CMPE 411
Computer Architecture

Lecture 10

Single-Cycle Datapath and Control

October 3, 2017

www.csee.umbc.edu/~younis/CMPE411/CMPE411.htm

Lecture's Overview

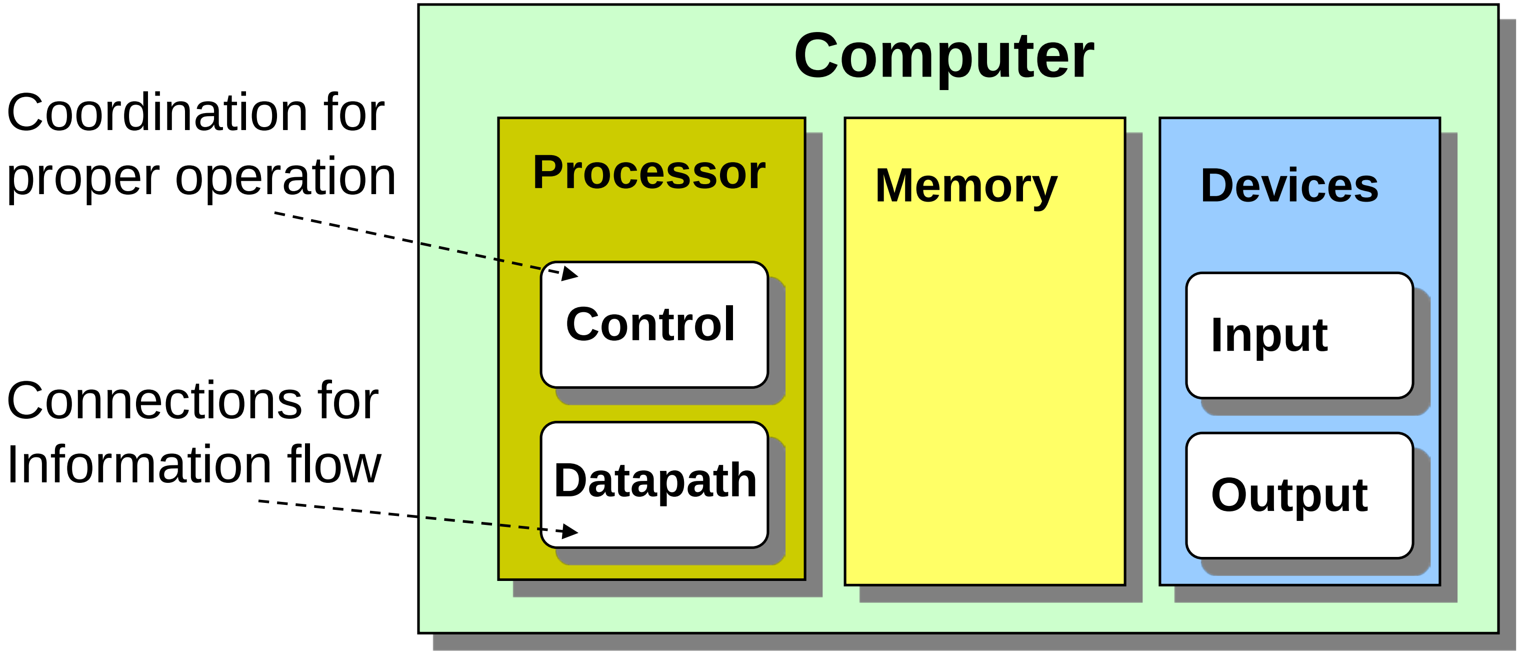
❑ ***Previous Lecture:***

- Representation of floating point numbers
  (Sign, exponent, mantissa, single & double precision, IEEE 754)
- Floating point arithmetic
  (Addition and Multiplication)
- Normalizing Floating point numbers
  (Rounding, zero floating point number, special interpretation)

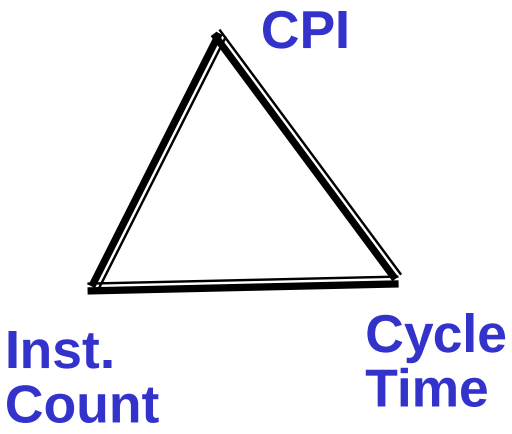
❑ ***This Lecture:***

- Processor design steps
- Building a datapath
- Control unit design
- Assemble a single cycle processor

Introduction


Computer
Coordination for proper operation
Connections for Information flow
Processor
Control
Datapath
Memory
Devices
Input
Output

- ❑ We studied the user prospective: instruction set architecture, performance
- ❑ Performance of a machine is determined by:
  - ➔ Instruction count
  - ➔ Clock cycle time
  - ➔ Clock cycles per instruction
- ❑ Processor design (datapath and control) will determine:
  - ➔ Clock cycle time
  - ➔ Clock cycles per instruction



How to Design a Processor: step-by-step

1. Analyze instruction set => datapath requirements
   - the meaning of each instruction is given by the *register transfers*
   - datapath must include storage element for ISA registers possibly more
   - datapath must support each register transfer
2. Select a set of datapath components and establish clocking methodology
3. Assemble datapath that meets the requirements
4. Analyze the implementation of each instruction to determine setting of control points that affects the register transfer
5. Assemble the control logic

* Slide is courtesy of Dave Patterson

The MIPS Instruction Formats

- All MIPS instructions are 32 bits, in one of three formats:

R-type

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

I-type

31 26 21 16 0

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

J-type

31 26 0

op	target address
6 bits	26 bits

- The different fields are:
  - op: operation of the instruction
  - rs, rt, rd: the source and destination register specifiers
  - shamt: shift amount
  - funct: selects the variant of the operation in the “op” field
  - address / immediate: address offset or immediate value
  - target address: target address of the jump instruction

* Slide is courtesy of Dave Patterson

Step 1a: The instruction Subset for today

- ❑ ADD and SUB
  - ➔ add rd, rs, rt
  - ➔ sub rd, rs, rt

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

- ❑ OR Immediate:
  - ➔ ori rt, rs, imm16

31 26 21 16 0

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- ❑ LOAD & STORE Word
  - ➔ lw rt, rs, imm16
  - ➔ sw rt, rs, imm16

31 26 21 16 0

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- ❑ BRANCH:
  - ➔ beq rs, rt, imm16

31 26 21 16 0

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

* Slide is courtesy of Dave Patterson

Logical Register Transfers

- ❑ Logical register transfer gives the meaning of the instructions
- ❑ All start by fetching the instruction

op | rs | rt | rd | shamt | funct = MEM[PC]

op | rs | rt | Imm16 = MEM[PC]

inst.	**Register Transfers**
ADD	**R[rd] ⇐ R[rs] + R[rt]; PC ⇐ PC + 4**
SUB	**R[rd] ⇐ R[rs] – R[rt]; PC ⇐ PC + 4**
ORi	**R[rt] ⇐ R[rs] + zero_ext(Imm16); PC ⇐ PC + 4**
LW	**R[rt] ⇐ MEM[R[rs] + sign_ext(Imm16)]; PC ⇐ PC + 4**
SW	**MEM[R[rs] + sign_ext(Imm16)] ⇐ R[rt]; PC ⇐ PC + 4**

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

* Slide is courtesy of Dave Patterson

Back to Processor Design

Step 1: Requirements of the Instruction Set

- ❑ Memory: instruction & data
- ❑ Registers (32 x 32): read RS, read RT, Write RT or RD
- ❑ Program Counter
- ❑ Extender
- ❑ Add and Sub register or extended immediate
- ❑ Add 4 or the extended immediate to PC

