MIPS-Lite Single-Cycle Control

COE608: Computer Organization and Architecture

Dr. Gul N. Khan
http://www.ee.ryerson.ca/~gnkhan
Electrical and Computer Engineering
Ryerson University

Overview

- Single cycle Data path Review
- Data path Analysis for different instructions
- Data path Control Signals.
- ALU control Signals
- Decoding Control Signals
- Single-cycle Control and its Performance

Part of section 4.4 from the Textbook

Overview

5 steps to design a processor

1. Analyze instruction set => data path <u>requirements</u>
2. Select set of data path components & establish clock methodology
3. <u>Assemble</u> data path meeting the requirements
4. Analyze implementation of each instruction to determine setting of control points that affects the register transfer.
5. Assemble the control logic

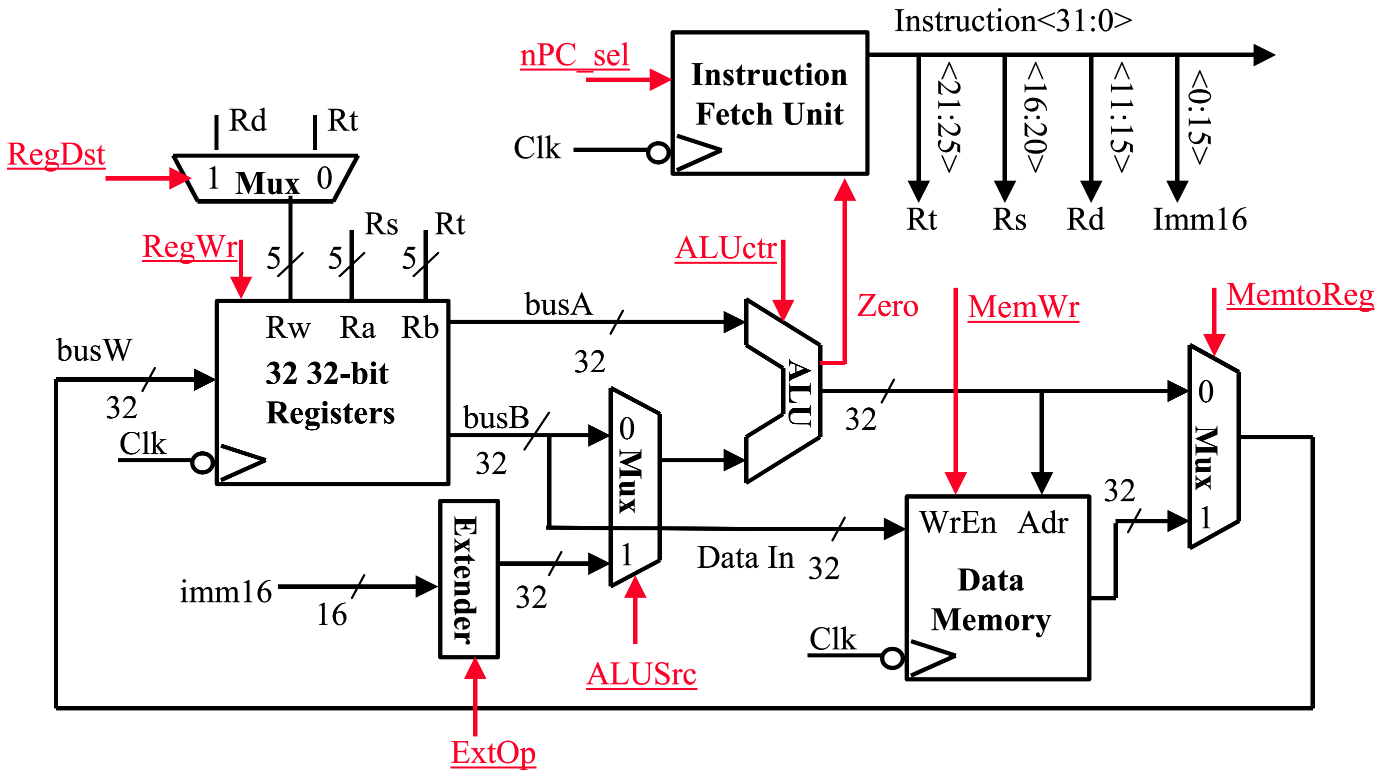
MIPS makes it easier

Single cycle data path

CPI=1

A Single Cycle Data path

We have everything except control signals (underline). How to generate the control signals?



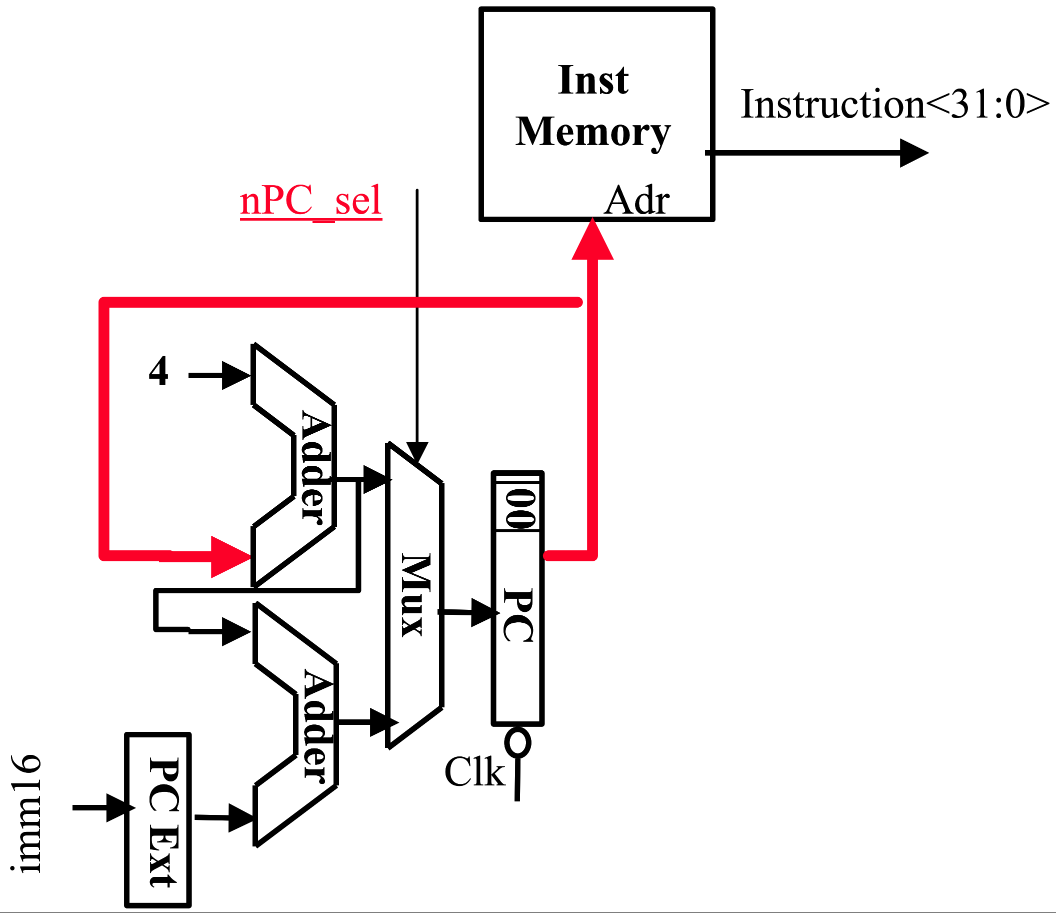
The Add Instruction

31 26	21	16	11	6	0
op	**rs**	**rt**	**rd**	**shamt**	**funct**
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

