[PC]
RF[rt] ←RFc0[rd]
PC ← PC + 4

MICROARCHITECTURE

ELSEVIER

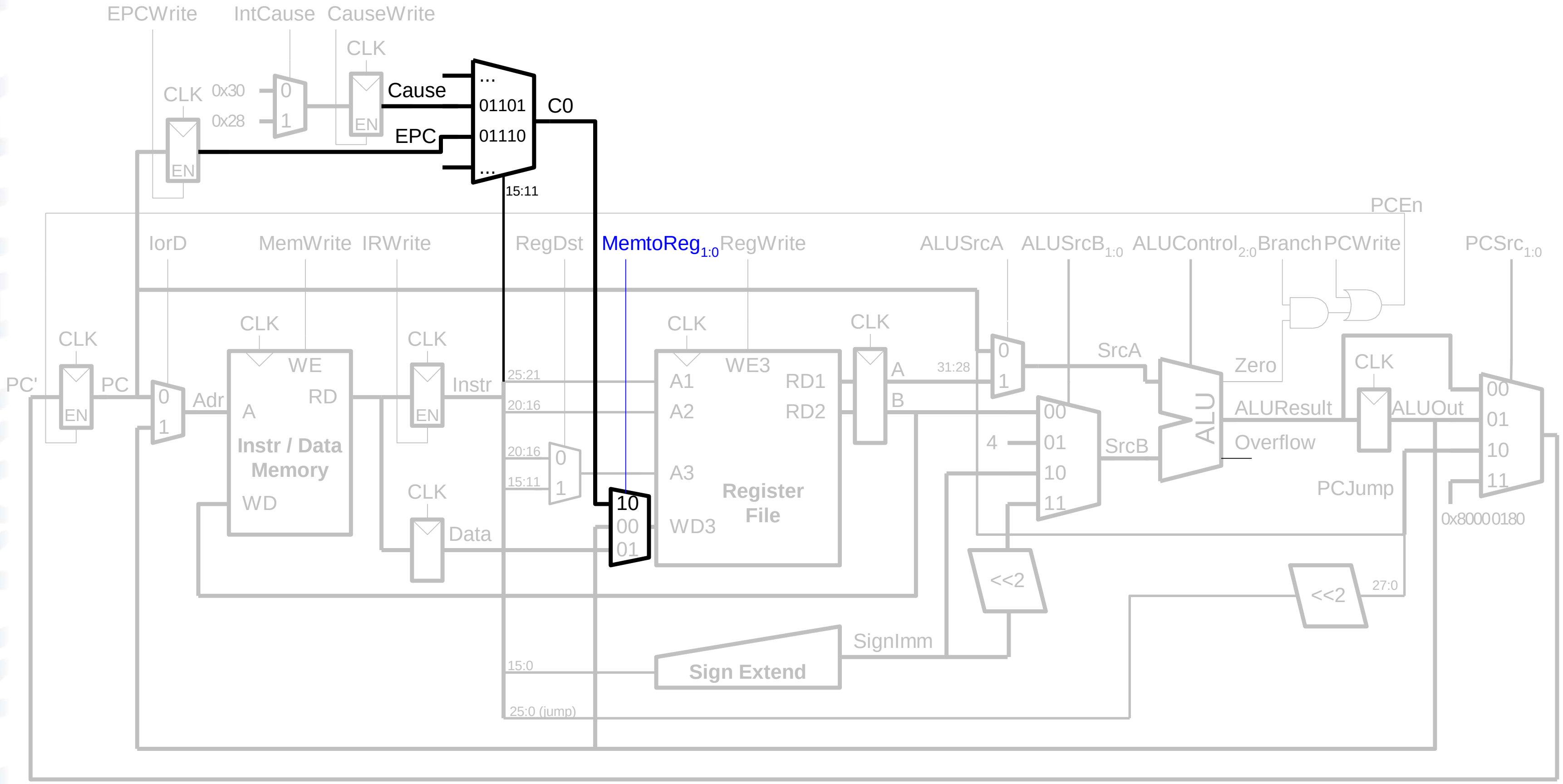
Exception Hardware: EPC & Cause



Never mind the *multi-cycle* datapath, focus on the exception hardware.

ELSEVIER

Exception Hardware: mfc0

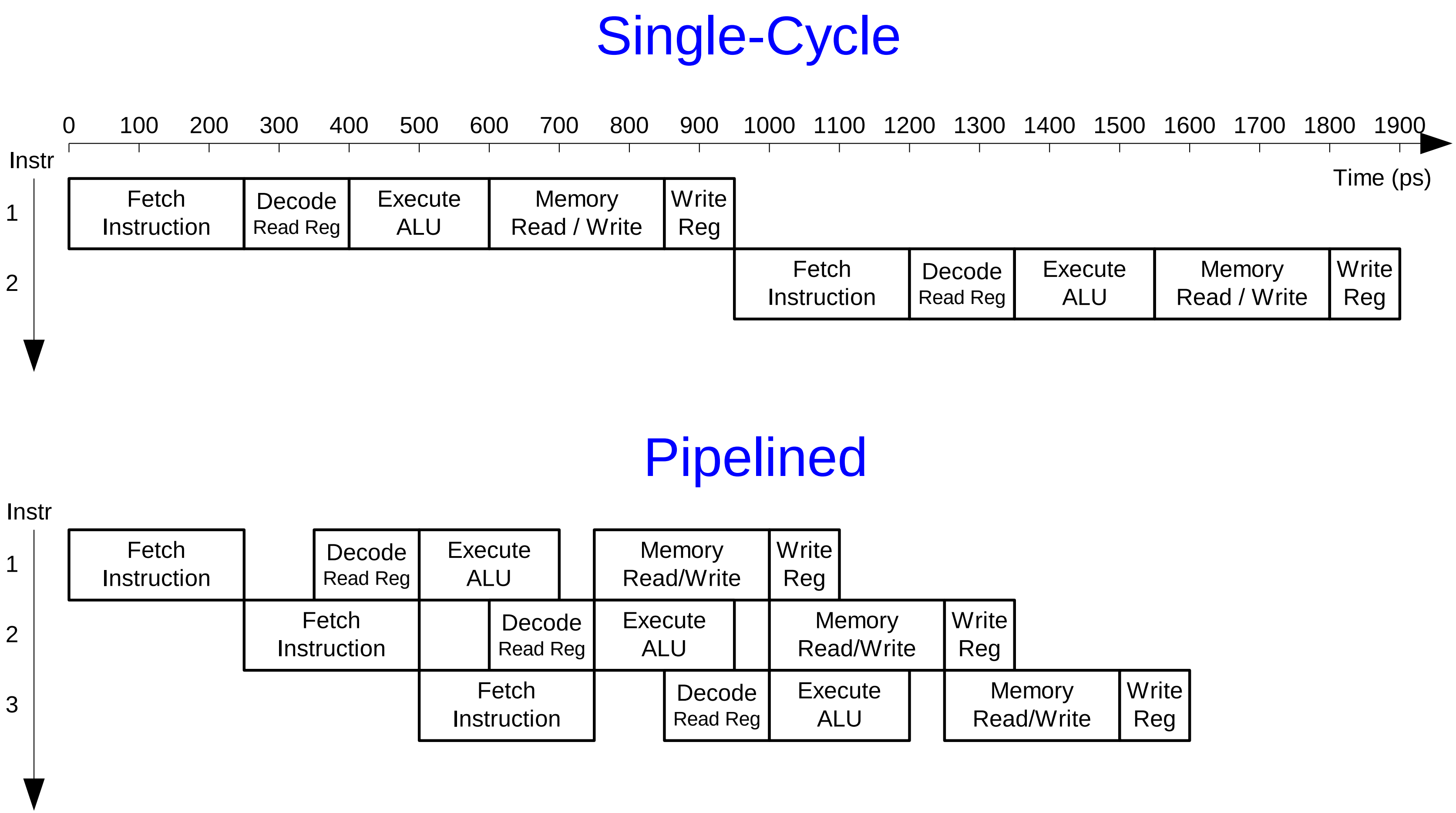


Never mind the *multi-cycle* datapath, focus on the exception hardware.

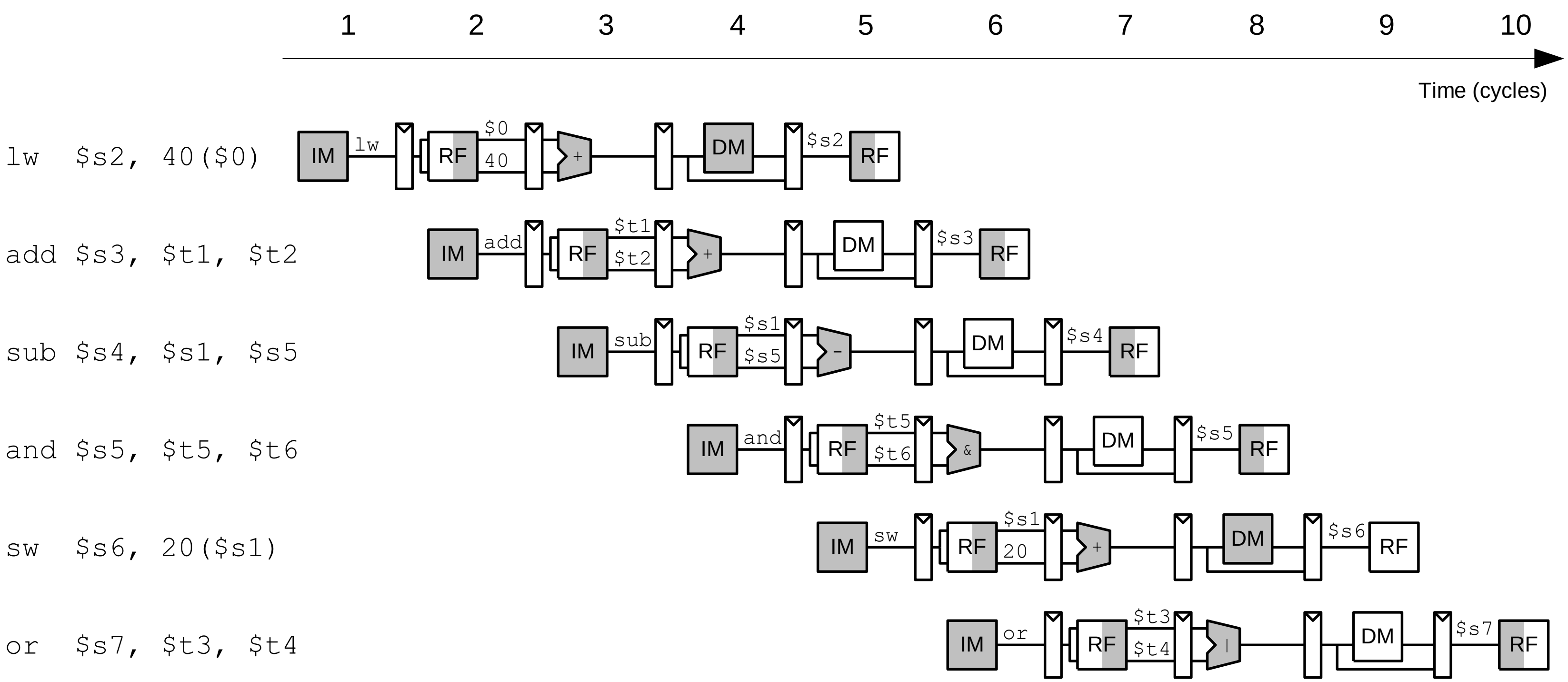
Pipelined MIPS Processor

- Temporal parallelism
- Divide single-cycle processor into 5 stages:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add pipeline registers between stages