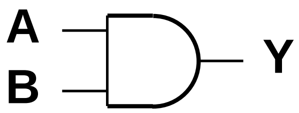
Step 2: Components of the Datapath

- ❑ Combinational Elements
- ❑ Storage Elements
  - ➔ Clocking methodology

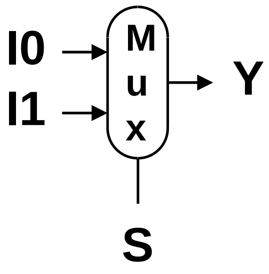
* Slide is courtesy of Dave Patterson

Combinational Elements

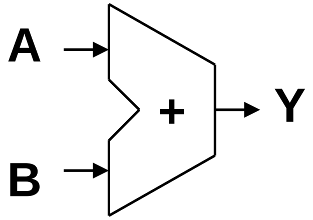
- AND-gate
  - Y = A & B



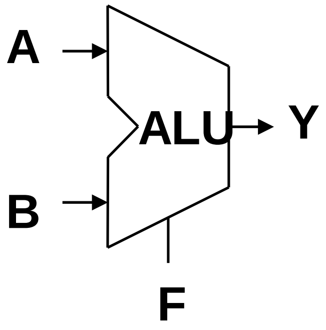
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B



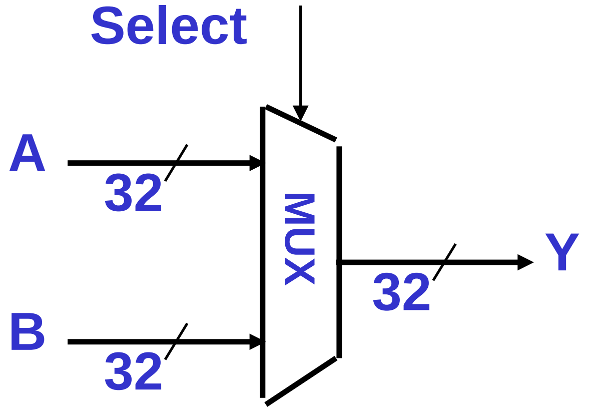
- Arithmetic/Logic Unit
  - Y = F(A, B)

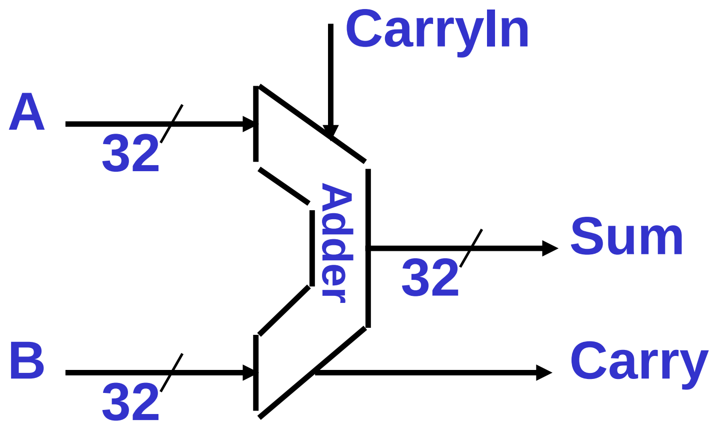


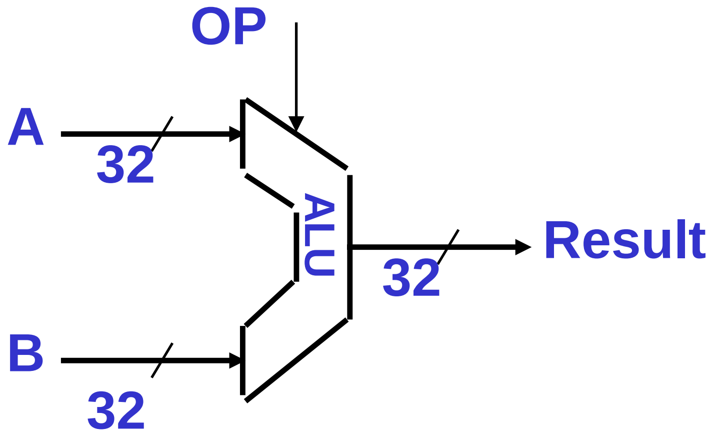
* Slide is courtesy of Dave Patterson

Combinational Logic Elements

Basic Building Blocks





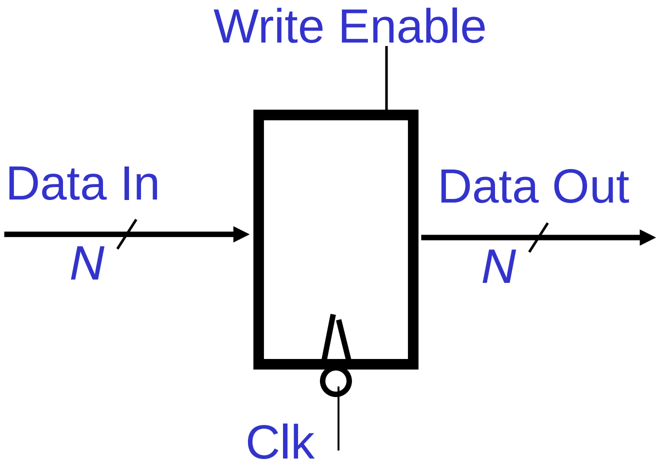


* Figures are courtesy of Dave Patterson

Storage Elements

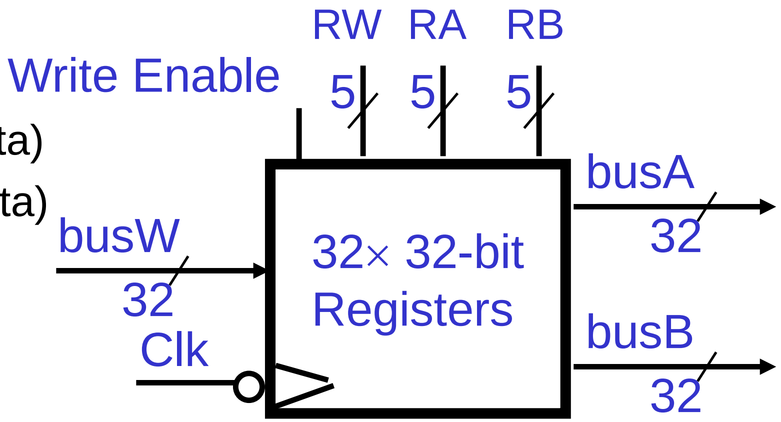
Register:

- ❑ Similar to the D Flip Flop except
  - ➔ N-bit input and output
  - ➔ Write Enable input
- ❑ Write Enable:
  - ➔ negated (0): Data Out will not change
  - ➔ asserted (1): Data Out will become Data In



Register File:

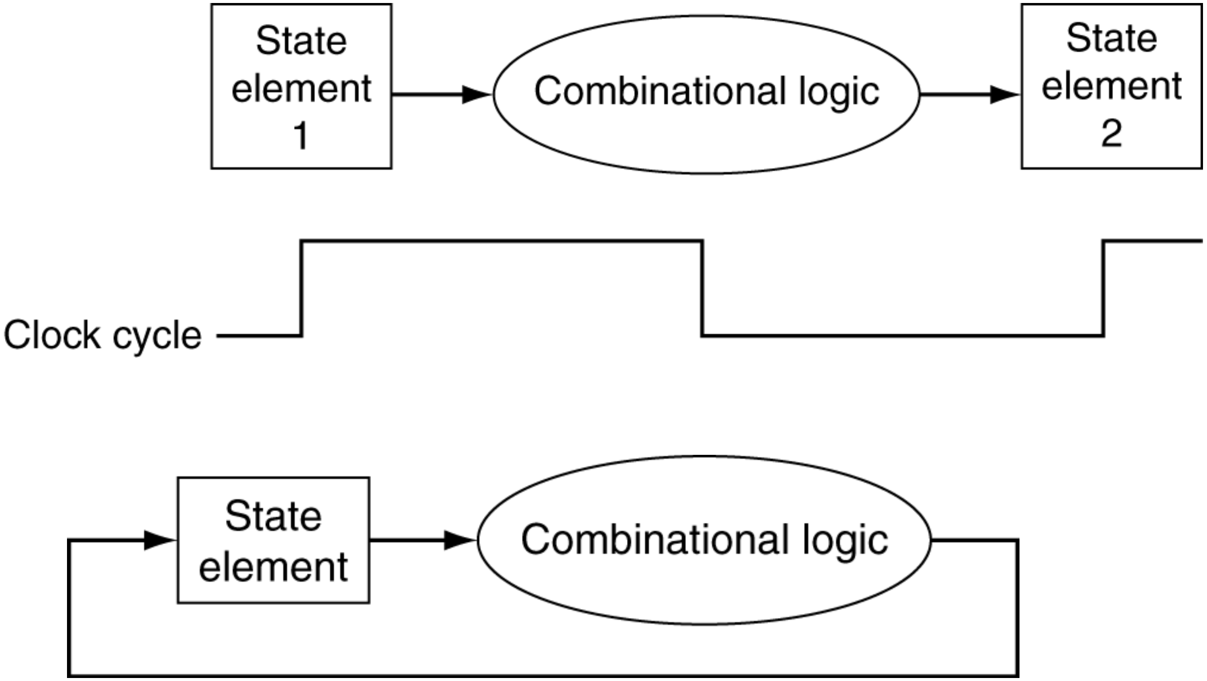
- ❑ Consists of 32 registers:
  - ➔ Two 32-bit output busses: busA and busB
  - ➔ One 32-bit input bus: busW
- ❑ A register is selected by:
  - ➔ RA (number) selects a register to put on busA (data)
  - ➔ RB (number) selects a register to put on busB (data)
  - ➔ RW (number) selects a register to be written via busW (data) when Write Enable is 1
- ❑ Clock input (CLK)
  - ➔ The CLK input is a factor ONLY during write operation
  - ➔ During read operation, behaves as a combinational logic block: RA or RB valid => busA or busB valid after "access time."