add rd, rs, rt

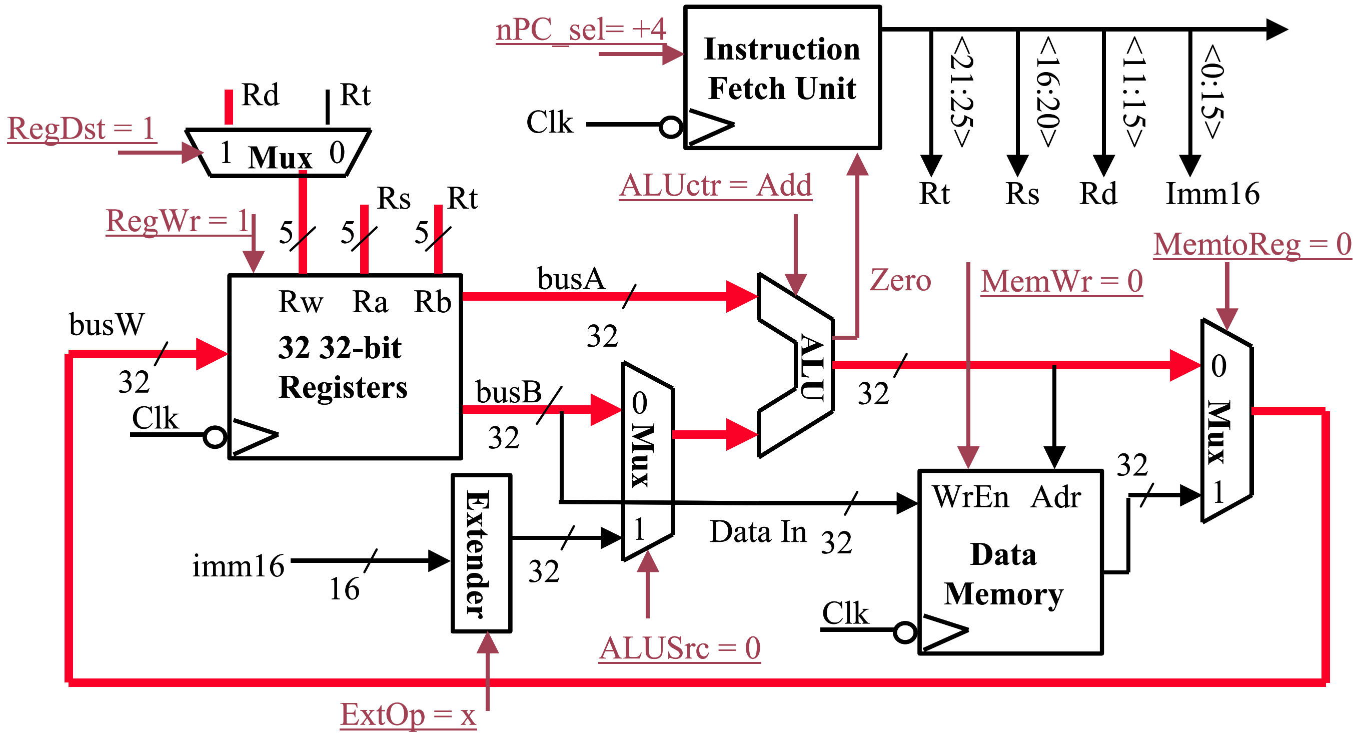
mem[PC]	Fetch the instruction from memory
R[rd] <= R[rs] + R[rt]	Actual operation
PC <= PC + 4	Calculate the next instruction's address

Instruction Fetch Unit at the Beginning



Data path during Add

R[rd] <= R[rs] + R[rt]