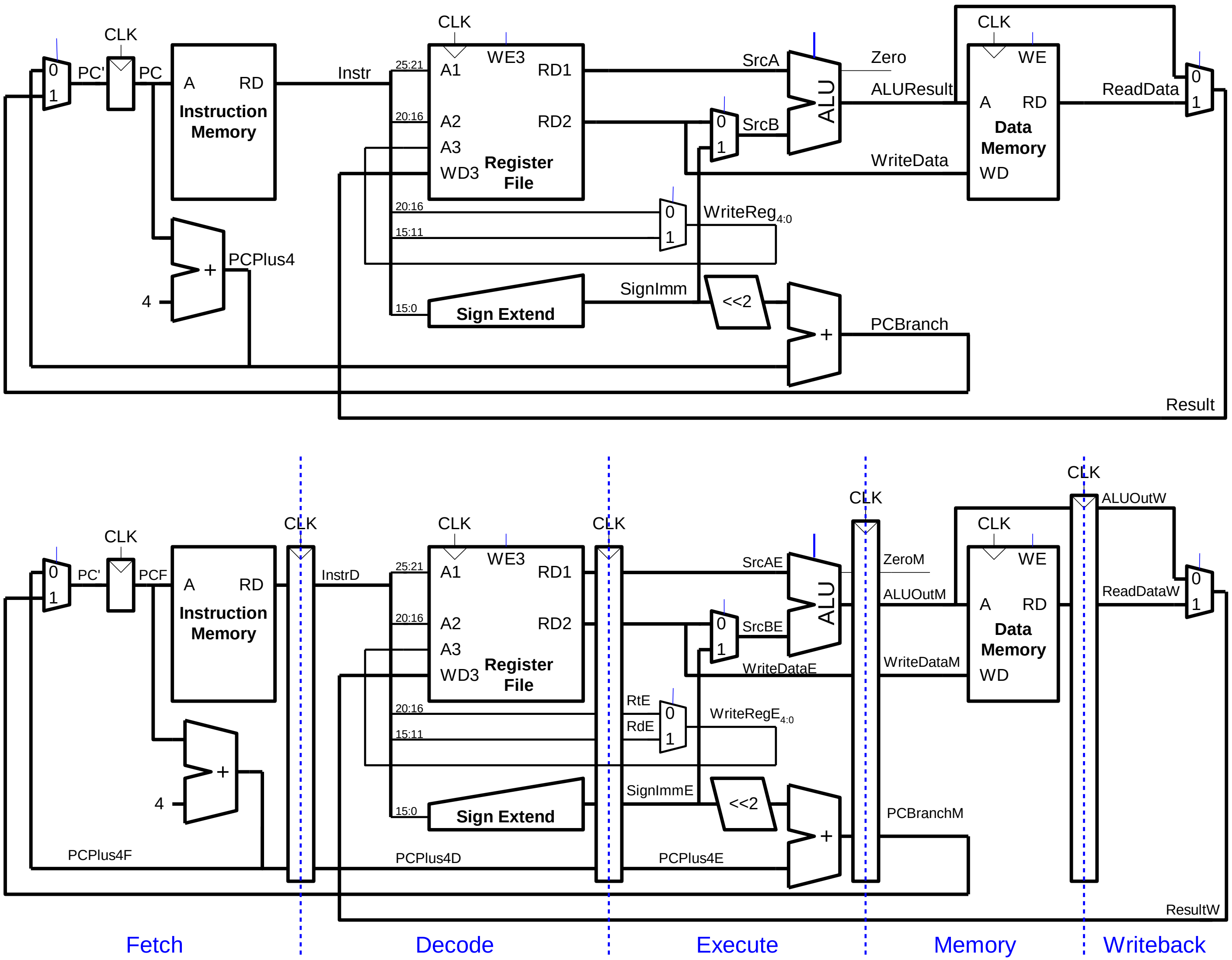
Single-Cycle vs. Pipelined



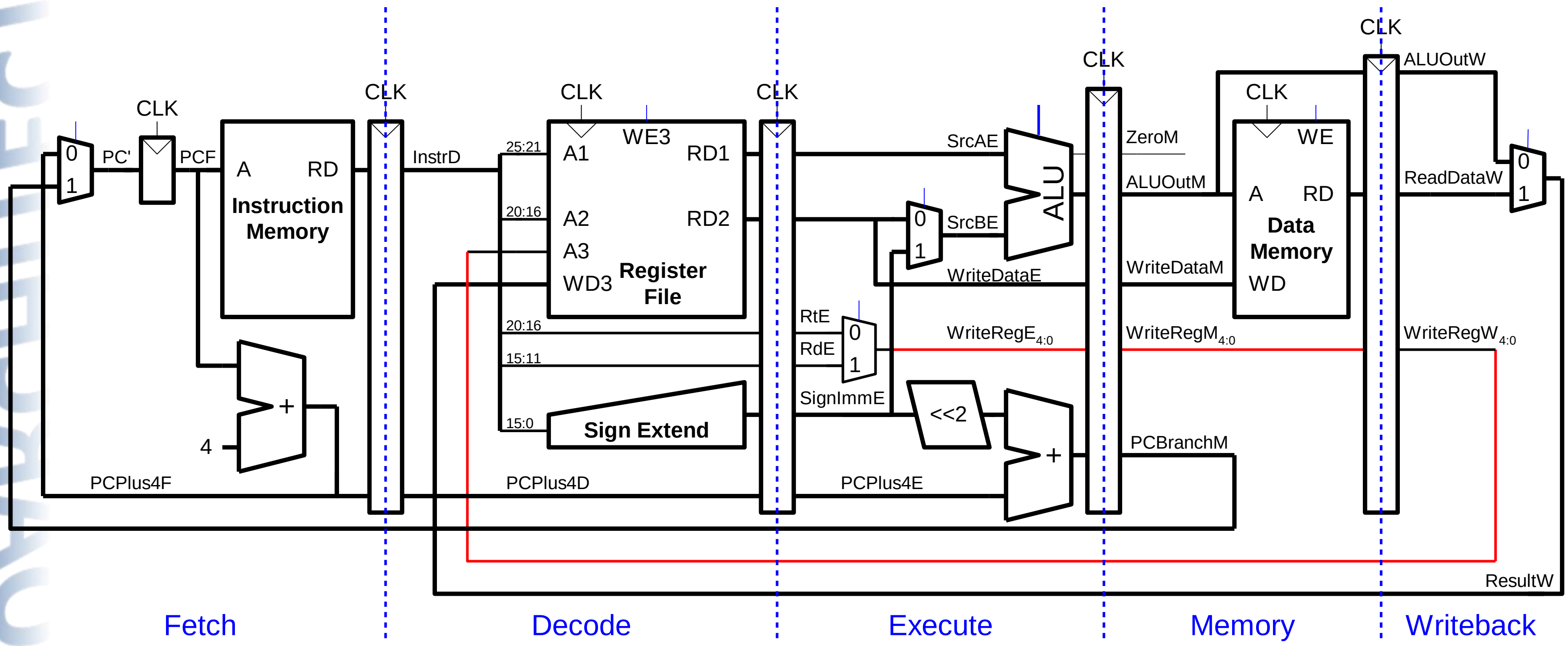
Pipelined Processor Abstraction



Single-Cycle & Pipelined Datapath

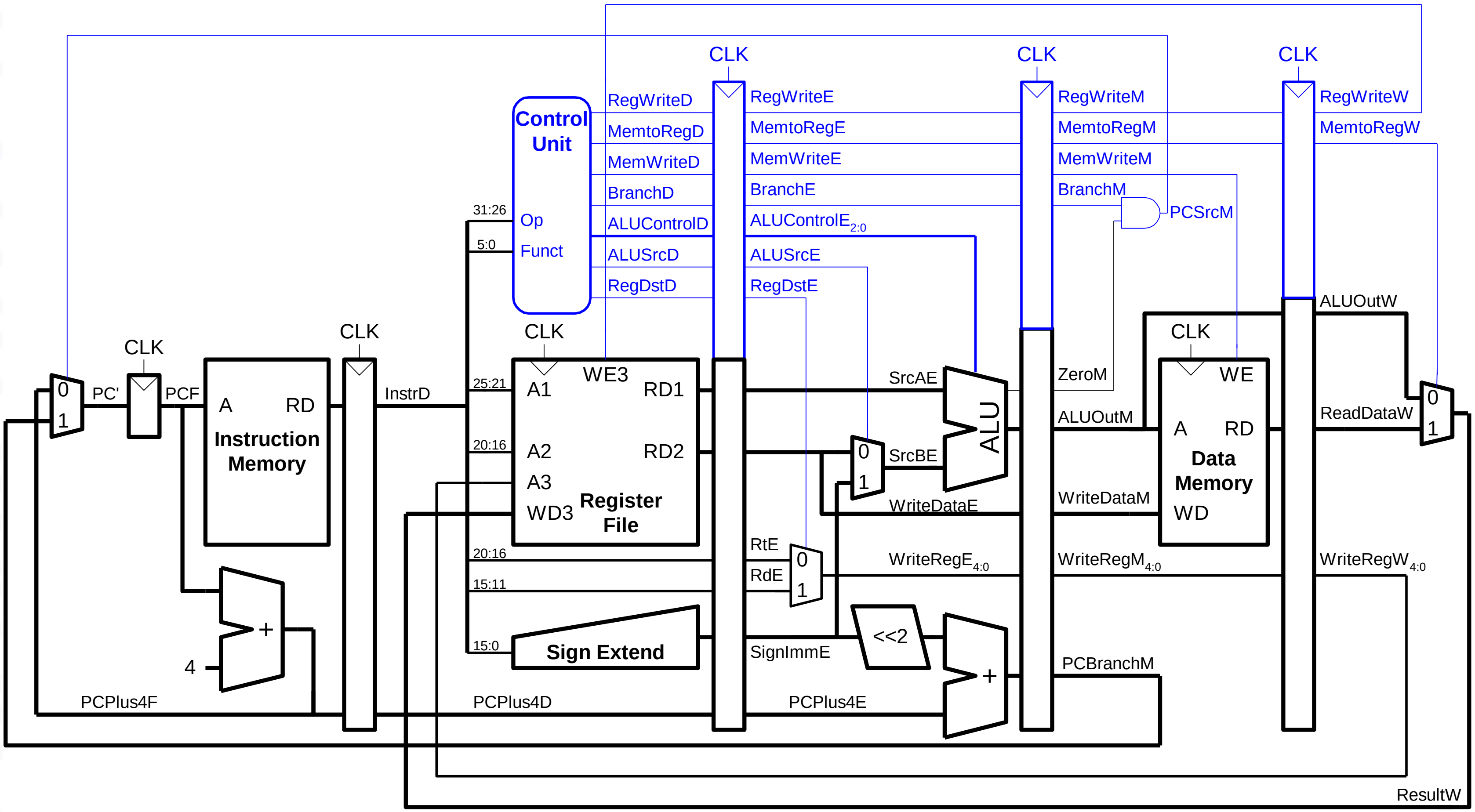


Corrected Pipelined Datapath



WriteReg **must arrive at same time as** ***Result***

Pipelined Processor Control

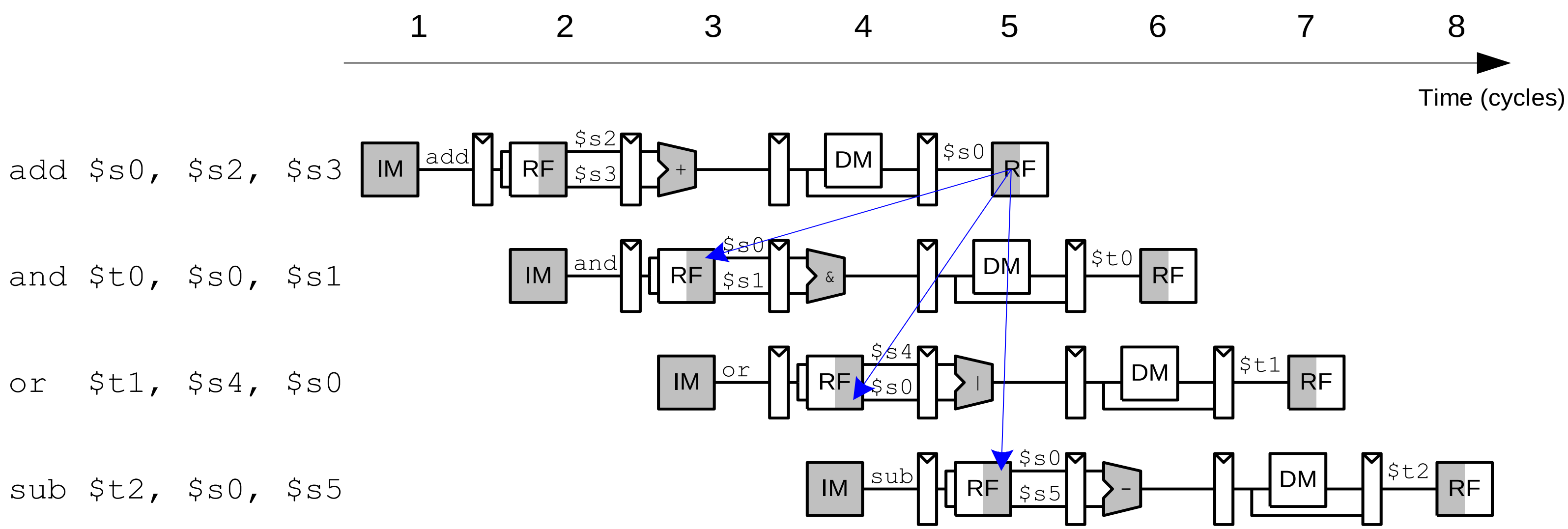


- **Same control unit as single-cycle processor**
- **Control delayed to proper pipeline stage**

Pipeline Hazards

- When an instruction depends on result from instruction that hasn't completed
- Types:
  - **Data hazard:** register value not yet written back to register file