* Slide is courtesy of Dave Patterson

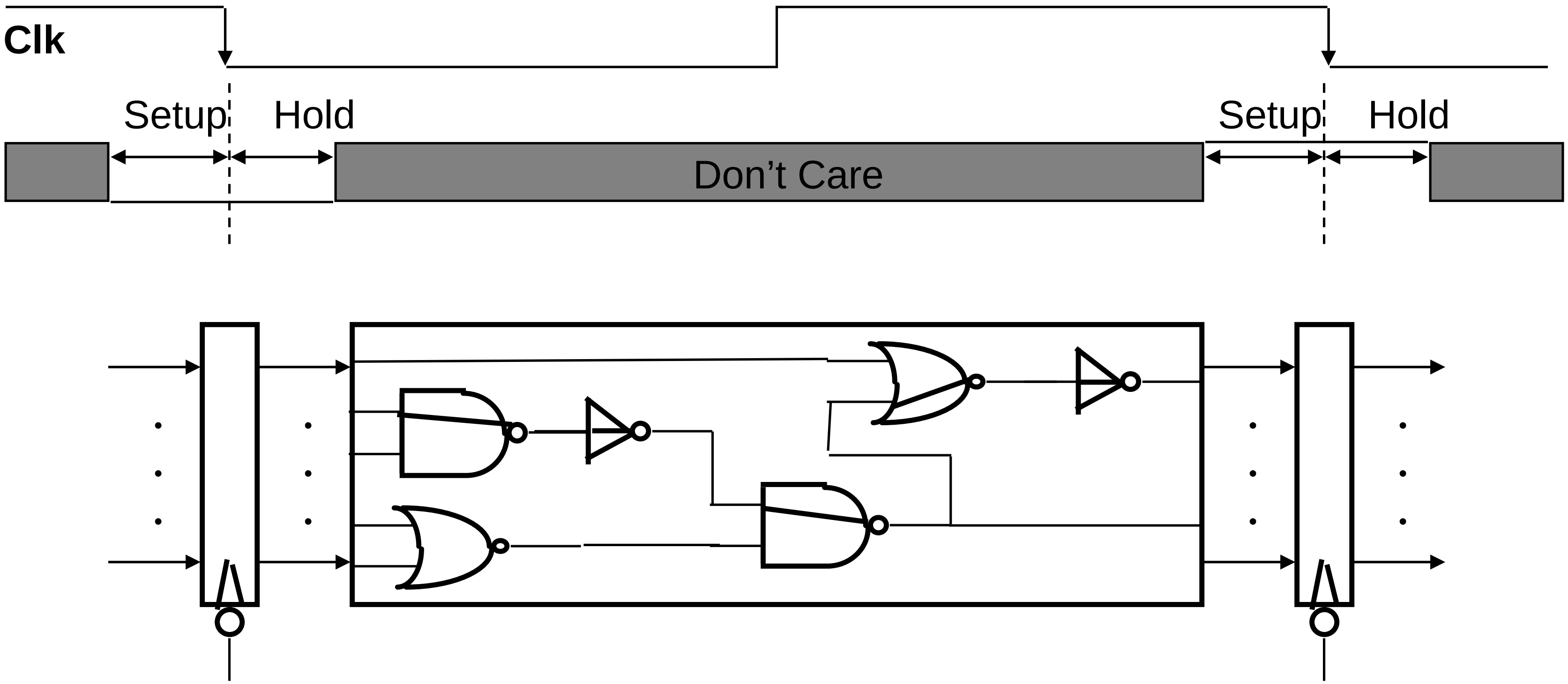
Clocking Methodology

- ❑ Combinational logic transforms data during clock cycles
  - ➔Between clock edges
  - ➔Input from state elements, output to state element
  - ➔Longest delay determines clock period



* Slide is courtesy of Dave Patterson

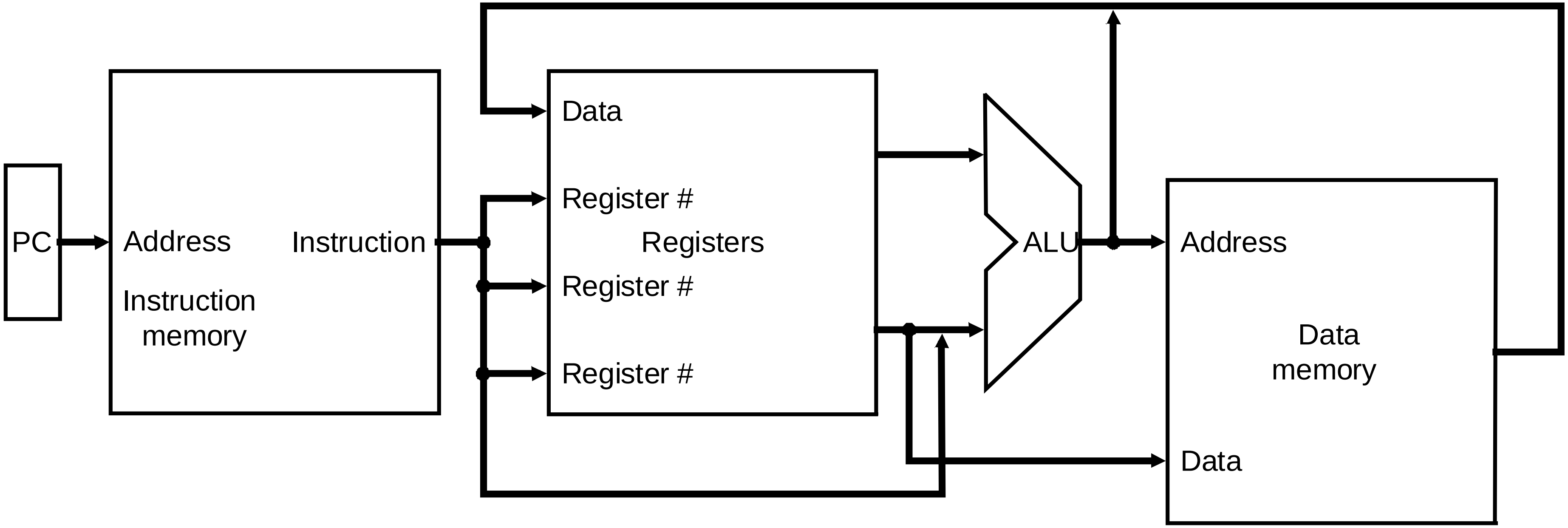
Clocking Methodology



- ❑ All storage elements are clocked by the same clock edge
- ❑ Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew
- ❑ (CLK-to-Q + Shortest Delay Path - Clock Skew) > Hold Time