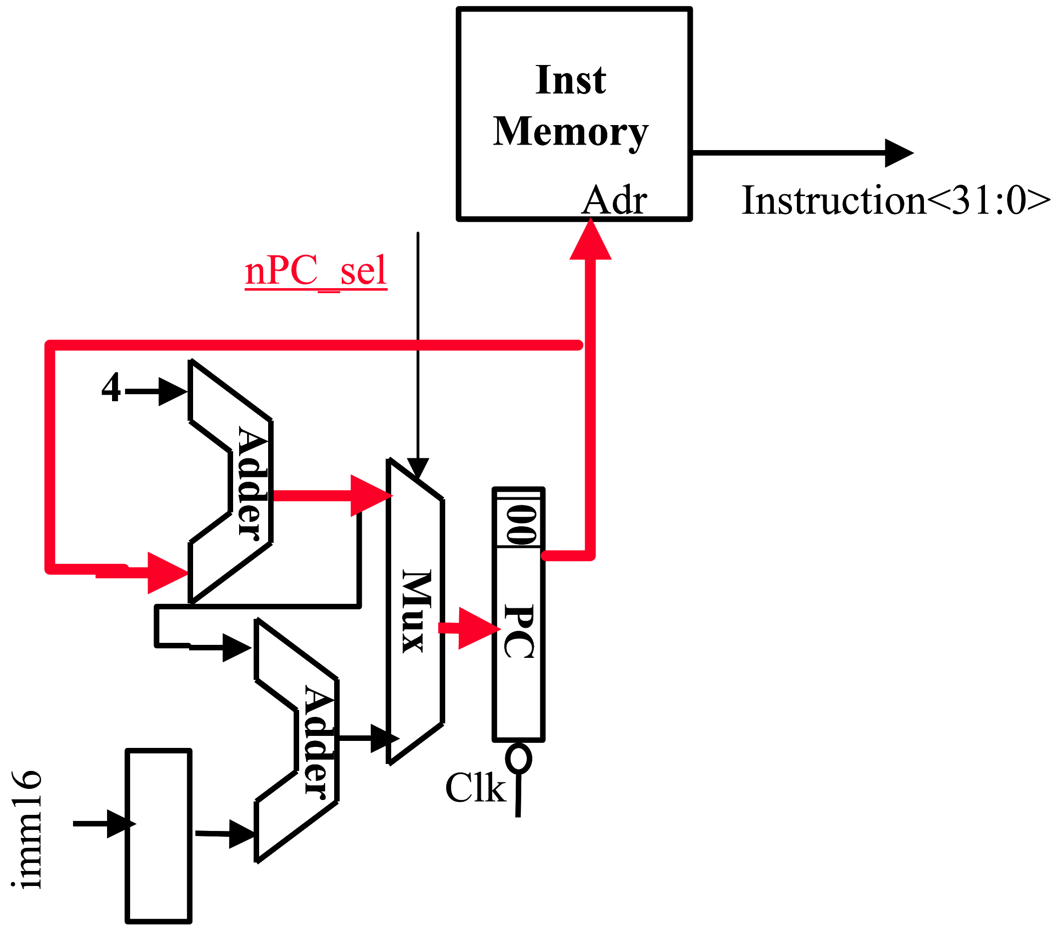


Memory Read when MemWr = 0

IFU at the end of Add

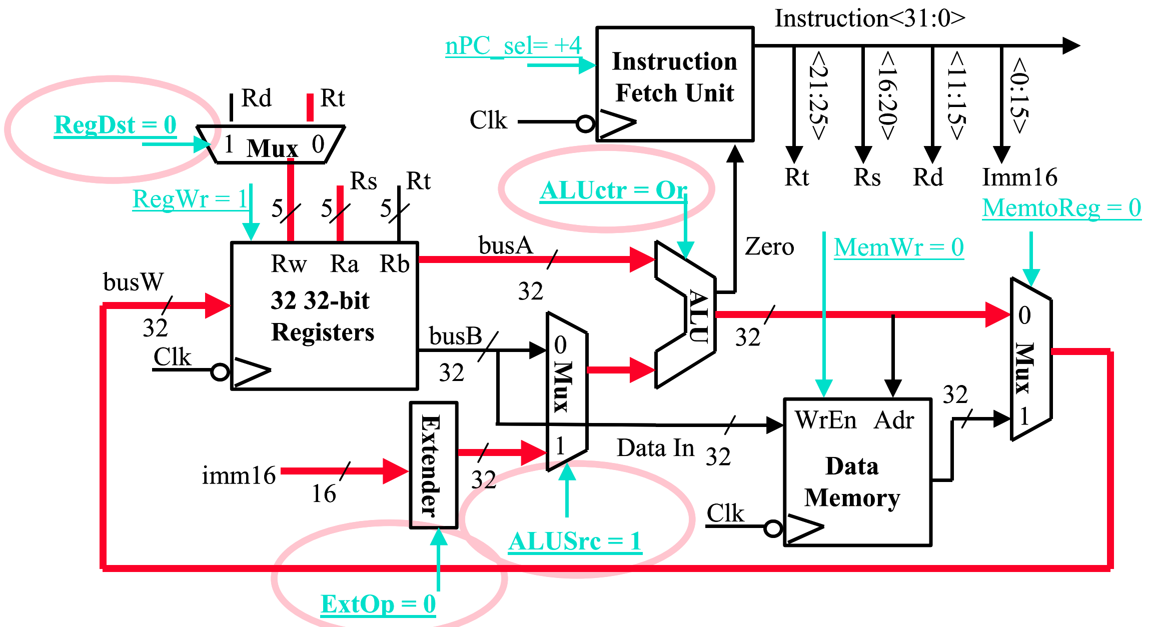
PC <= PC + 4

This is the same for all instructions except: Branch and Jump



Data path during ori

R[rt] <= R[rs] or ZeroExt[Imm16]

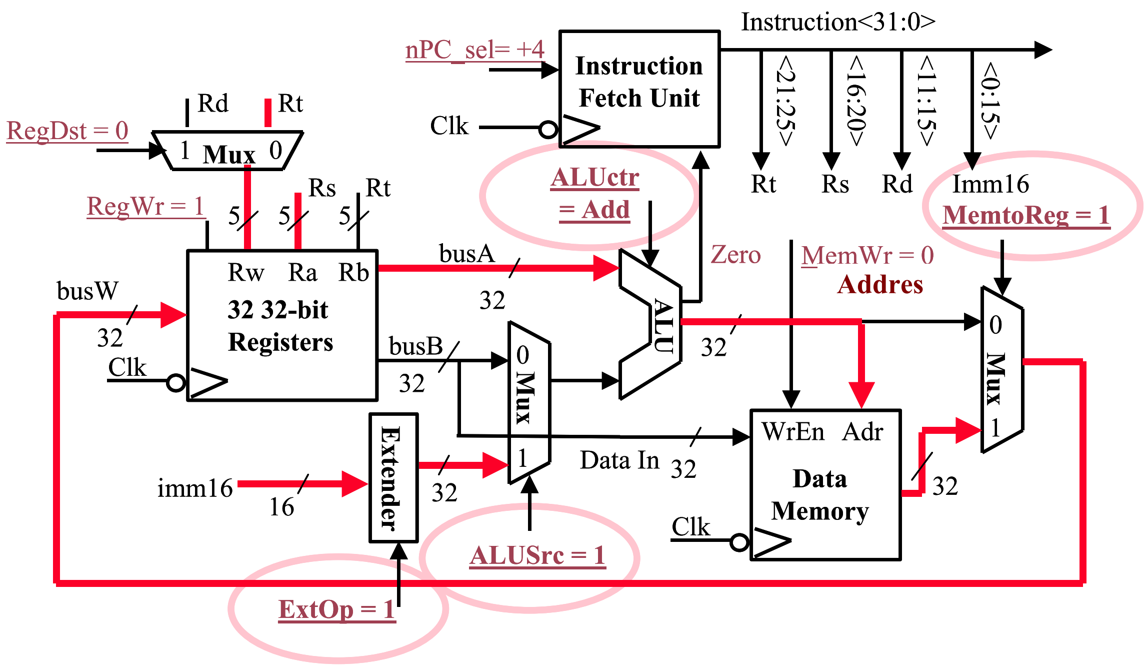


Zero Extend

The Rs field is fed to Ra address port: R[rs] is placed on busA. Other ALU operand will come from the immediate field.

Data path during Load

R[rt] <= Data Memory {R[rs] + SignExt[imm16]}

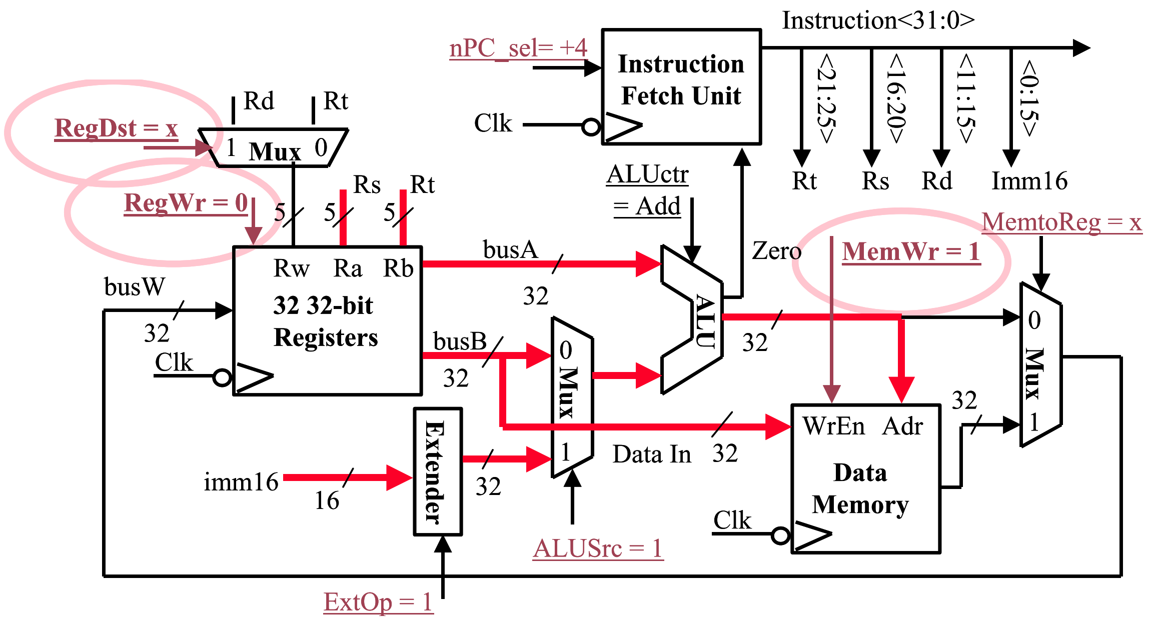


Sign Extend

Add R[rs] to Sign Extended Immediate field to form memory address.
Use memory address to access memory and write data back to R[rt].