  - **Control hazard:** next instruction not decided yet (caused by branches) or target address not calculated yet (jumps and branches)

Data Hazard



1
2
3
4
5
6
7
8
Time (cycles)
add $s0, $s2, $s3
and $t0, $s0, $s1
or $t1, $s4, $s0
sub $t2, $s0, $s5
IM
add
RF
$s2
$s3
+
DM
$s0
RF
IM
and
RF
$s0
$s1
&
DM
$t0
RF
IM
or
RF
$s4
$s0
|
DM
$t1
RF
IM
sub
RF
$s0
$s5
-
DM
$t2
RF

Handling Data Hazards

<u>2 SW fixes</u>

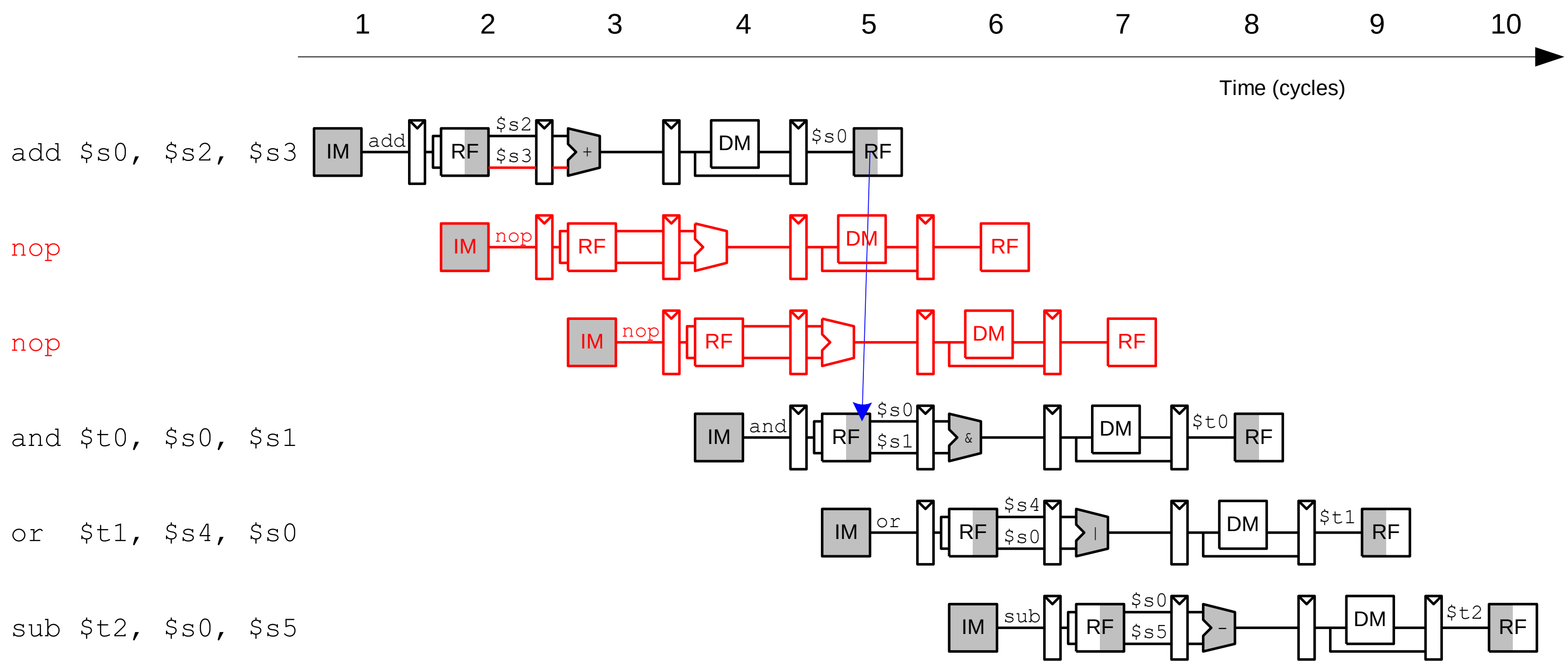
- Insert `nop`s in code at compile time
- Rearrange code at compile time

<u>2 HW fixes</u>

- Stall the processor at run time
- Forward data at run time

Compile-Time Hazard Elimination

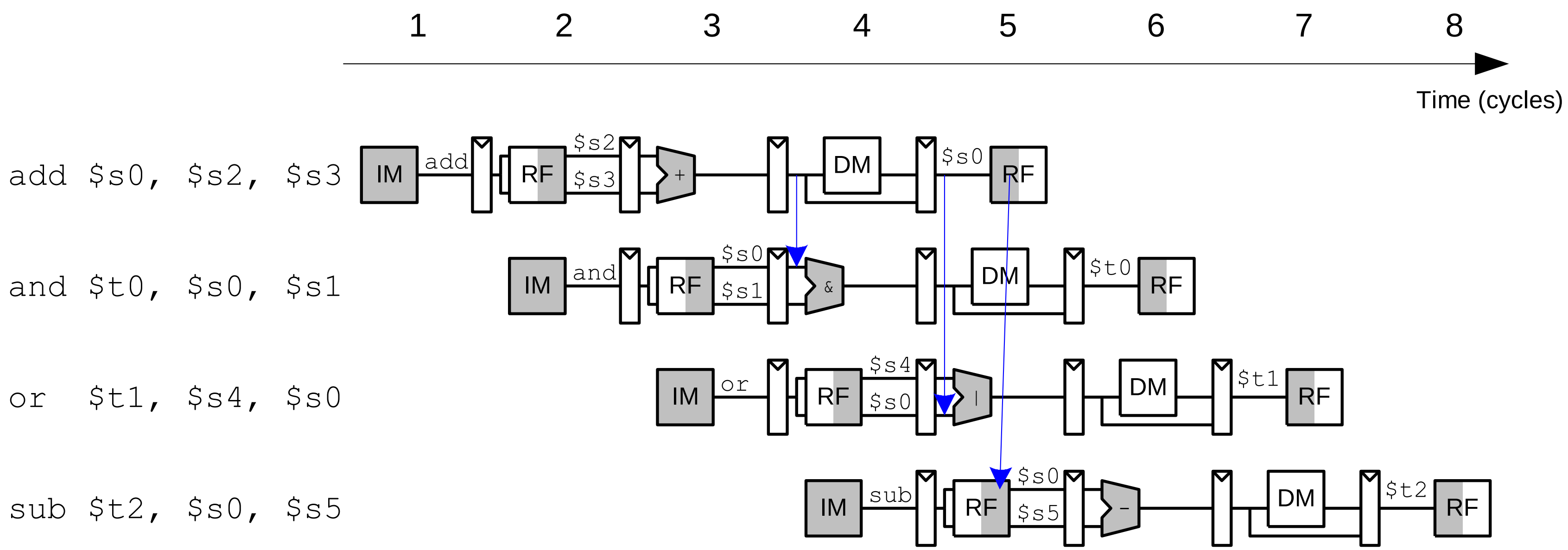
- Insert enough `nop`s for result to be ready
- Or move independent useful instructions forward