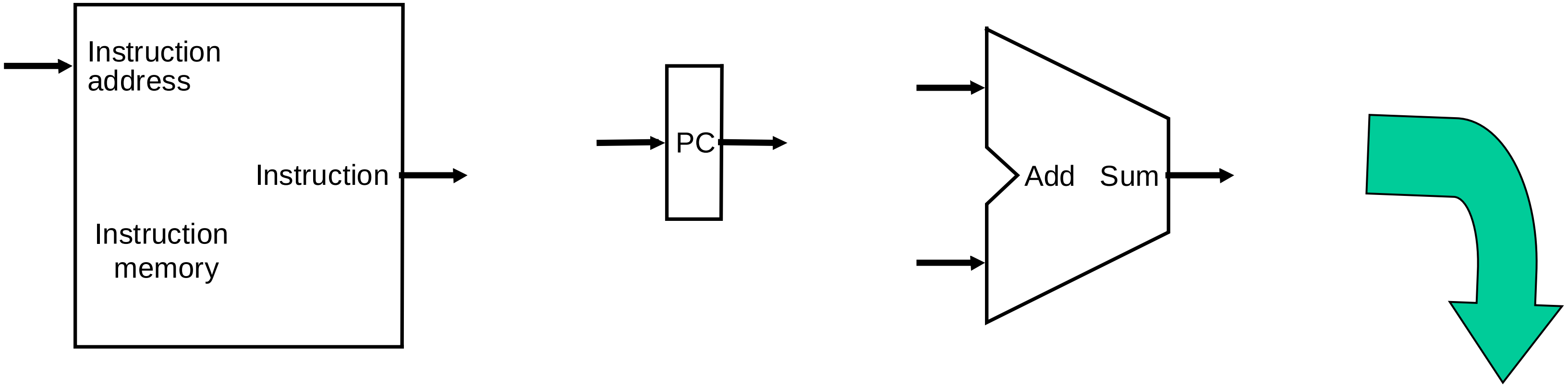
* Slide is courtesy of Dave Patterson

Step 3: Datapath Assembly



- ❑ Register Transfer Requirements ☑ Datapath Assembly
- ❑ Datapath should support:
  - ➔ Instruction fetch
  - ➔ Operands reading
  - ➔ Operation execution

Instruction Fetch Unit



a. Instruction memory

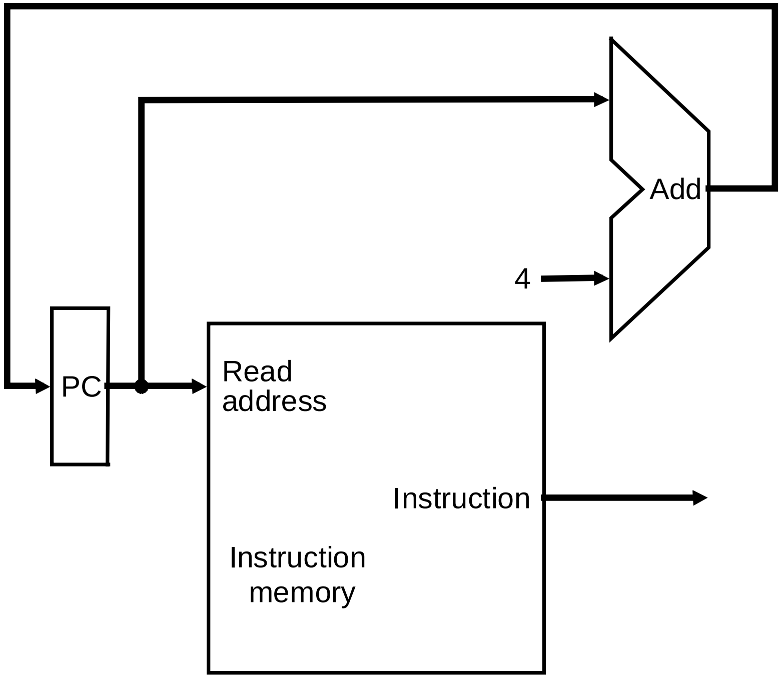
b. Program counter

c. Adder

- ❑ Fetch the Instruction: mem[PC]
- ❑ Update the program counter:
  - ➔ Sequential Code:

    PC <- PC + 4
  - ➔ Branch and Jump:

    PC <- “something else”