Data path during Store

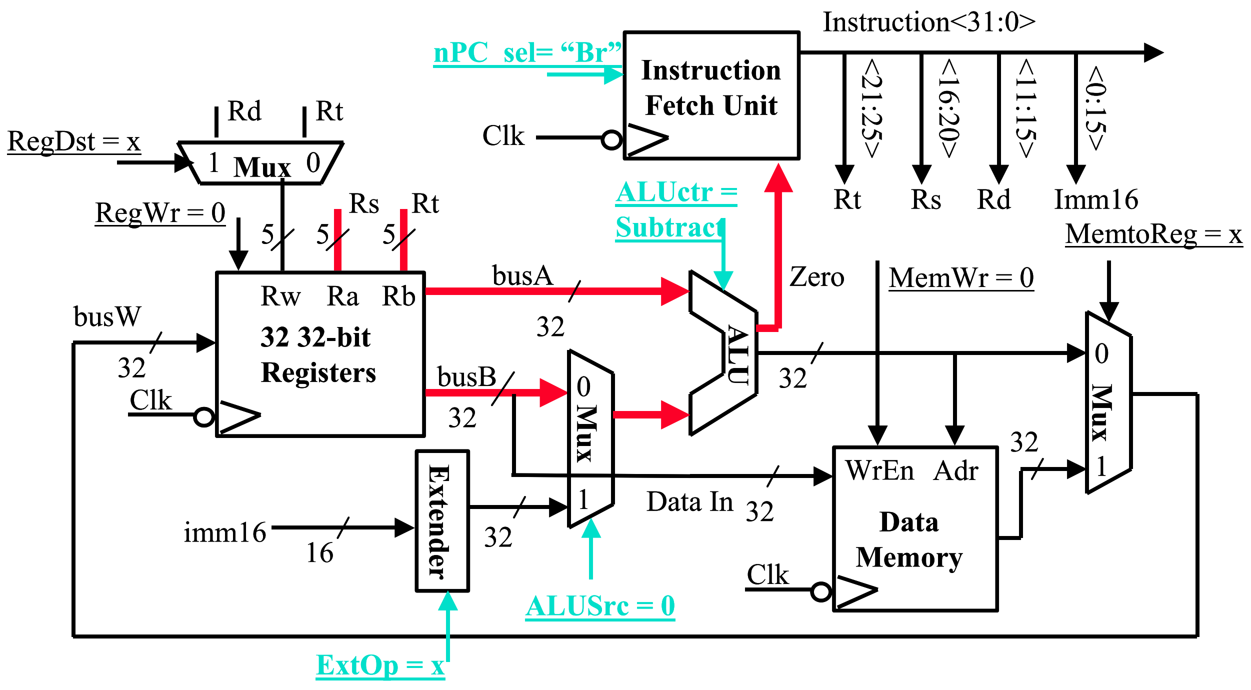
Data Memory {R[rs] + SignExt[imm16]} <= R[rt]



Store sends the contents of register specified by Rt to data memory.

Data path during Branch

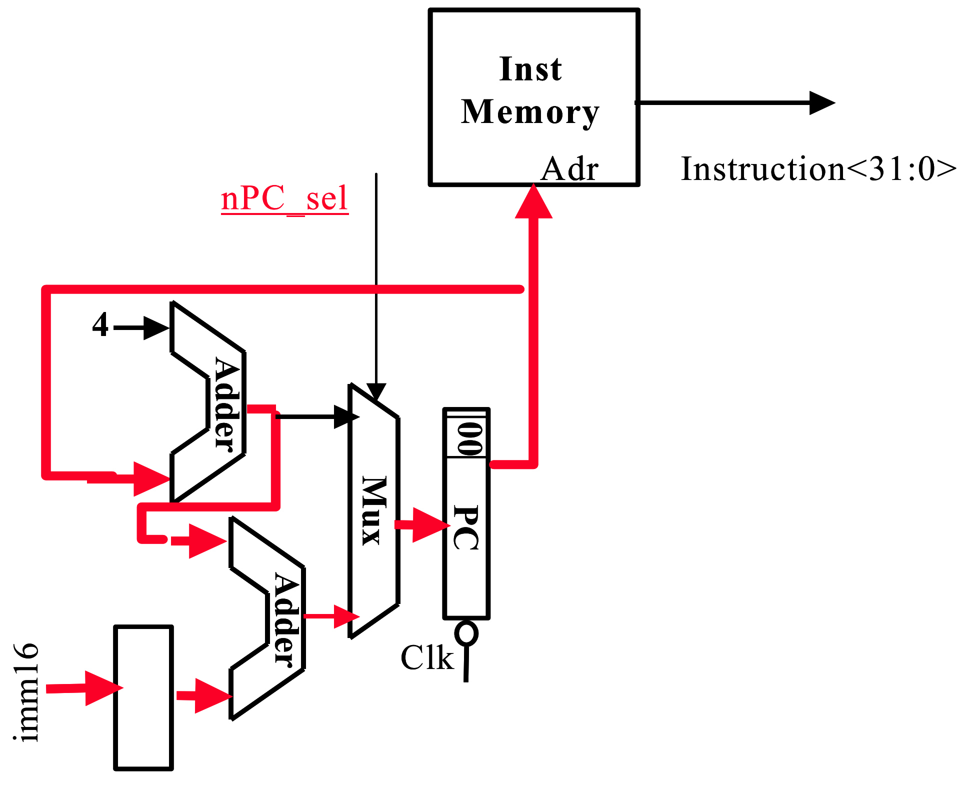
if (R[rs] - R[rt] == 0) then Zero <= 1 ; else Zero <= 0



Don't care

Subtracts the register specified in the Rt field from the register specified in the Rs field and set Zero condition accordingly.