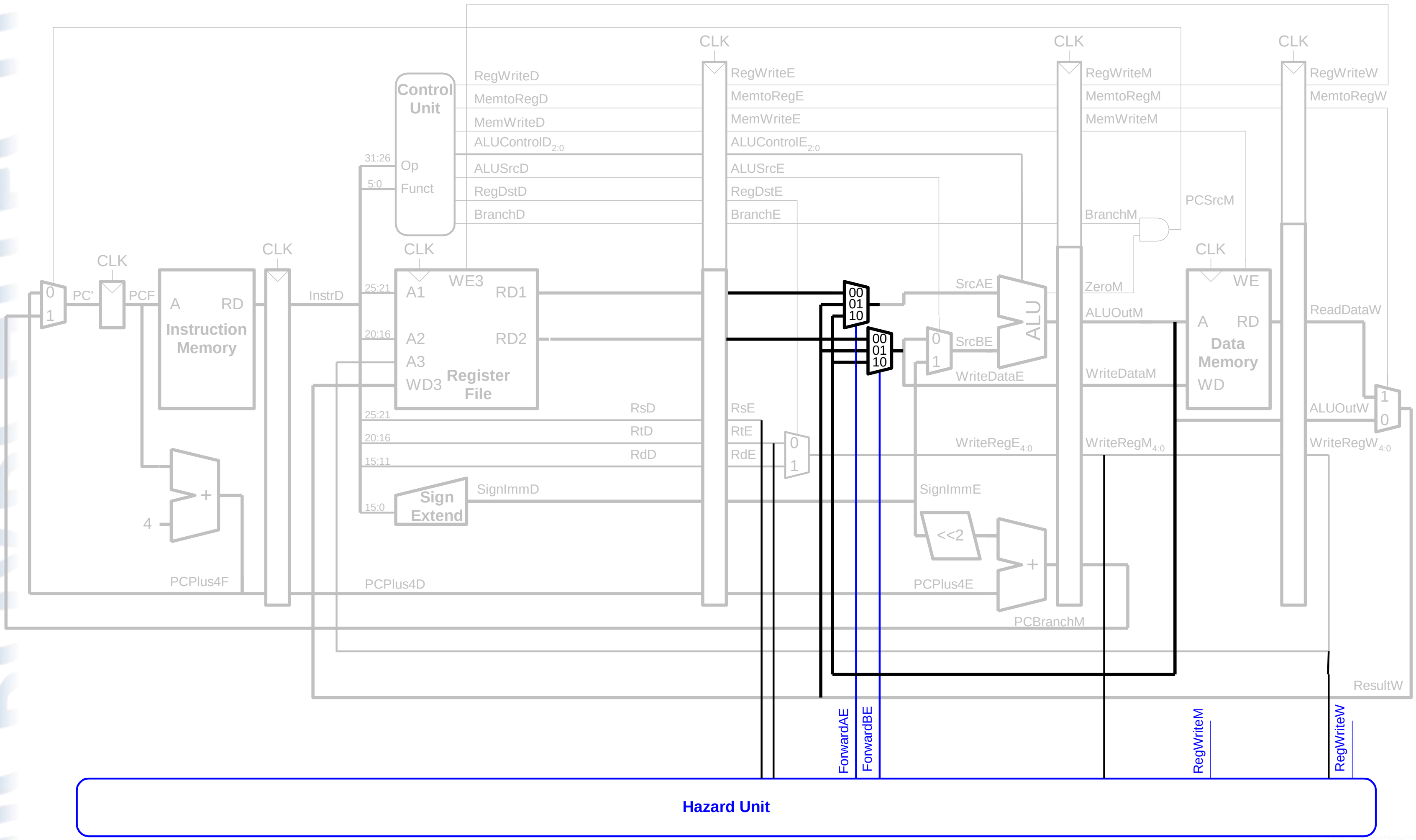
ELSEVIER

MICROARCHITECTURE

Data Forwarding



Data Forwarding



Data Forwarding

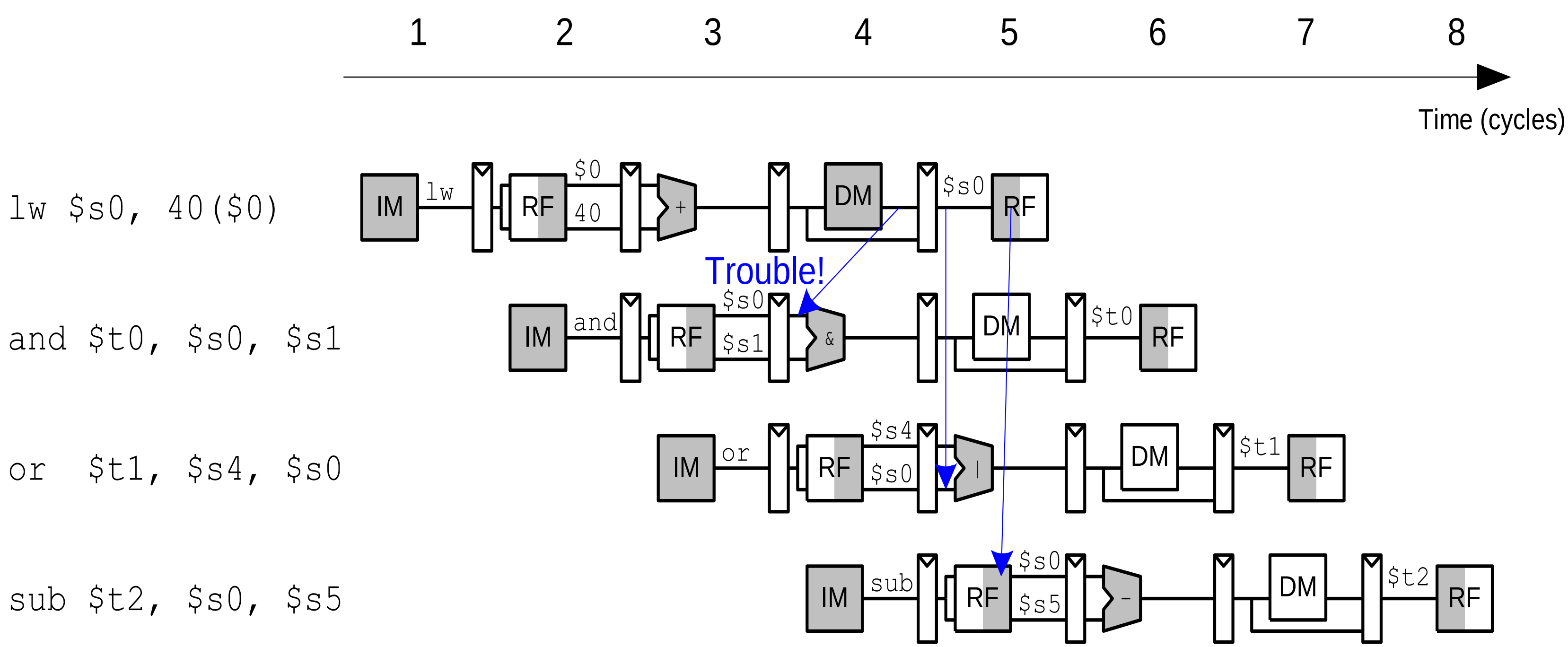
- Forward to Execute stage from either:
  - Memory stage or
  - Writeback stage
- Forwarding logic for *ForwardAE*:

```
if  ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)
        then   ForwardAE = 10
else
     if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW)
        then   ForwardAE = 01
      else     ForwardAE = 00
```

Forwarding logic for *ForwardBE* same, but replace *rsE* with *rtE*

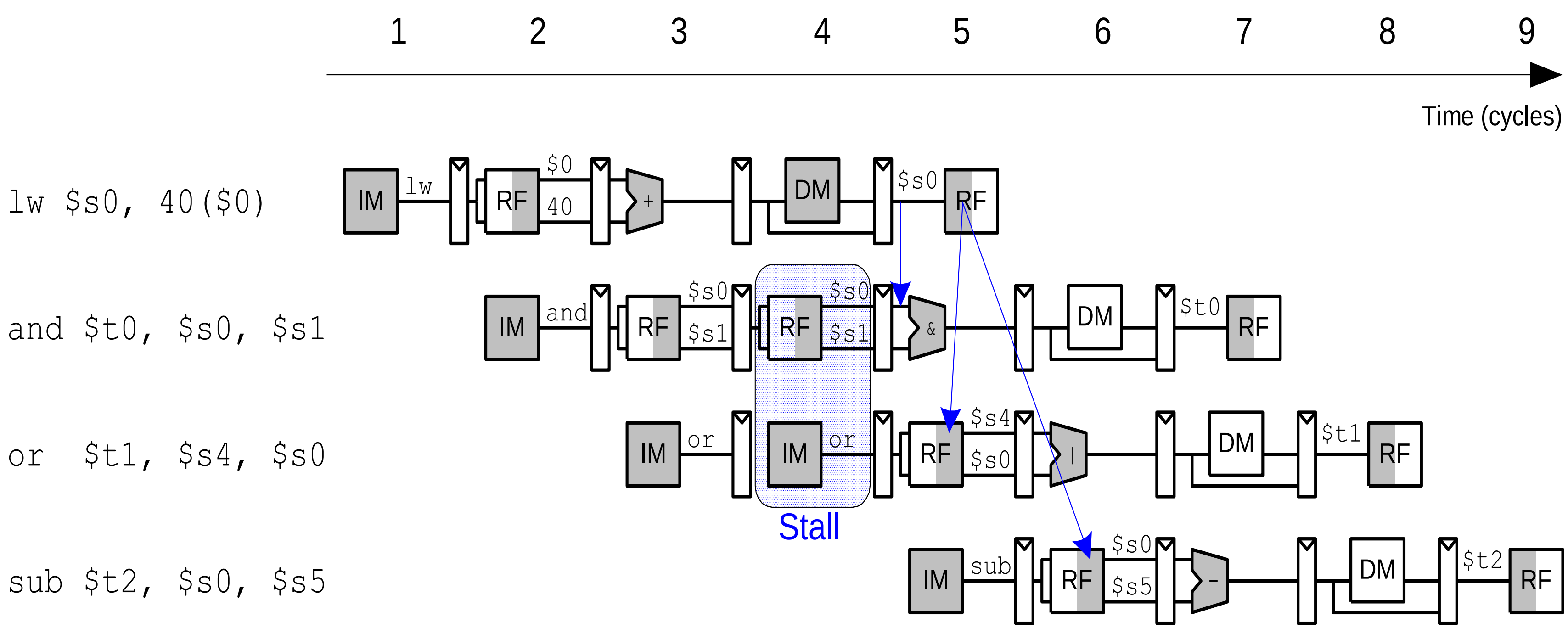
Stalling

Forwarding on a load-use hazard isn't possible!