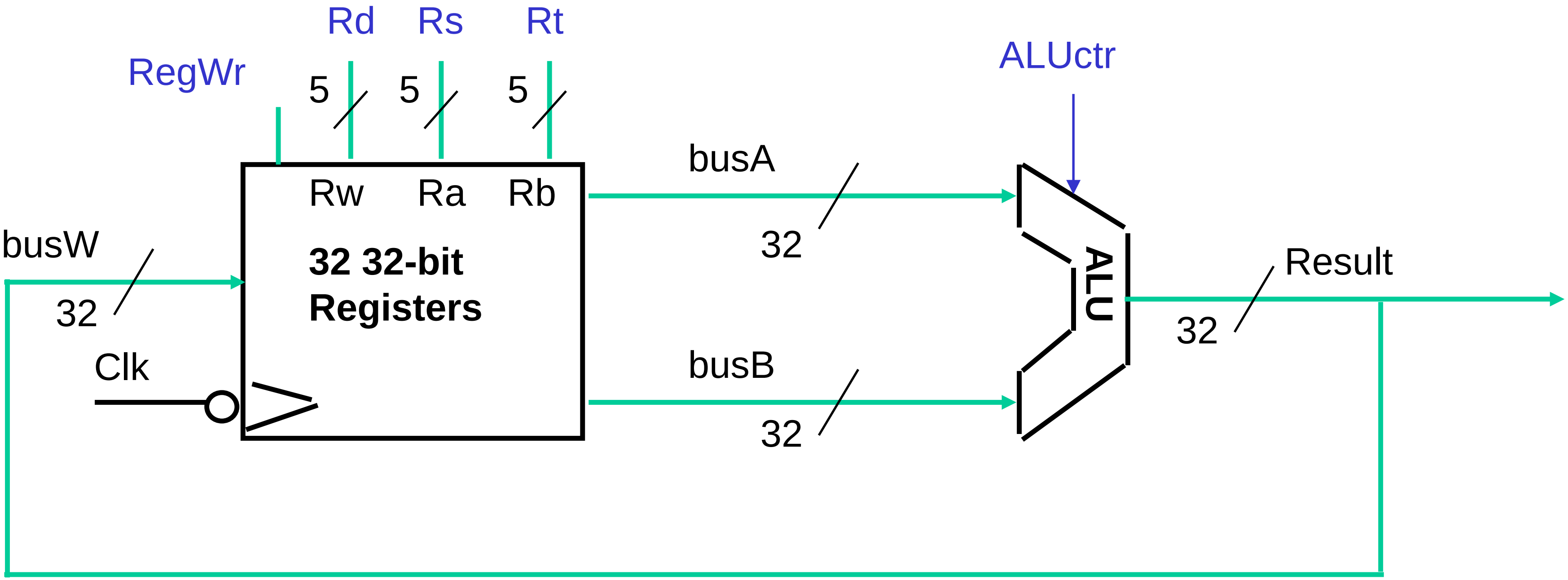


Supporting Add & Subtract

R[rd] ⇐ R[rs] op R[rt]   Example: add   rd, rs, rt

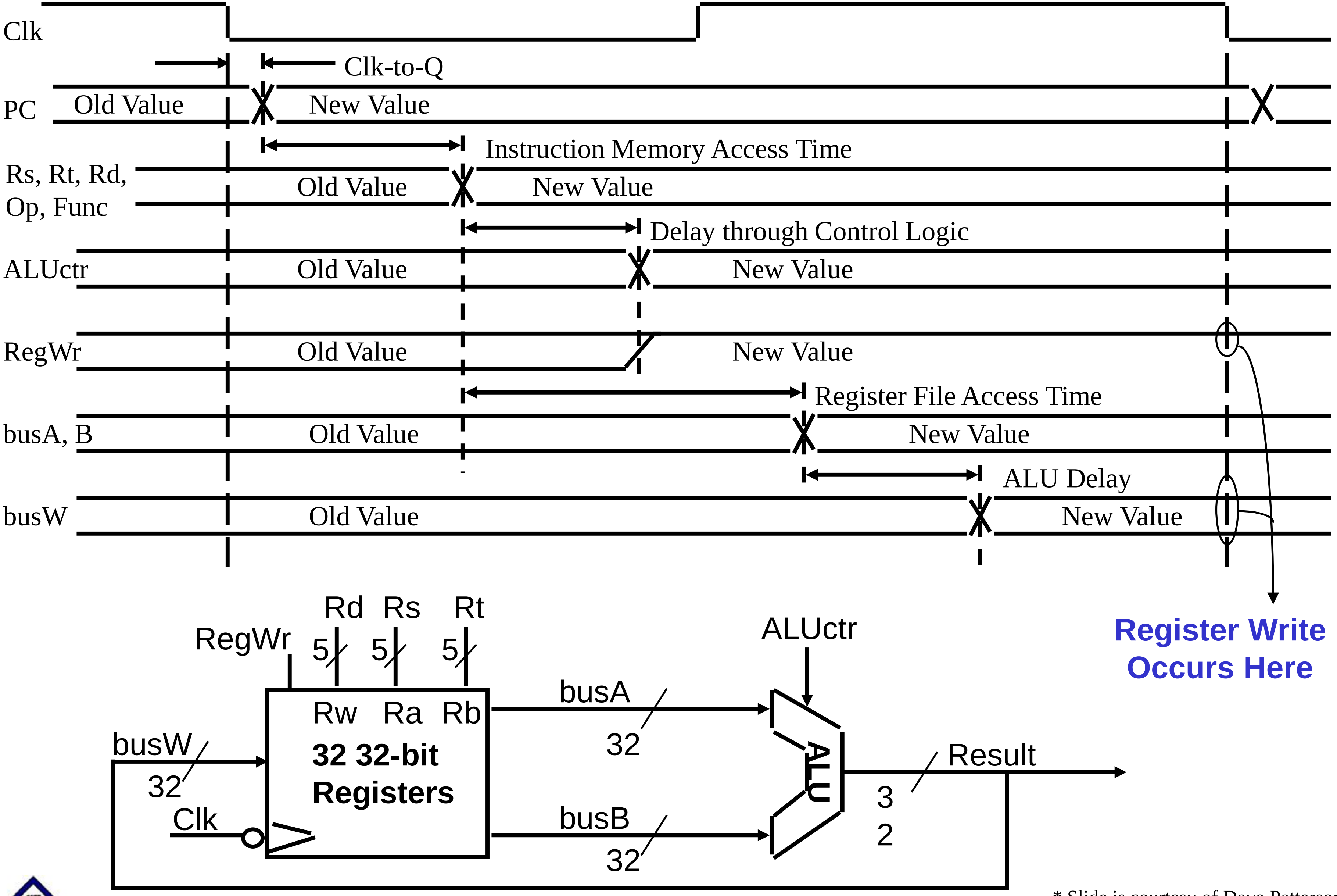
- ➔ Ra, Rb, and Rw come from instruction's rs, rt, and rd fields
- ➔ ALUctr and RegWr: control logic after decoding the instruction

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	



* Slide is courtesy of Dave Patterson