IFU at the End of Branch

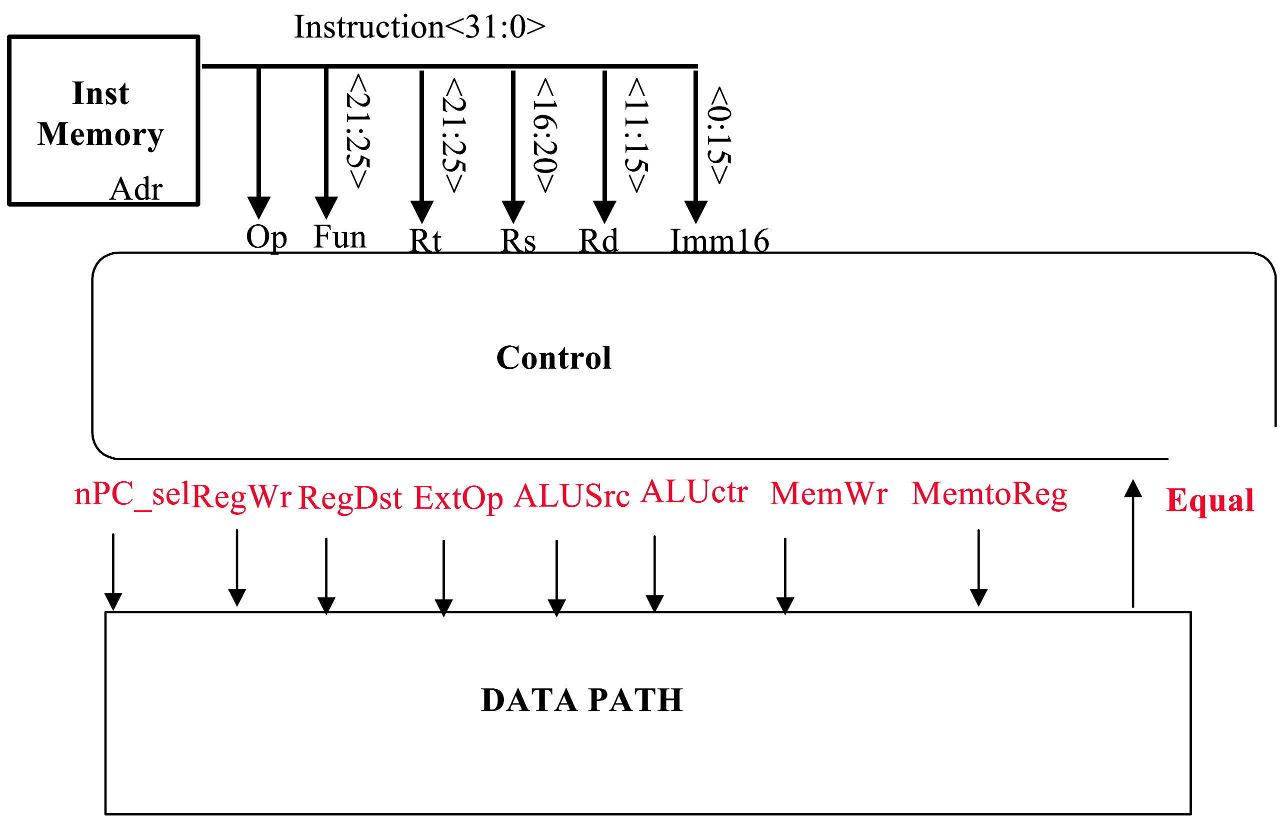


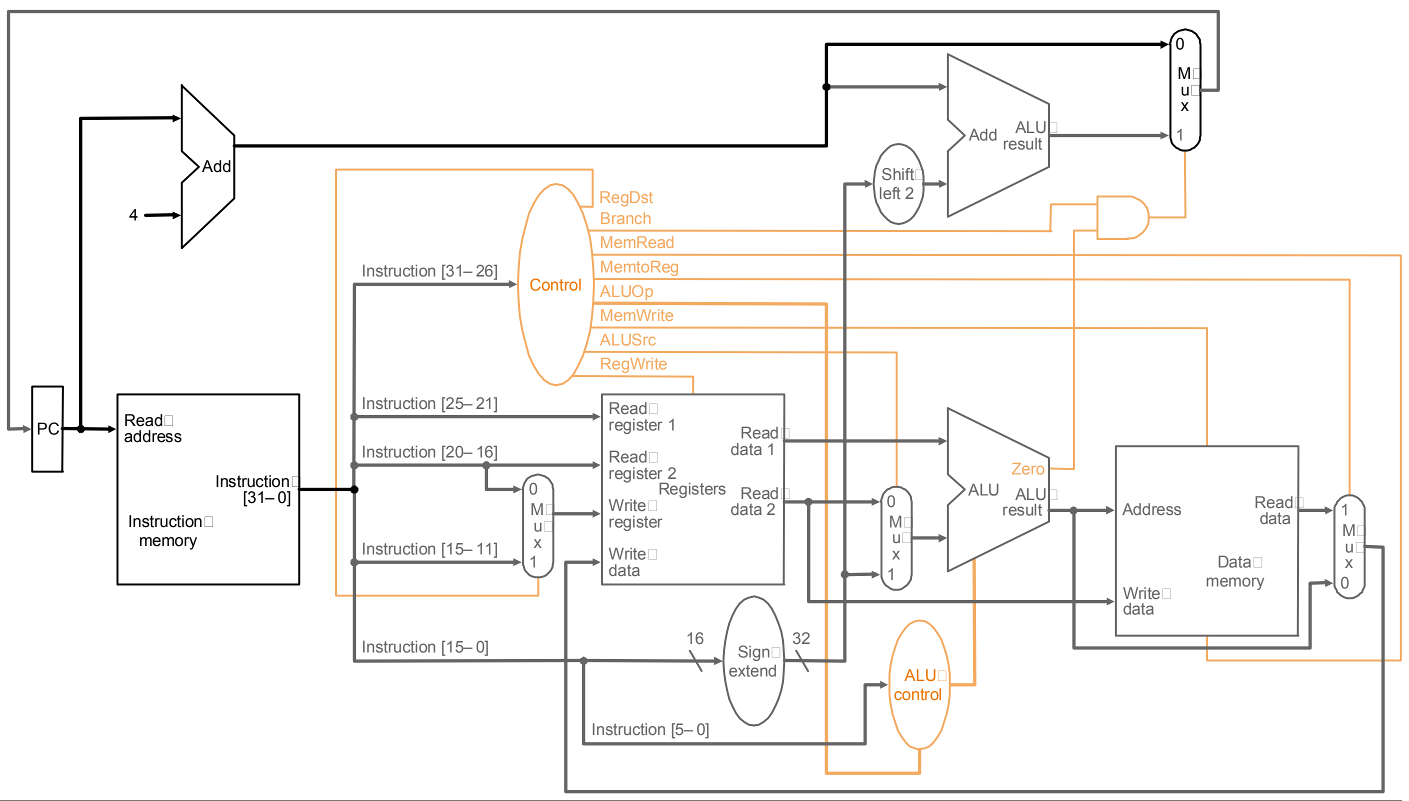
Sign Extend

PC = PC + 4 + Imm16

When the branch condition Zero is true (Zero = 1).

Given Data path: RTL => Control





Summary of Control Signals

<u>inst   Register Transfer</u>

ADD R[rd] <= R[rs] + R[rt]; PC <= PC + 4

ALUsrc = busB, ALUctr = “add”, RegDst = rd, RegWr, nPC_sel = “+4”

SUB R[rd] <= R[rs] – R[rt]; PC <= PC + 4

ALUsrc = busB, ALUctr = “sub”, RegDst = rd, RegWr, nPC_sel = “+4”

ORi   R[rt] <= R[rs] + zero_ext(Imm16); PC <= PC + 4

ALUsrc = Imm, Extop = “Z”, ALUctr = “or”, RegDst = rt, RegWr, nPC_sel = “+4”

LOAD R[rt] <= MEM[R[rs] + sign_ext(Imm16)];
PC <= PC + 4

ALUsrc = Imm, Extop = “Sn”, ALUctr = “add”, MemWr = 0, MemtoReg, RegDst = rt, RegWr,   nPC_sel = “+4”

STORE MEM[R[rs] + sign_ext(Imm16)] <= R[rt];
PC <= PC + 4

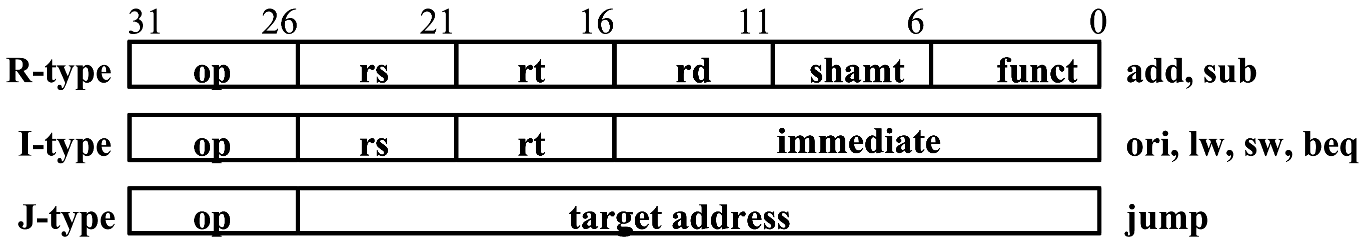
ALUsrc = Imm, Extop = “Sn”, ALUctr = “add”, MemWr = 1, nPC_sel = “+4”

BEQ if (R[rs] == R[rt]) then
PC <= PC + sign_ext(Imm16)] || 00 else PC <= PC + 4

nPC_sel = “Br”, ALUctr = “sub”

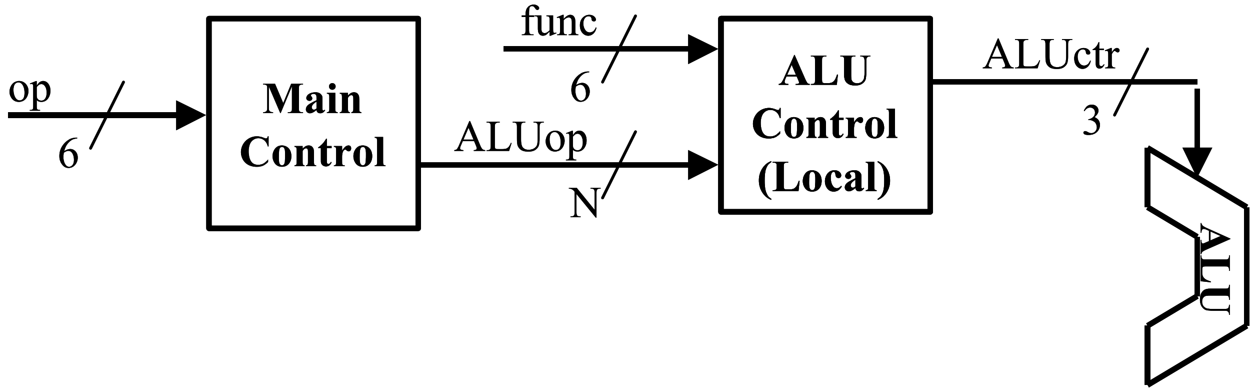
Summary of Control Signals

See → func	10 0000	10 0010	Don't Care				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	sub	ori	lw	sw	beq	jump
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWr	1	1	1	1	0	0	0
MemWr	0	0	0	0	1	0	0
nPCsel	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr<2:0>	Add	Subtract	Or	Add	Add	Subtract	xxx