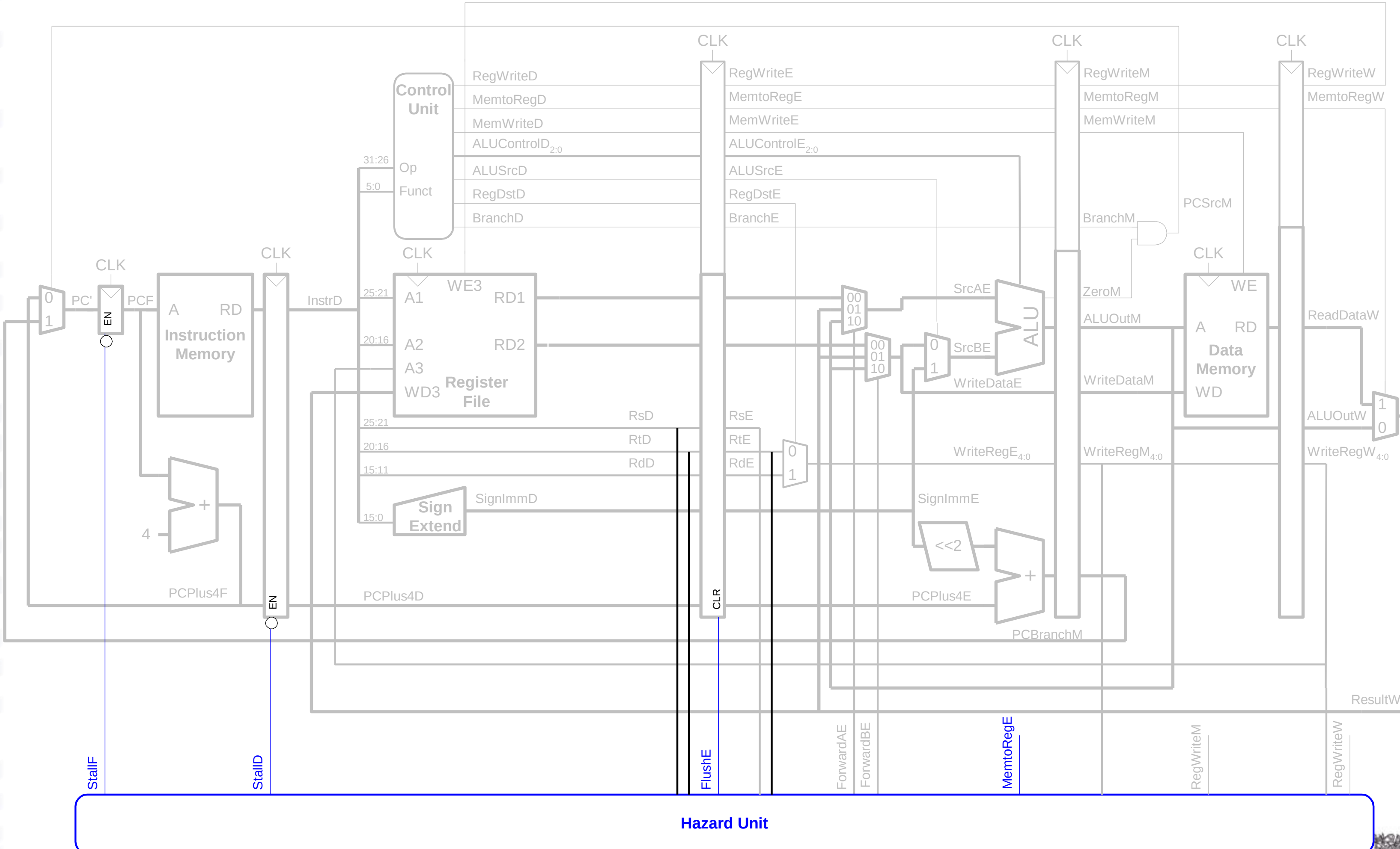


Stalling

The HW solution is to stall the pipeline



Stalling Hardware



Stalling Logic

lwstall =
  ((*rsD*==*rtE*) OR (*rtD*==*rtE*)) AND *MemtoRegE*

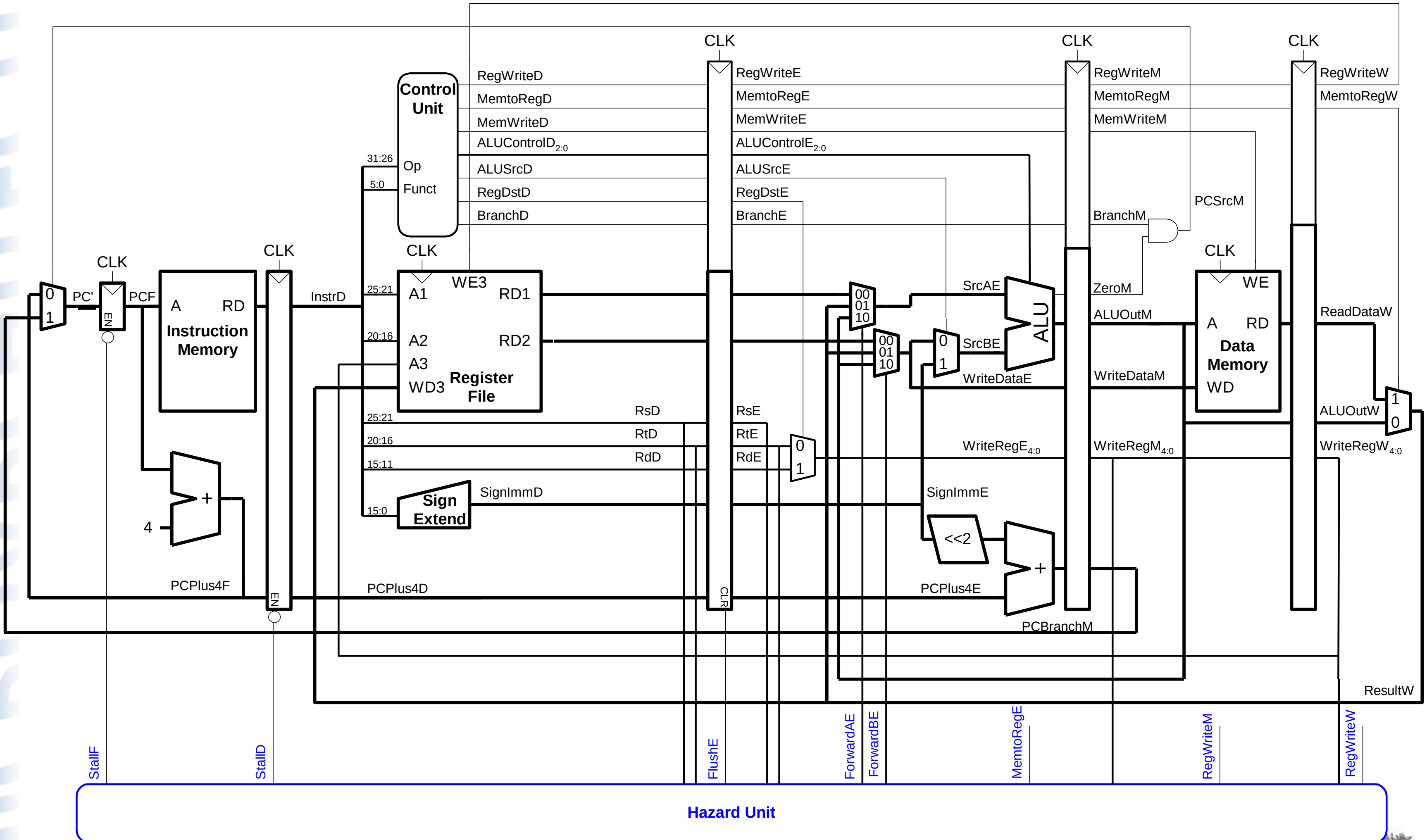
StallF = *StallD* = *FlushE* = *lwstall*

- By flushing the Execute stage, and stalling Fetch and Decode stages, the instruction flushed will simply be repeated in then next clock cycle, but this time with correct (forwarded) data!

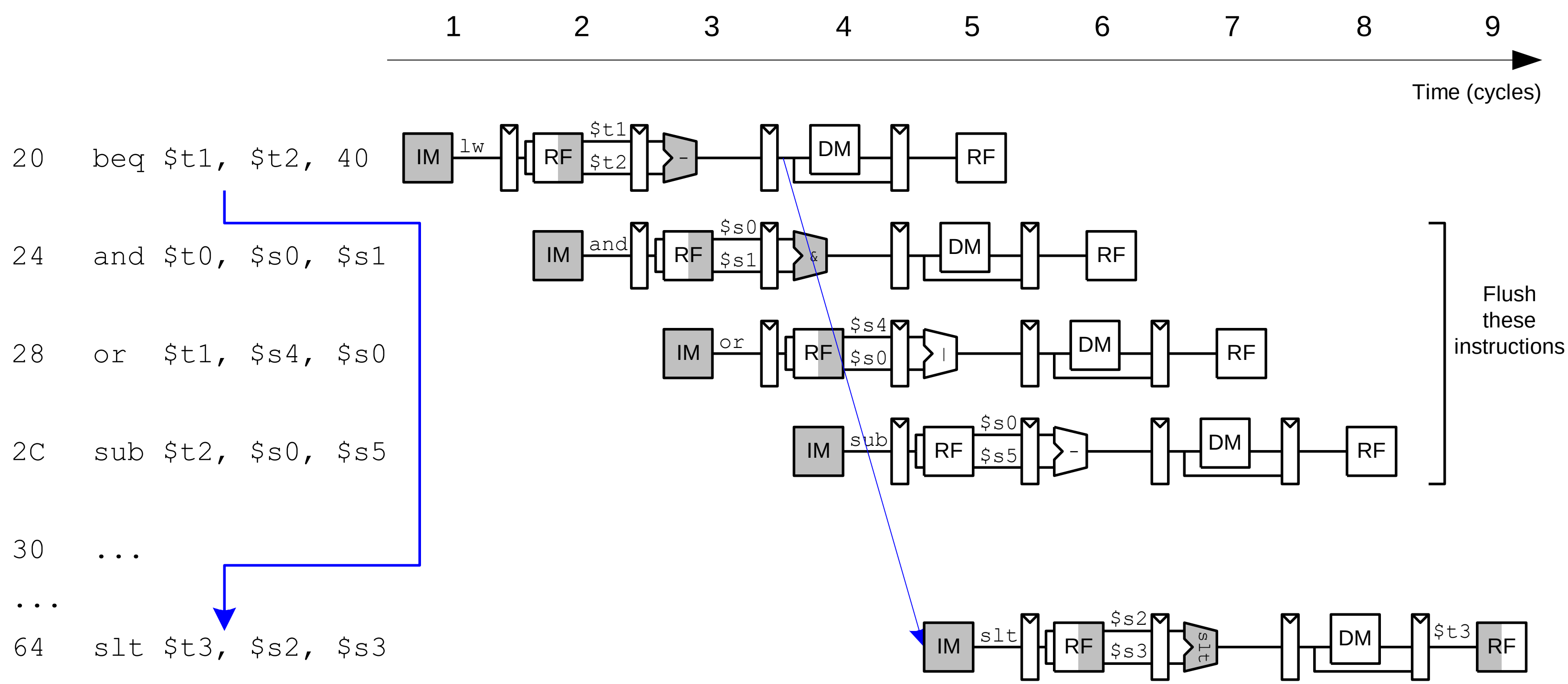
Control Hazards

- **`beq`:**
  - branch not determined until 4th stage of pipeline
  - Instructions after the branch are fetched *before* the branch occurs
  - These instructions must be flushed if branch happens
- **Branch misprediction penalty**
  - the # of instruction flushed, when branch is taken
  - may be reduced by determining branch earlier