Register-Register Timing



Register Write Occurs Here

* Slide is courtesy of Dave Patterson

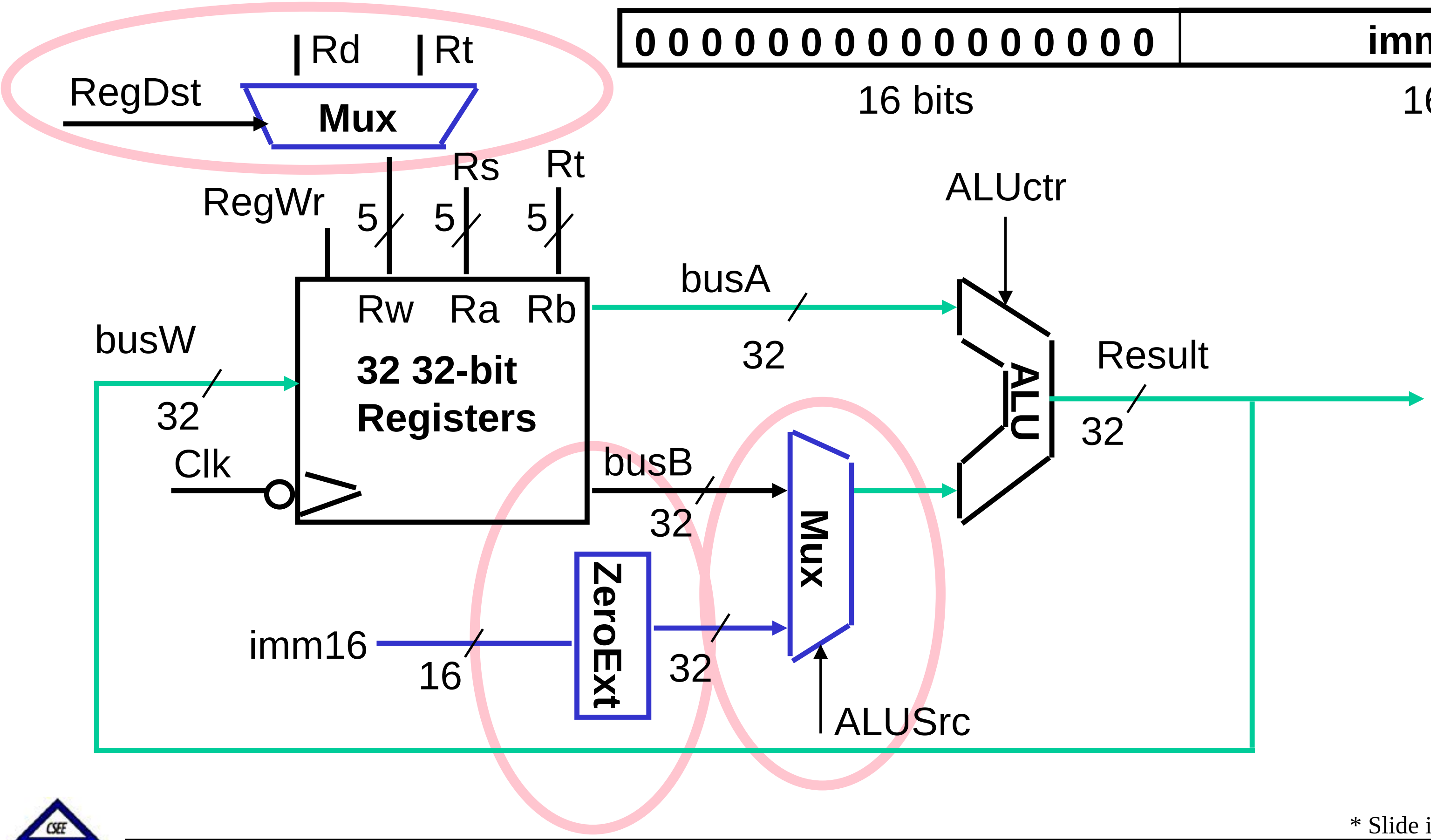
Logical Operations with Immediate

R[rt] ⇐ R[rs] op ZeroExt[imm16]

31 26	21	16	0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits

rd?

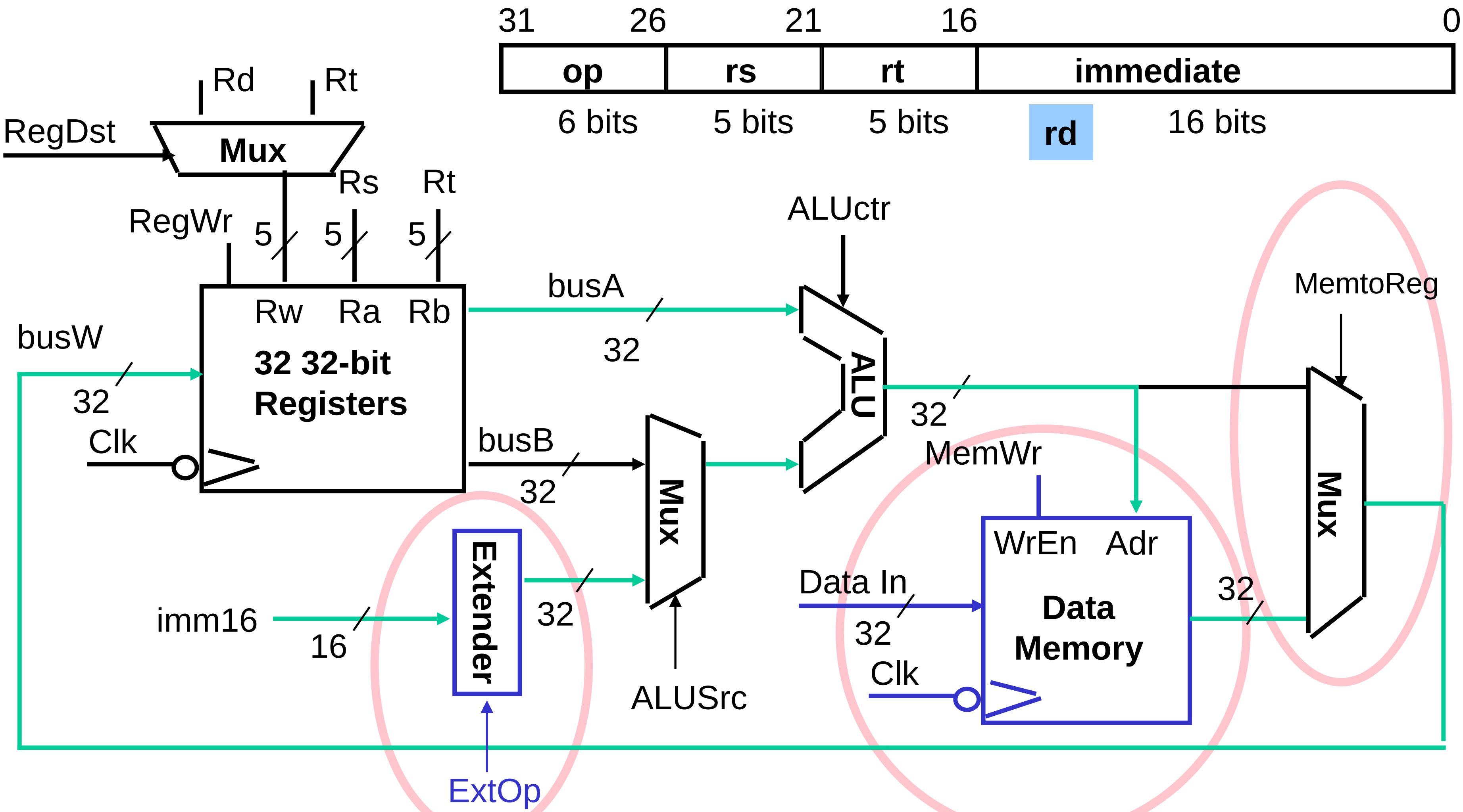
31 16	15 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	**immediate**
16 bits	16 bits