Local Decoding

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx



The Encoding of ALUop

ALUop has to be 2 bits wide to represent:

- (1) "R-type" instructions
- "I-type" instructions that require the ALU to perform:
  - (2) Or, (3) Add, and (4) Subtract

For full MIPS, ALUop has to be 3 bits to represent:

(1) "R-type" instructions

- "I-type" instructions that require the ALU to perform:
  (2) Or, (3) Add, (4) Subtract, and (5) And (e.g. andi)

	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx

funct<5:0>	**Instruction Operation**
10 0000	add
10 0010	subtract
10 0100	and
10 0101	or
10 1010	set-on-less-than

ALUctr<2:0>	**ALU Operation**
000	Add
001	Subtract
010	And
110	Or
111	Set-on-less-than

The Truth Table for ALUctr

funct<3:0>	Instruction Op.
0000	add
0010	subtract
0100	and
0101	or
1010	set-on-less-than

ALUop (Symbolic)	**R-type**	**ori**	**lw**	**sw**	**beq**
	"R-type"	Or	Add	Add	Subtract
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01

ALUop			**func**				**ALU Operation**	**ALUctr**		
bit<2>	**bit<1>**	**bit<0>**	**bit<3>**	**bit<2>**	**bit<1>**	**bit<0>**		**bit<2>**	**bit<1>**	**bit<0>**
0	0	0	x	x	x	x	Add	0	1	0
0	x	1	x	x	x	x	Subtract	1	1	0
0	1	x	x	x	x	x	Or	0	0	1
1	x	x	0	0	0	0	Add	0	1	0
1	x	x	0	0	1	0	Subtract	1	1	0
1	x	x	0	1	0	0	And	0	0	0
1	x	x	0	1	0	1	Or	0	0	1
1	x	x	1	0	1	0	Set on <	1	1	1

The Logic Equation for ALUctr<2>

ALUop			**func**				
bit<2>	**bit<1>**	**bit<0>**	**bit<3>**	**bit<2>**	**bit<1>**	**bit<0>**	**ALUctr<2>**
0	x	1	x	x	x	x	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

This makes func<3> a don't care

$$\text{ALUctr<2>} = \overline{\text{ALUop<2>}}\ \&\ \text{ALUop<0>} + \text{ALUop<2>}\ \&\ \overline{\text{func<2>}}\ \&\ \text{func<1>}\ \&\ \overline{\text{func<0>}}$$

The Logic Equation for ALUctr<1>

ALUop			**func**				
bit<2>	**bit<1>**	**bit<0>**	**bit<3>**	**bit<2>**	**bit<1>**	**bit<0>**	**ALUctr<1>**
0	0	0	x	x	x	x	1
0	x	1	x	x	x	x	1
1	x	x	0	0	0	0	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

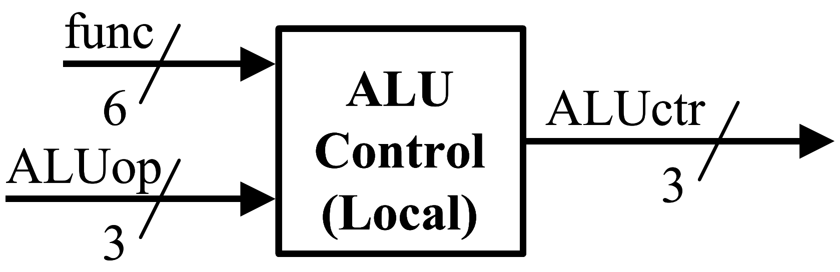
$$\text{ALUctr<1>} = \overline{\text{ALUop<2>}}\ \&\ \overline{\text{ALUop<1>}} + \text{ALUop<2>}\ \&\ \overline{\text{func<2>}}\ \&\ \overline{\text{func<0>}}$$

ALU Control Block

The Logic Equation for ALUctr<0>

ALUop			func				
bit<2>	**bit<1>**	**bit<0>**	**bit<3>**	**bit<2>**	**bit<1>**	**bit<0>**	**ALUctr<0>**
0	1	x	x	x	x	x	1
1	x	x	0	1	0	1	1
1	x	x	1	0	1	0	1

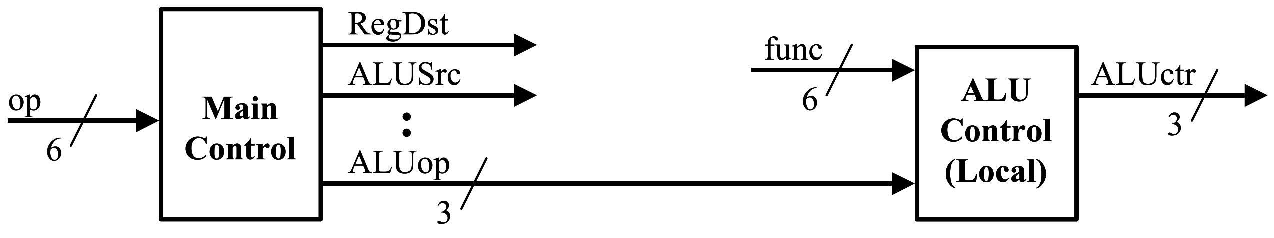
$$\begin{aligned} \text{ALUctr<0>} = {} & \overline{\text{ALUop<2>}} \,\&\, \text{ALUop<1>} \\ & + \text{ALUop<2>} \,\&\, \overline{\text{func<3>}} \,\&\, \text{func<2>} \,\&\, \overline{\text{func<1>}} \\ & \&\, \text{func<0>} + \text{ALUop<2>} \,\&\, \text{func<3>} \,\&\, \overline{\text{func<2>}} \\ & \&\, \text{func<1>} \,\&\, \overline{\text{func<0>}} \end{aligned}$$



Logic for each control signal

```
nPC_sel <= if (OP == BEQ) then EQUAL else 0

ALUsrc  <=  if (OP == “R-type”) then “busB” else “immed”

ALUctr  <= if (OP == “R-type”) then   funct
       elseif (OP == ORi) then “OR” elseif (OP == BEQ) then
       “sub”  else “add”
ExtOp    <= if (OP == ORi) then “zero” else “sign”

MemWr <= (OP ==  Store)

MemtoReg    <= (OP ==  Load)

RegWr   <= if ((OP ==  Store) || (OP == BEQ)) then 0 else 1

RegDst  <= if ((OP ==  Load) || (OP == ORi)) then 0 else 1
```

“Truth Table” for Main Control



op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWr	1	1	1	0	0	0
MemWr	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	0	1	0	0	0	x
ALUop <0>	0	0	0	0	1	x