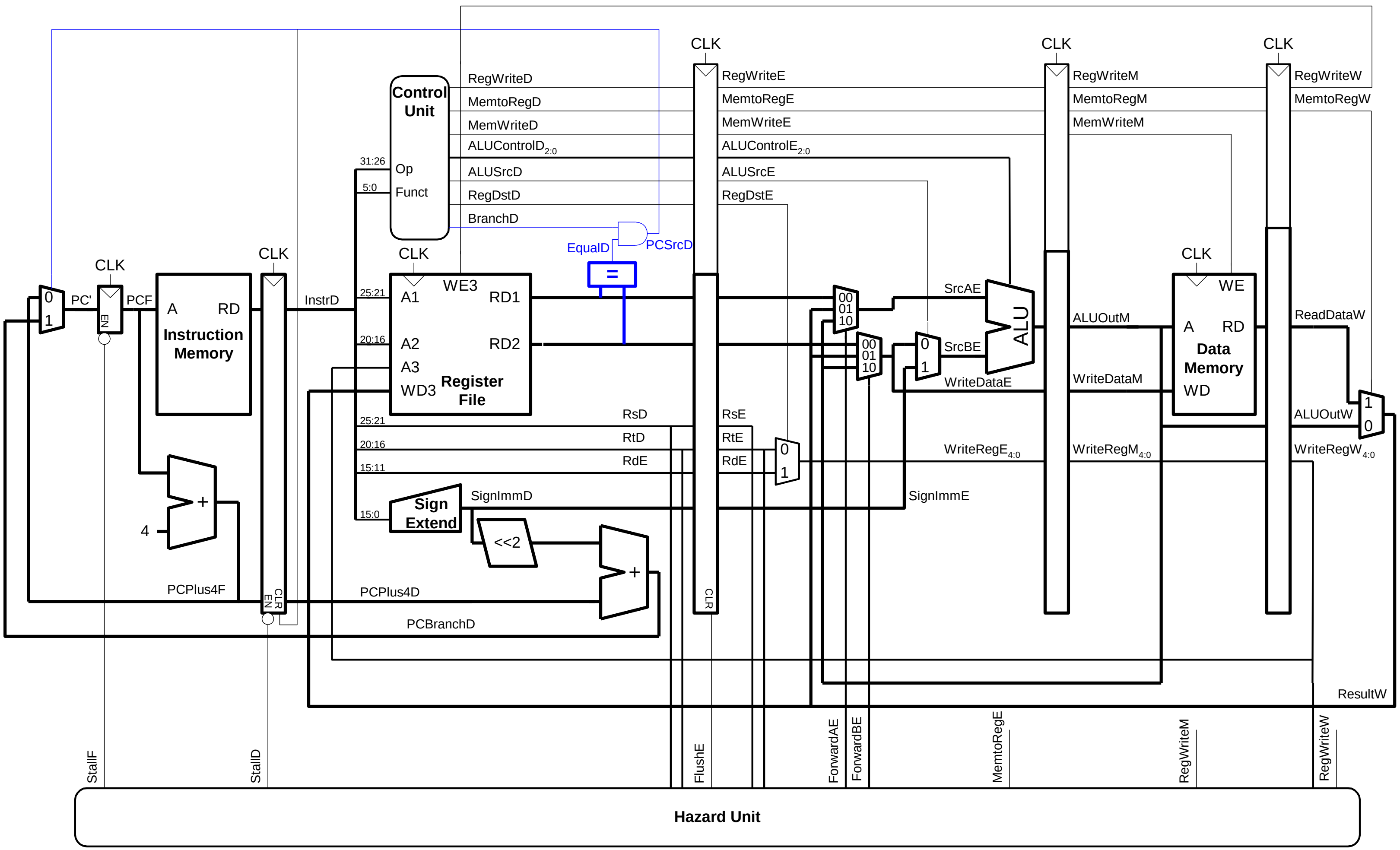
Control Hazards: Original Pipeline



Control Hazards



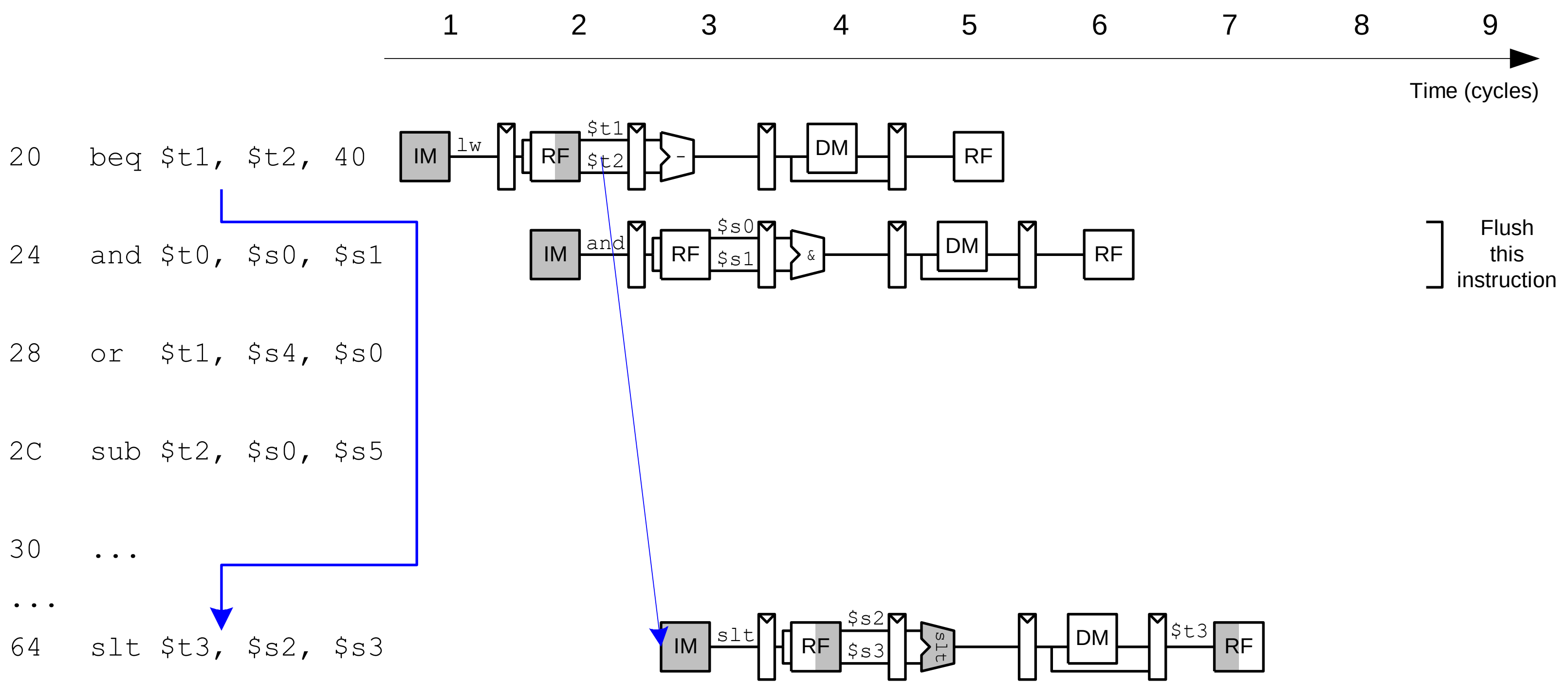
Early Branch Resolution



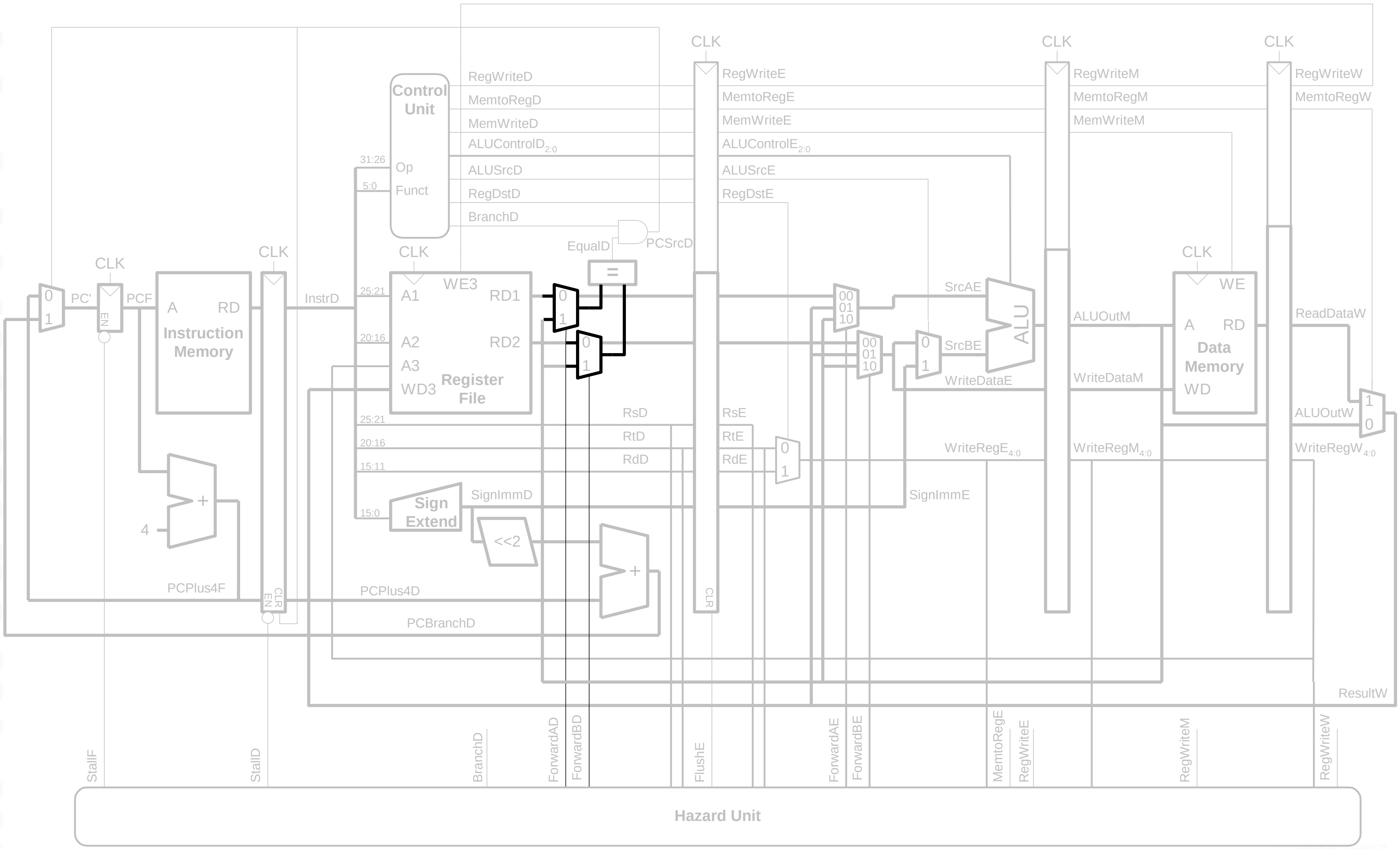
But: introduced another data hazard in Decode stage!

ELSEVIER

Early Branch Resolution



Forwarding to Early-branch HW



Control Forwarding & Stalling Logic

- **Forwarding logic:**

```
ForwardAD = (rsD !=0) AND (rsD == WriteRegM) AND RegWriteM
ForwardBD = (rtD !=0) AND (rtD == WriteRegM) AND RegWriteM
```

- **Stalling logic:**

```
branchstall = BranchD AND RegWriteE AND
                 (WriteRegE == rsD OR WriteRegE == rtD)
              OR
              BranchD AND MemtoRegM AND
                 (WriteRegM == rsD OR WriteRegM == rtD)

StallF = StallD = FlushE = (lwstall OR branchstall)
```

Branch Prediction

- Guess whether branch will be taken
  - Backward branches are usually taken (in bottom-tested loops)
  - Consider history to improve guess
- Good prediction significantly reduces fraction of branches requiring a flush
- Requires HW for history table, etc

Pipelined Performance Example

- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type
- Suppose:
  - 40% of loads used by next instruction
  - 25% of branches mispredicted
  - All jumps flush next instruction (JTA not ready)
- **What is the average CPI?**

Calculation of Average CPI

- **Average CPI is the weighted average of CPI_{lw}, CPI_{sw}, CPI_{beq}, CPI_{j} and $CPI_{R\text{-}type}$**
- **For pipeline processors, CPI = 1 + # of stall cycles**

Load CPI = 1 when no stall, = 2 when load-use occurs (1 stall)

- $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$
- $CPI_{sw} = 1$

Branch CPI = 1 when no stall, = 2 when it mispredicts and stalls

- $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$

Jump CPI = 2 since it always requires 1 stall

- $CPI_{j} = 2$
- $CPI_{R\text{-}type} = 1$

Average CPI = (0.25)(1.4) + (0.1)(1) + (0.11)(1.25) + (0.02)(2) + (0.52)(1) **= 1.15**