* Slide is courtesy of Dave Patterson

Supporting Load Operations

R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: lw rt, rs, imm16



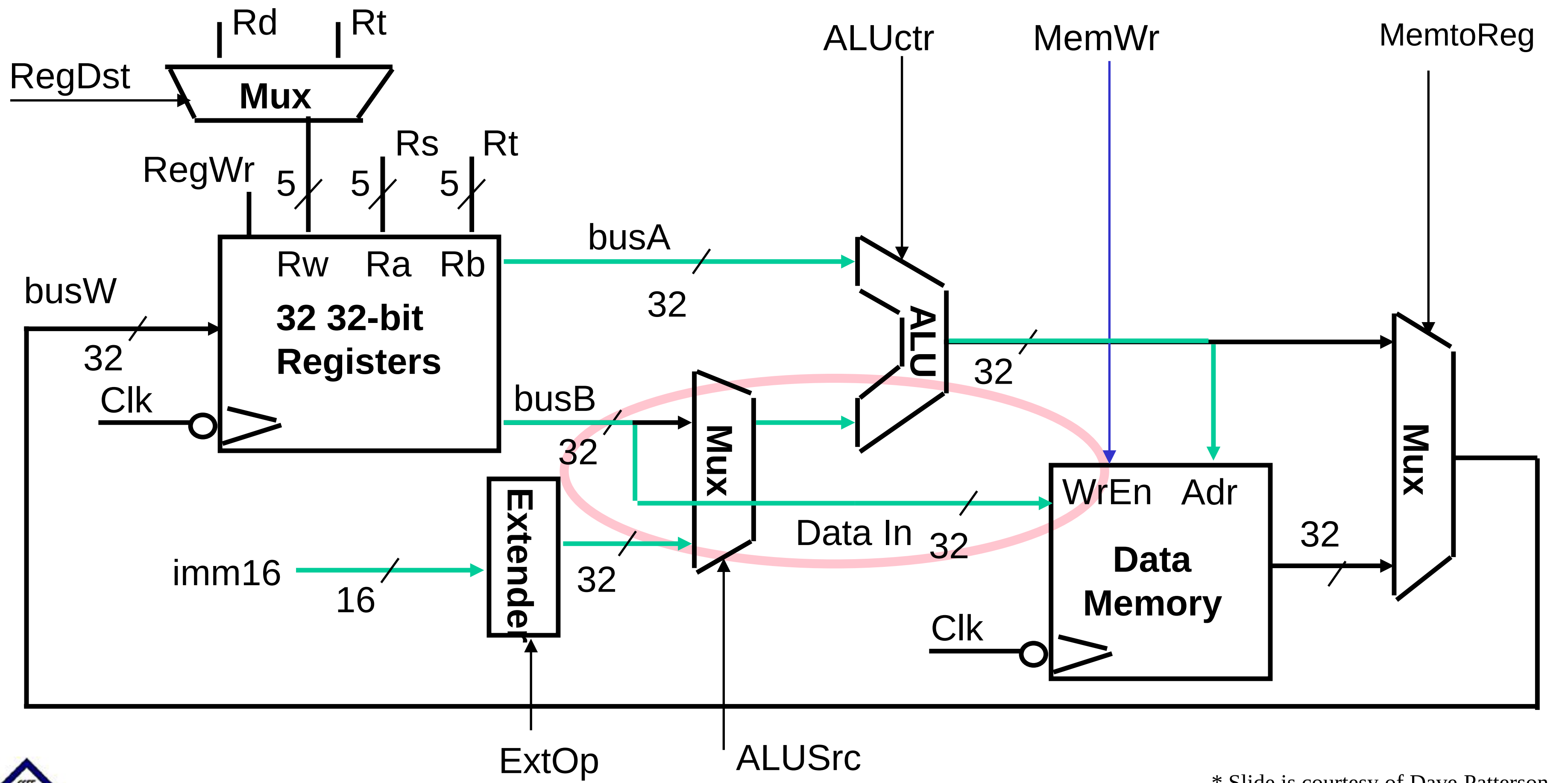
* Slide is courtesy of Dave Patterson

Supporting Store Operations

Mem[R[rs] + SignExt[imm16] <- R[rt]]   Example: sw   rt, rs, imm16

31 26 21 16 0

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits



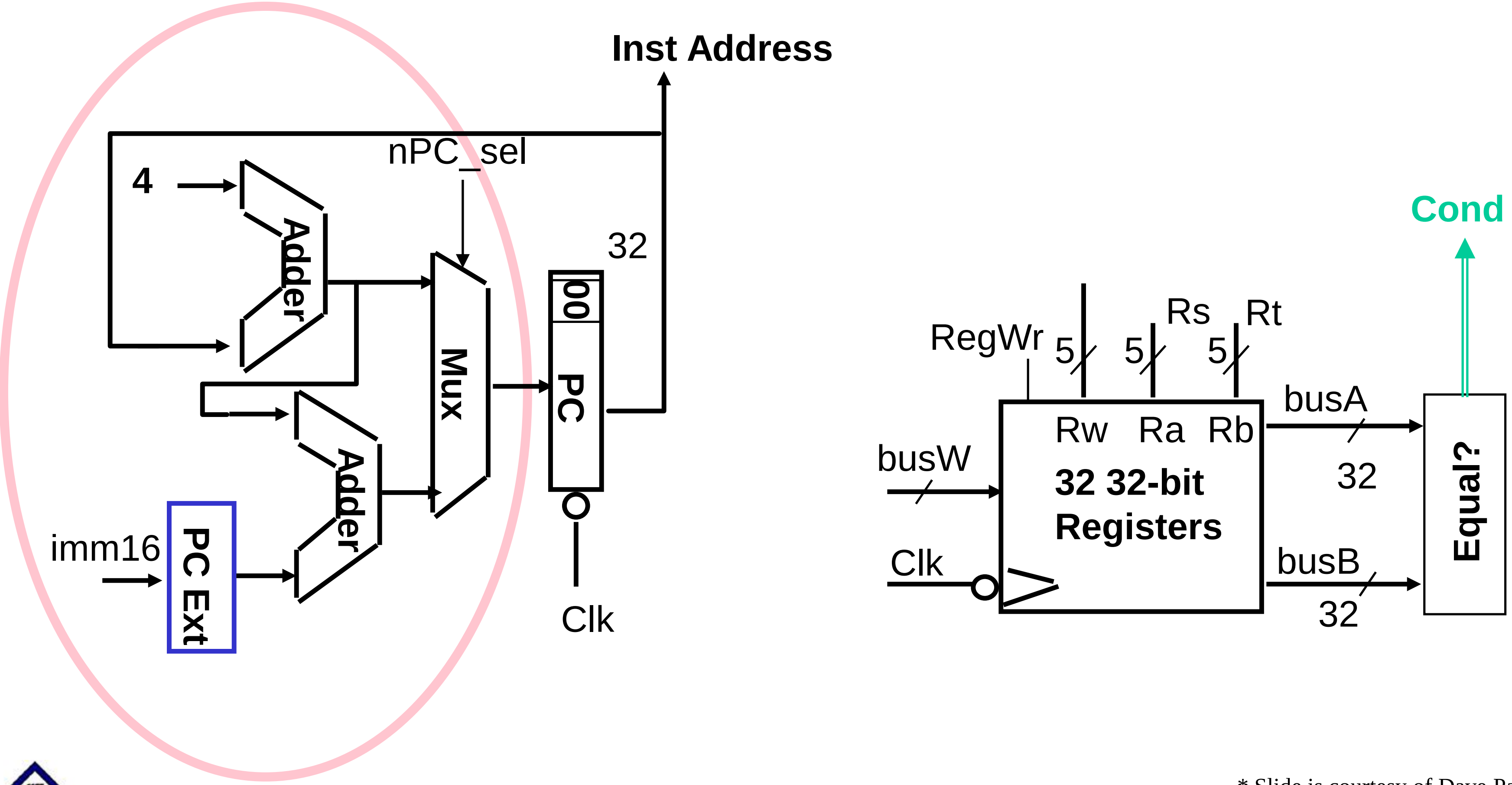
* Slide is courtesy of Dave Patterson

Datapath for Branch Operations

beq rs, rt, imm16

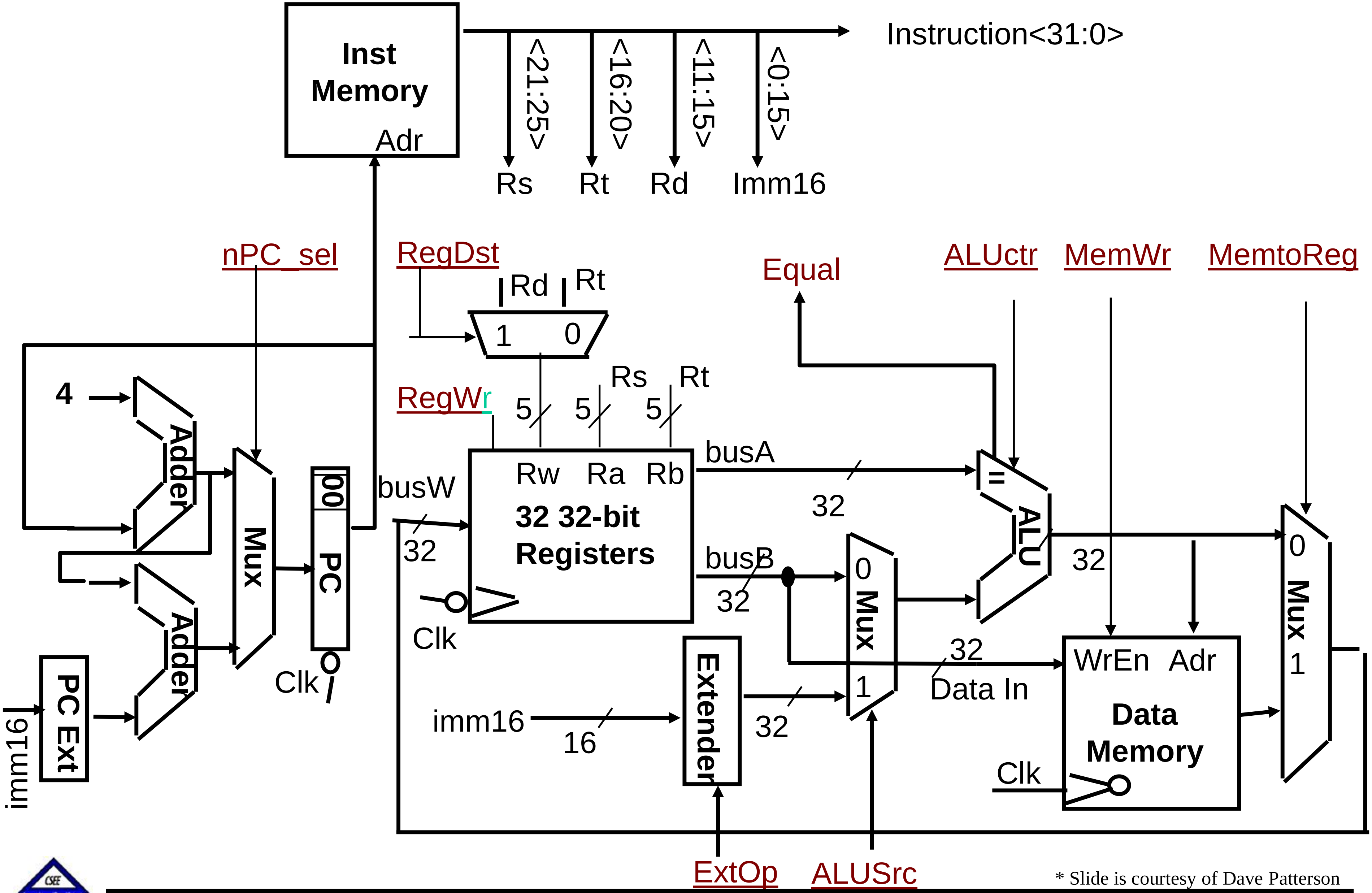
Datapath generates condition (equal)

31	26	21	16	0
op	**rs**	**rt**	**immediate**	
6 bits	5 bits	5 bits	16 bits	



* Slide is courtesy of Dave Patterson

Processor Datapath



* Slide is courtesy of Dave Patterson

Critical Path

Register file and ideal memory:

- ➔ The CLK input is a factor ONLY during write operation
- ➔ During read operation, behave as combinational logic:
- ➔ Address valid ⇒ Output valid after "access time"