The “Truth Table” for RegWrite

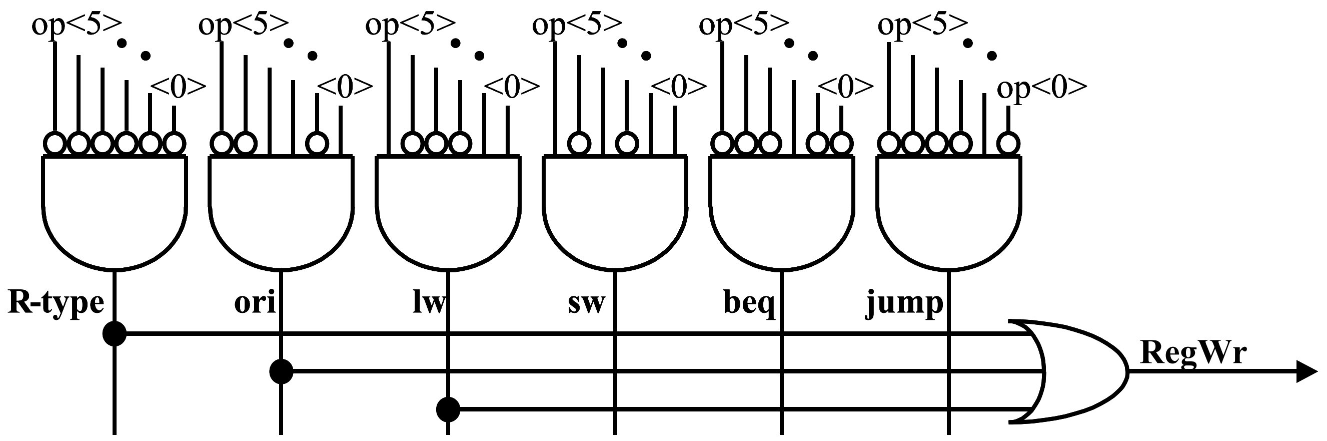
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegWrite	1	1	1	0	0	0

RegWrite = R-type + ori + lw

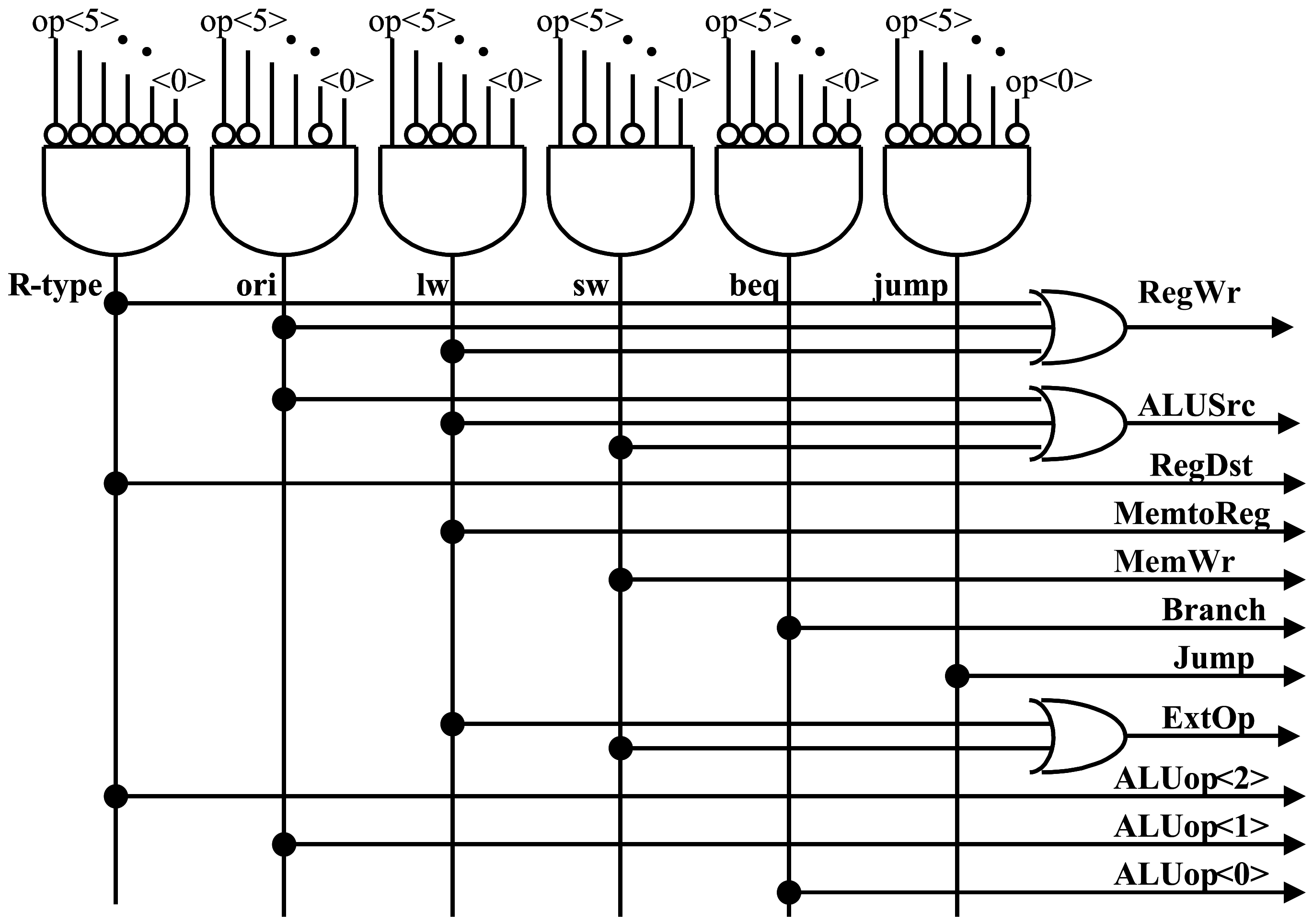
$$= \overline{op<5>} \;\&\; \overline{op<4>} \;\&\; \overline{op<3>} \;\&\; \overline{op<2>} \;\&\; \overline{op<1>} \;\&\; \overline{op<0>}$$

$$+ \; \overline{op<5>} \;\&\; \overline{op<4>} \;\&\; op<3> \;\&\; op<2> \;\&\; \overline{op<1>} \;\&\; op<0>$$

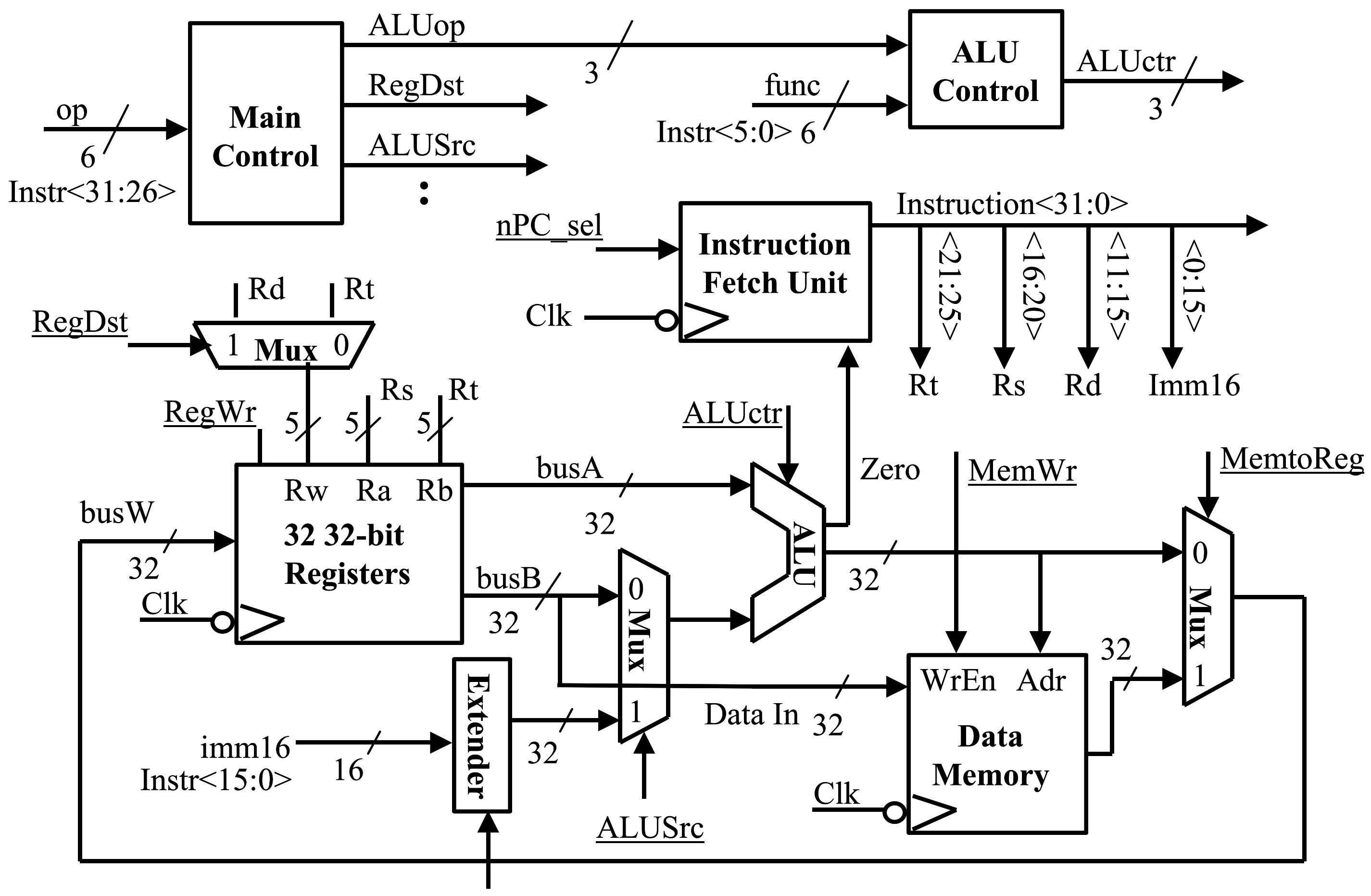
$$+ \; op<5> \;\&\; \overline{op<4>} \;\&\; \overline{op<3>} \;\&\; \overline{op<2>} \;\&\; op<1> \;\&\; op<0>$$