Pipelined Performance

- Pipelined processor critical path:

$$T_c = \max\{$$

$$t_{pcq} + t_{\text{mem}} + t_{\text{setup}}$$

$$2(t_{RFread} + t_{\text{mux}} + t_{\text{eq}} + t_{\text{AND}} + t_{\text{mux}} + t_{\text{setup}})$$

$$t_{pcq} + t_{\text{mux}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{setup}}$$

$$t_{pcq} + t_{\text{memwrite}} + t_{\text{setup}}$$

$$2(t_{pcq} + t_{\text{mux}} + t_{\text{RFwrite}})\}$$

Pipelined Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\mathrm{read}}$	150
Register file setup	$t_{RF\mathrm{setup}}$	20
Equality comparator	t_{eq}	40
AND gate	t_{AND}	15
Memory write	T_{memwrite}	220
Register file write	$t_{RF\mathrm{write}}$	100 ps

$$T_c = 2(t_{RFread} + t_{\mathrm{mux}} + t_{\mathrm{eq}} + t_{\mathrm{AND}} + t_{\mathrm{mux}} + t_{\mathrm{setup}})$$
$$= 2[150 + 25 + 40 + 15 + 25 + 20] \text{ ps} = \mathbf{550\ ps}$$

MICROARCHITECTURE

ELSEVIER

Pipelined Performance Example

Program with IC = 100 billion instructions

Execution Time = IC × CPI × T_c

$= (100 \times 10^9)(1.15)(550 \times 10^{-12})$

= 63 seconds

Processor Performance Comparison

Processor	Execution Time (seconds)	Speedup (single-cycle as baseline)
Single-cycle	92.5	1
Multicycle	133	0.70
Pipelined	63	1.47

Advanced Microarchitecture

- Deep Pipelining
- Branch Prediction
- Superscalar Processors
- Out of Order Processors
- Register Renaming
- SIMD
- Multithreading
- Multiprocessors

Deep Pipelining

- 10-20 stages typical
- Number of stages limited by:
  - Pipeline hazards
  - Sequencing overhead
  - Power
  - Cost

MICROARCHITECTURE

ELSEVIER

Branch Prediction

- Ideal pipelined processor: CPI = 1
- Branch misprediction increases CPI
- **Static branch prediction:**
  – Check direction of branch (forward or backward)
  – If backward, predict taken
  – Else, predict not taken
- **Dynamic branch prediction:**
  – Keep history of last (several hundred) branches in *branch target buffer*, record:
    - Branch destination
    - Whether branch was taken

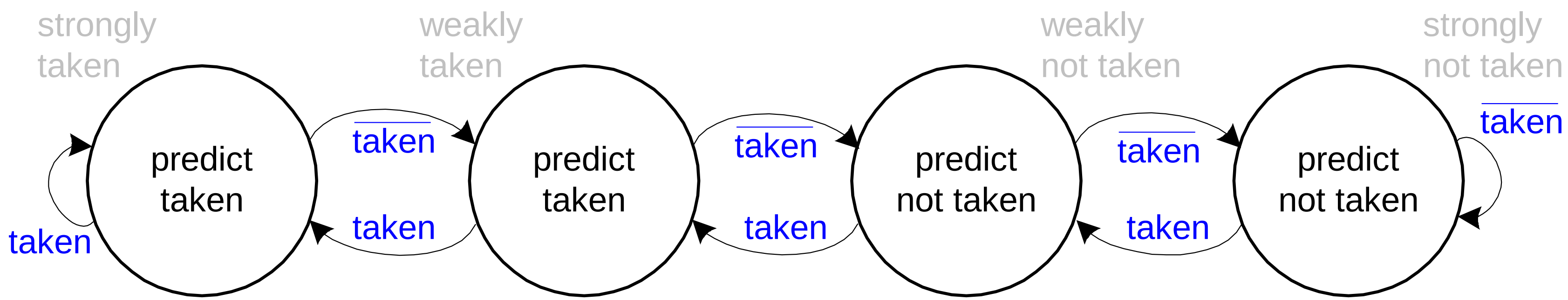
Branch Prediction Example

```
   add  $s1, $0, $0      # sum = 0
   add  $s0, $0, $0      # i   = 0
   addi $t0, $0, 10      # $t0 = 10
for:
   beq  $s0, $t0, done   # if i == 10, branch
   add  $s1, $s1, $s0    # sum = sum + i
   addi $s0, $s0, 1      # increment i
   j    for
done:
```

1-Bit Branch Predictor

- Remembers whether branch was taken the last time and does the same thing
- Mispredicts first and last branch of loop

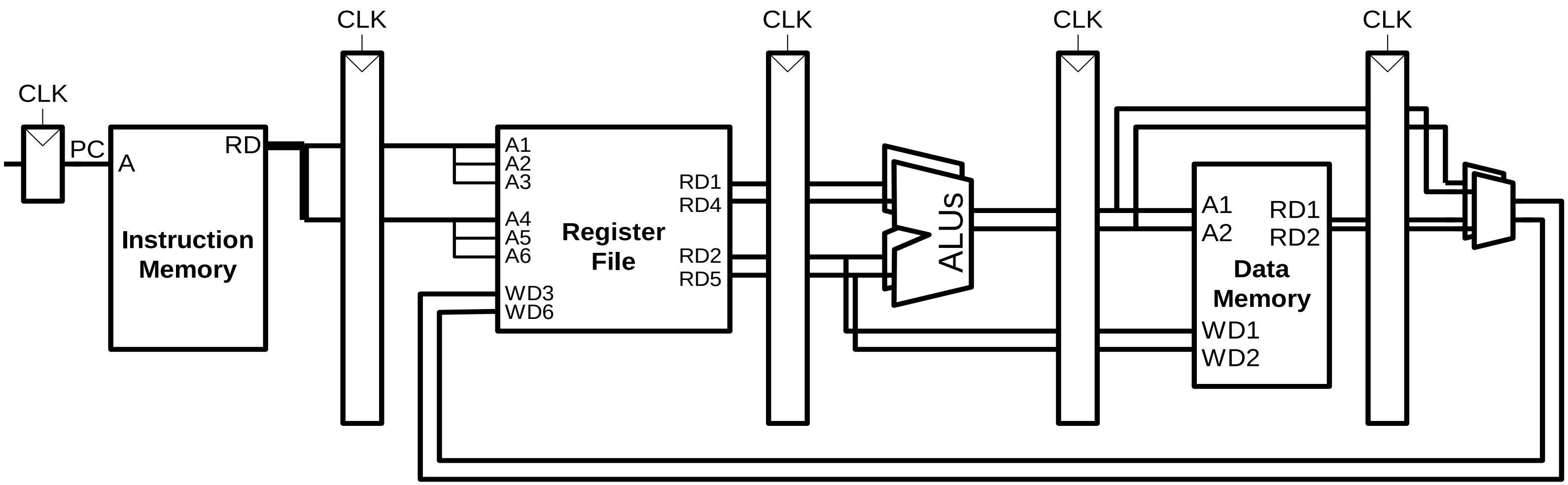
2-Bit Branch Predictor



Only mispredicts the last branch of the loop

Superscalar

- Multiple copies of datapath execute multiple instructions at once
- Dependencies make it tricky to issue multiple instructions at once



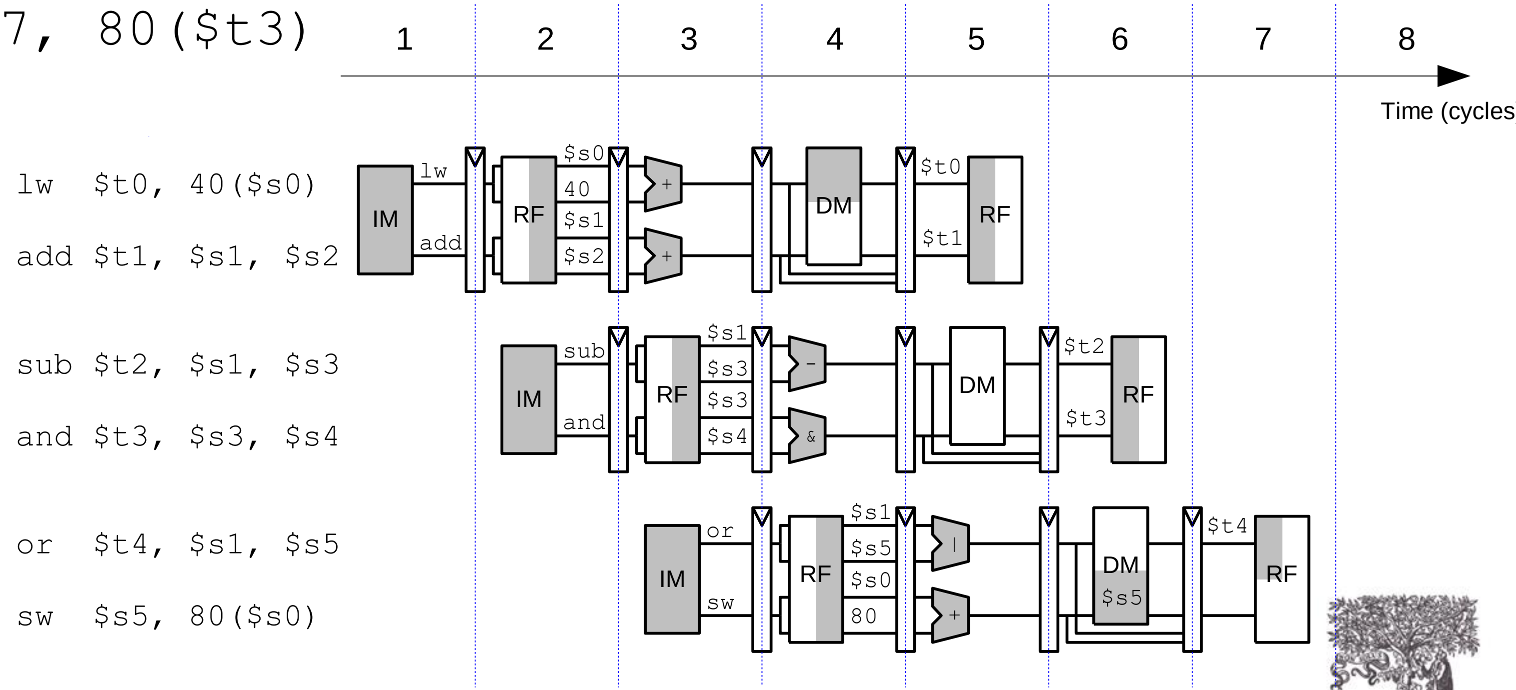
ELSEVIER

Superscalar Example

```
lw   $t0, 40($s0)
add  $t1, $t0, $s1
sub  $t0, $s2, $s3
and  $t2, $s4, $t0
or   $t3, $s5, $s6
sw   $s7, 80($t3)
```

Ideal IPC: 2

Actual IPC: 2



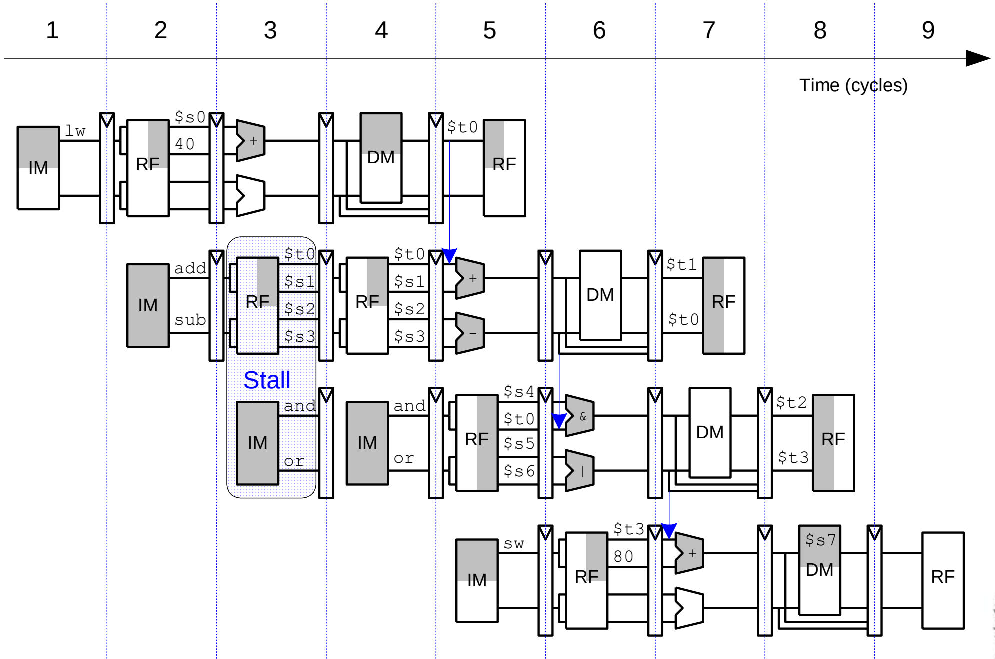
Superscalar with Dependencies

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

Ideal IPC: **2**

Actual IPC: **6/5 = 1.17**

```
lw   $t0, 40($s0)

add  $t1, $t0, $s1
sub  $t0, $s2, $s3

and  $t2, $s4, $t0
or   $t3, $s5, $s6

sw   $s7, 80($t3)
```



Out of Order Processor

- Looks ahead across multiple instructions
- Issues as many instructions as possible at once
- Issues instructions out of order (as long as no dependencies)
- **Dependencies:**
  - **RAW** (read after write): one instruction writes, later instruction reads a register
  - **WAR** (write after read): one instruction reads, later instruction writes a register
  - **WAW** (write after write): one instruction writes, later instruction writes a register

Out of Order Processor

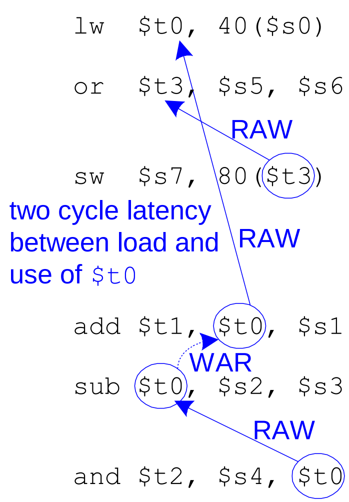
- **Instruction level parallelism (ILP):** number of instruction that can be issued simultaneously (average < 3)
- **Scoreboard:** table that keeps track of:
  - Instructions waiting to issue
  - Available functional units
  - Dependencies

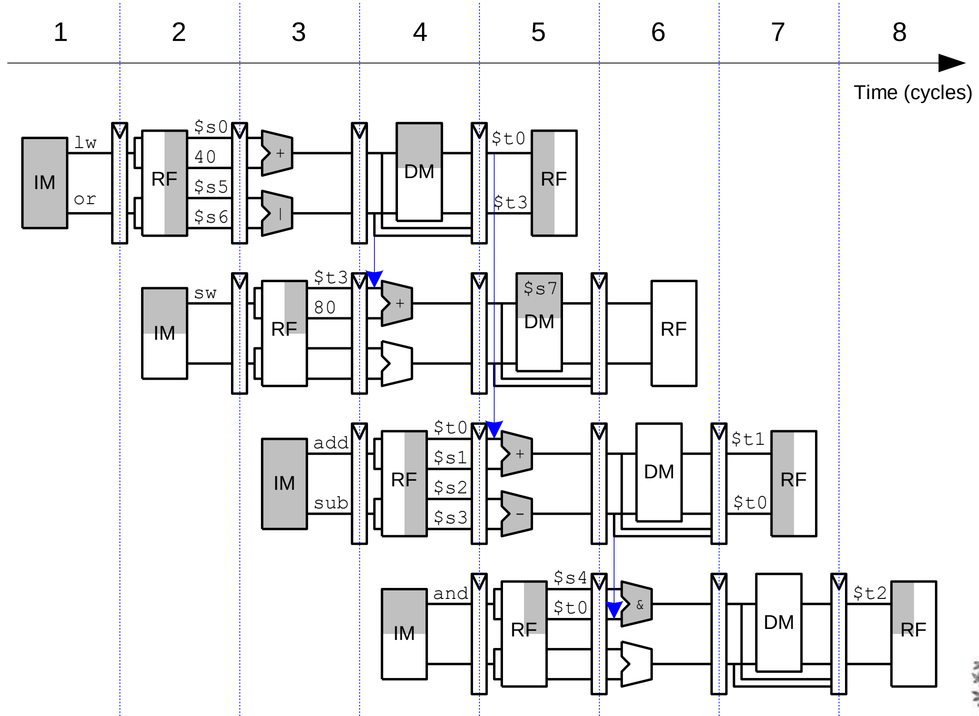
Out of Order Processor Example

```
lw   $t0, 40($s0)
add  $t1, $t0, $s1
sub  $t0, $s2, $s3
and  $t2, $s4, $t0
or   $t3, $s5, $s6
sw   $s7, 80($t3)
```

Ideal IPC: **2**

Actual IPC: **6/4 = 1.5**





Register Renaming

```
lw   $t0, 40($s0)
add  $t1, $t0, $s1
sub  $t0, $s2, $s3
and  $t2, $s4, $t0
or   $t3, $s5, $s6
sw   $s7, 80($t3)
```

Ideal IPC: 2

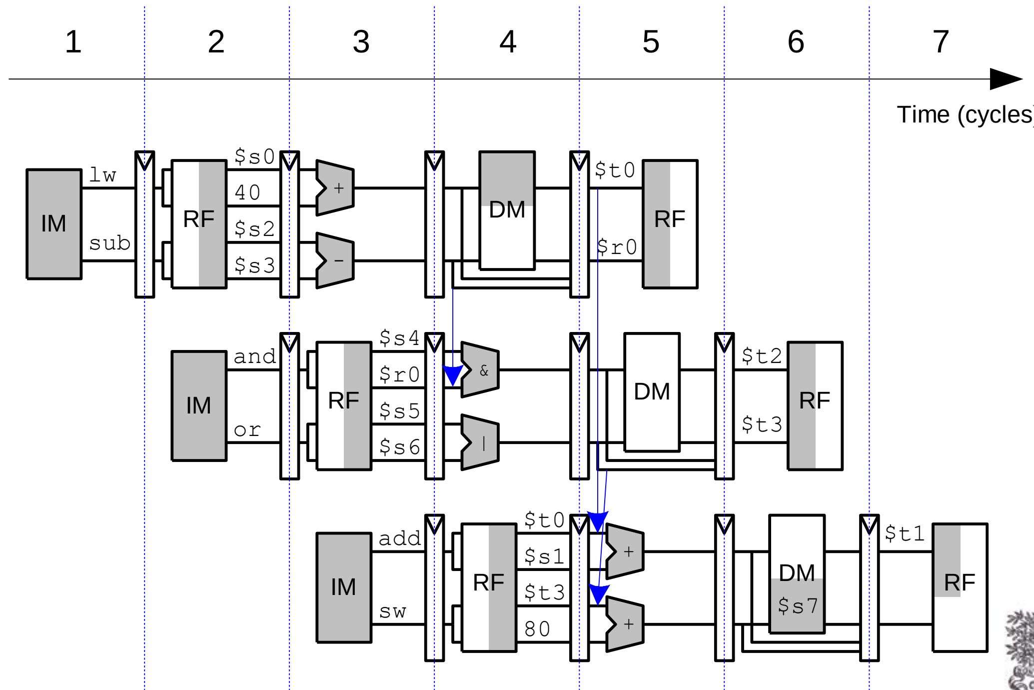
Actual IPC: 6/3 = 2

```
lw   $t0, 40($s0)

sub  $r0, $s2, $s3

and  $t2, $s4, $r0

or   $t3, $s5, $s6

add  $t1, $t0, $s1

sw   $s7, 80($t3)
```

2-cycle RAW

RAW

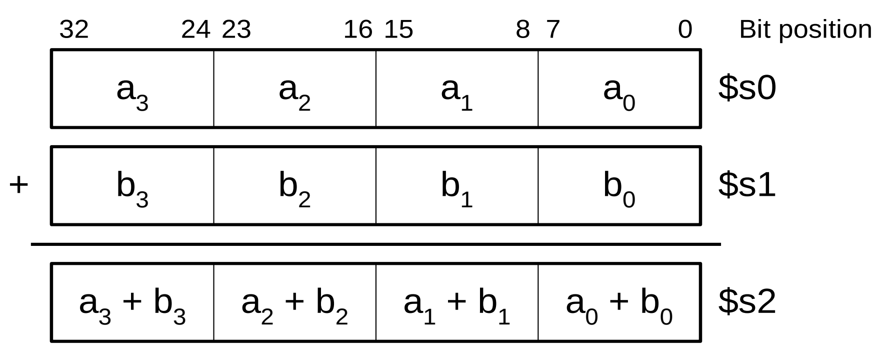
RAW



SIMD

- Single Instruction Multiple Data (SIMD)
  - Single instruction acts on multiple pieces of data at once
  - Common application: graphics
  - Perform short arithmetic operations (also called *packed arithmetic*)
- For example, add four 8-bit elements

```
padd8 $s2, $s0, $s1
```



Advanced Architecture Techniques

- **Multithreading**
  - Wordprocessor: thread for typing, spell checking, printing
- **Multiprocessors**
  - Multiple processors (cores) on a single chip

Threading: Definitions

- **Process:** program running on a computer
  - Multiple processes can run at once: e.g., surfing Web, playing music, writing a paper
- **Thread:** part of a program
  - Each process has multiple threads: e.g., a word processor may have threads for typing, spell checking, printing

Threads in Conventional Processor

- One thread runs at once
- When one thread stalls (for example, waiting for memory):
  - Architectural state of that thread stored
  - Architectural state of waiting thread loaded into processor and it runs
  - Called **context switching**
- Appears to user like all threads running simultaneously

Multithreading

- Multiple copies of architectural state
- Multiple threads **active** at once:
  - When one thread stalls, another runs immediately
  - If one thread can't keep all execution units busy, another thread can use them
- Does not increase instruction-level parallelism (ILP) of single thread, but increases throughput

Intel calls this "hyperthreading"

Multiprocessors

- Multiple processors (cores) with a method of communication between them
- Types:
  - **Homogeneous**: multiple cores with shared memory
  - **Heterogeneous:** separate cores for different tasks (for example, DSP and CPU in cell phone)
  - **Clusters:** each core has own memory system