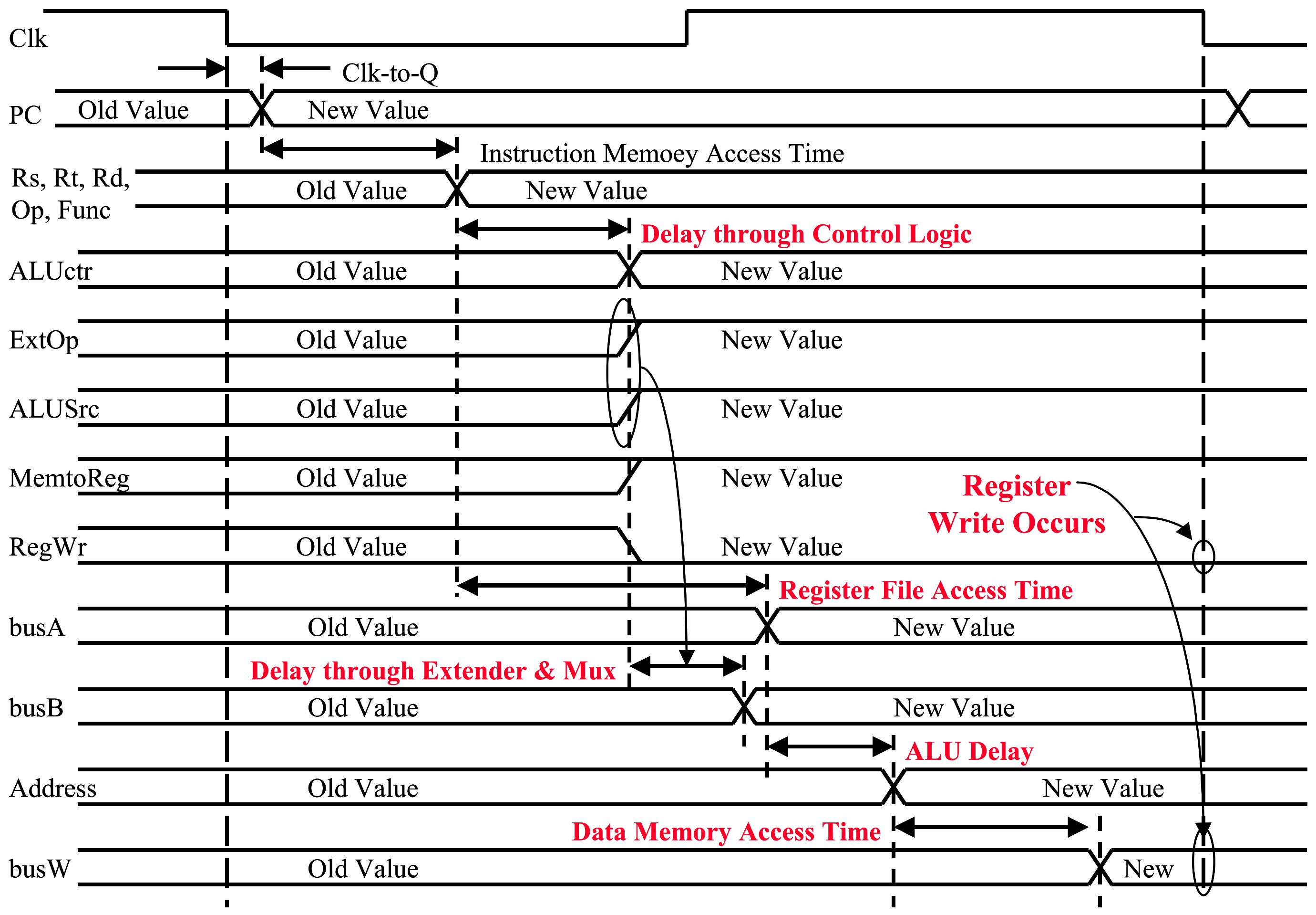
PLA Implementation



A Single Cycle Processor



Worst Case Timing (Load)



Single Cycle Processor: Drawback

Long cycle time: Cycle time must be long enough for the load instruction:

PC's Clock -to-Q + Instruction Memory Access Time + Register File Access Time + ALU Delay (address calculation) + Data Memory Access Time + Register File Setup Time + Clock Skew

Cycle time for load is much longer than needed for all other instructions.

More Problems

- what if we had a more complicated instruction like floating point?
- wasteful of area

One Solution:

- Use a "smaller" cycle time
- Different instructions take different numbers of cycles
- a "multi-cycle" data path

Single-cycle CPU Performance

Example: CPU-units operation time
Memory: 200ps, ALU/Adder:100ps   Reg-File: 50ps

Assume other hardware units have zero delay and following instruction mix.
Loads: 25%, Stores: 10%, ALU instructions: 45%
Branches: 15%, Jumps: 5%
Compare the following two implementations.

- Each instruction operates in 1 clock cycle of a fixed length. (CPI = 1)
- Each instruction executes in 1 clock cycle using a variable-length clock, which for each instruction is only as long as it needs to be (Impractical approach)

For both Implementations: Instruction Count and CPI are same.

When CPI = 1:
CPU EXE-Time = Inst. Count x Clock-Cycle Time

Single-cycle CPU Performance

Critical Path for Variable-Length clock CPU

Instructions	CPU units used by the Inst. Class				
R-type	IF	Reg Access	ALU	Reg Access	
Load	IF	Reg Access	ALU	Mem Access	Reg-Wr
Store	IF	Reg Access	ALU	Mem Access	
Branch	IF	Reg Access	ALU		
Jump	IF				

Instruction Class	Inst. Mem	Reg. Read	ALU Op	Data Mem	Reg. Write	Total
R-type	200	50	100	0	50	400ps
Load	200	50	100	200	50	600ps
Store	200	50	100	200	-	550ps
Branch	200	50	100	0	-	350ps
Jump	200			-	-	200ps

For Variable Clock machine, clock cycle varies from 200ps to 600ps.

Average-time/instruction =
600*.25+550*.1+400*.45+350*.15+200*.05 = 447.5ps

Performance Ratio = 600/447.5 = 1.34

Multi-cycle Approach

- We will be reusing functional units.
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data.
- Our control signals will not be determined solely by instruction
  - e.g., what should the ALU do for a "subtract" instruction?
- We'll use a finite state machine for control