<u>Critical Path</u> (Load Operation) =

PC's Clk-to-Q +

Instruction Memory's Access Time +

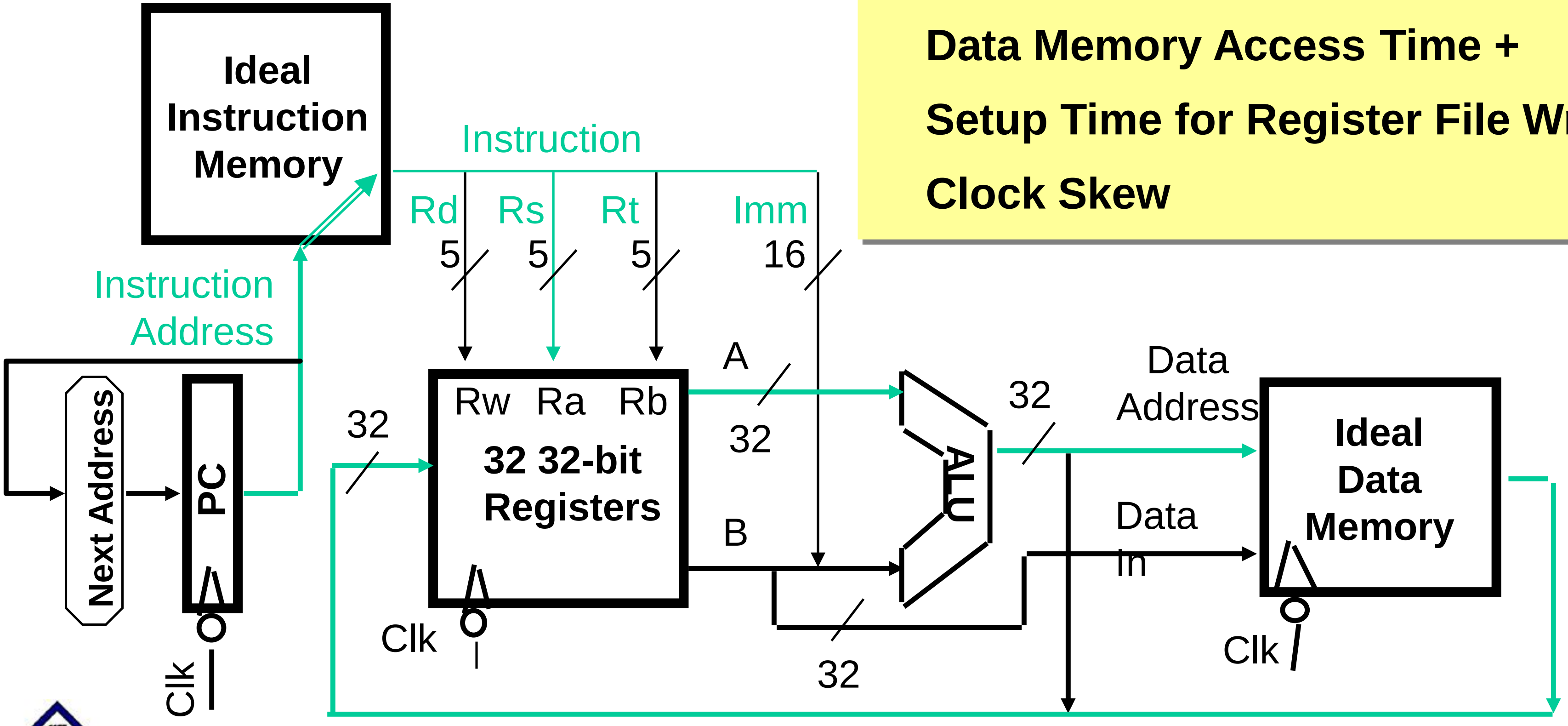
Register File's Access Time +

ALU to Perform a 32-bit Add +

Data Memory Access Time +

Setup Time for Register File Write +

Clock Skew



Summary

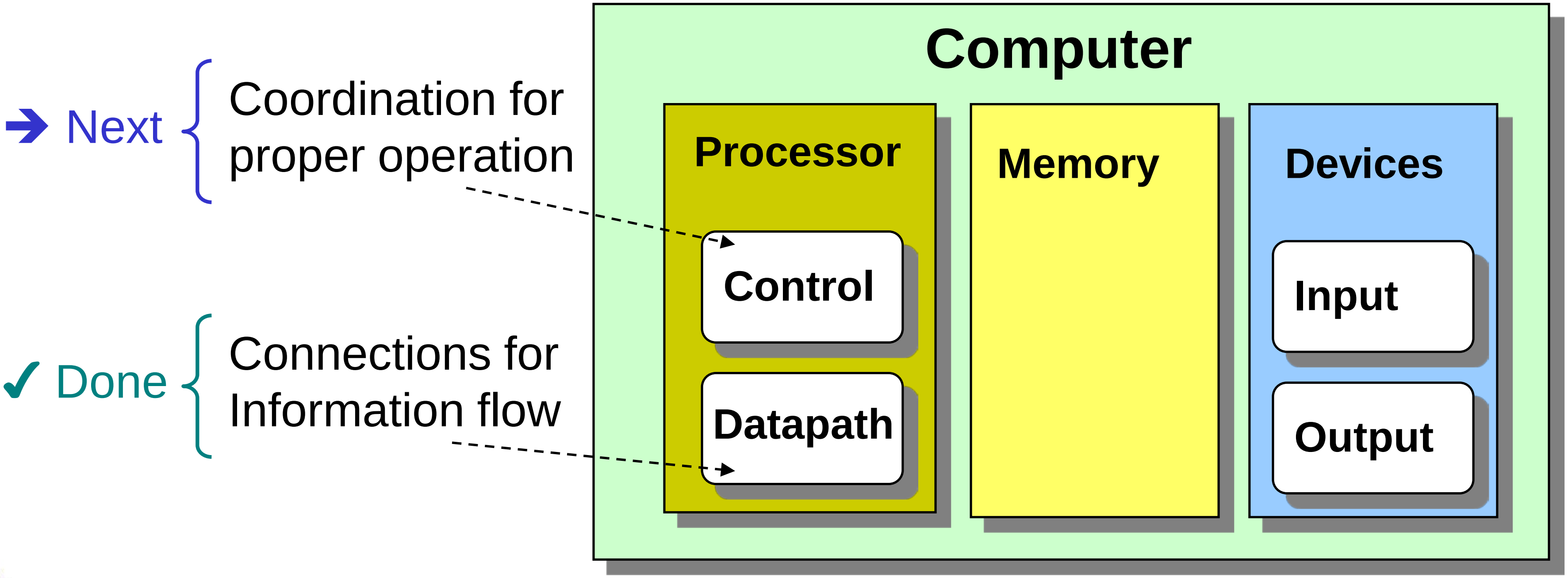
Design Steps:

1. Analyze instruction set => datapath requirements
2. Select set of datapath components and establish clocking methodology
3. Assemble datapath meeting the requirements

✓ done so far

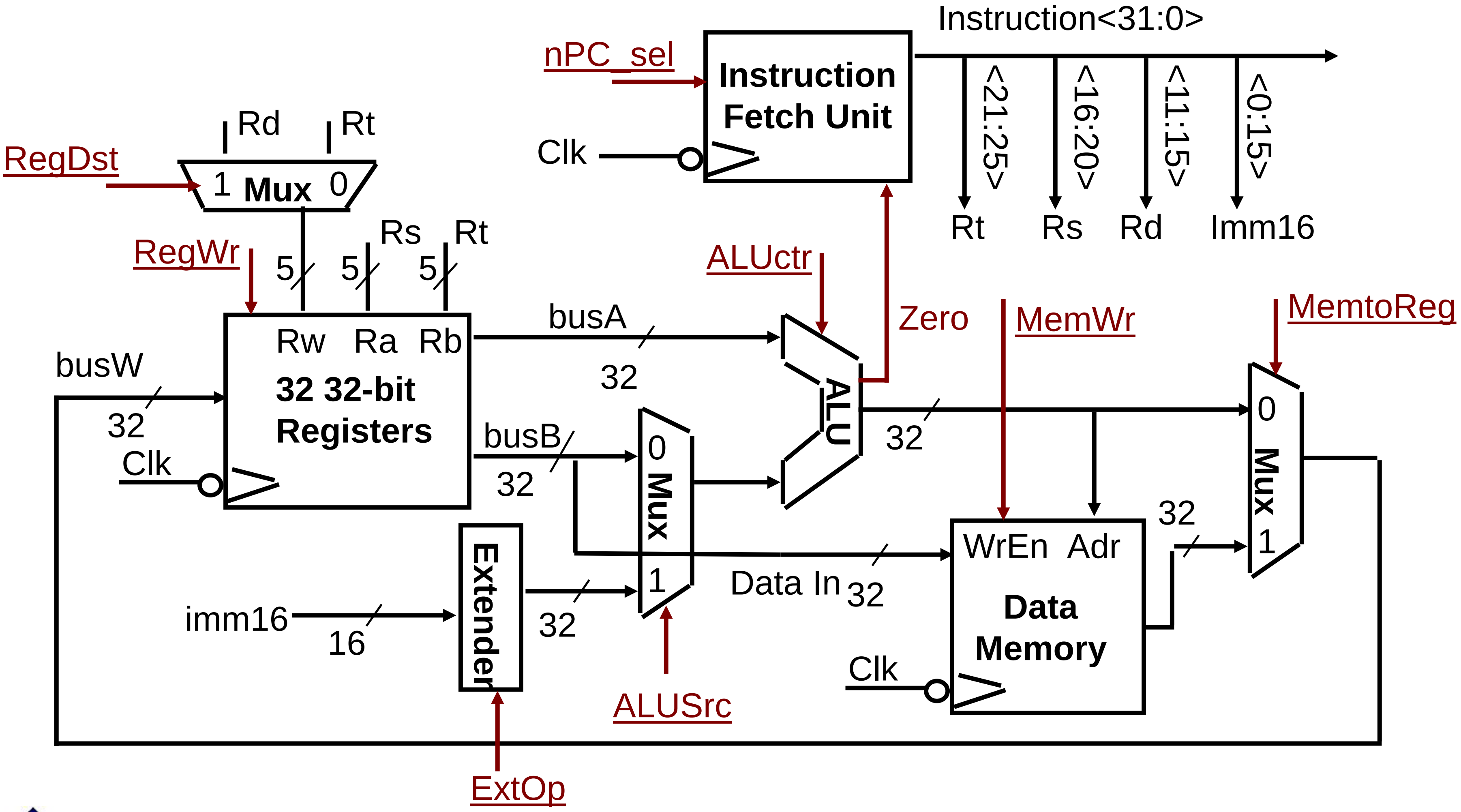
4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer
5. Assemble the control logic

➔ Next



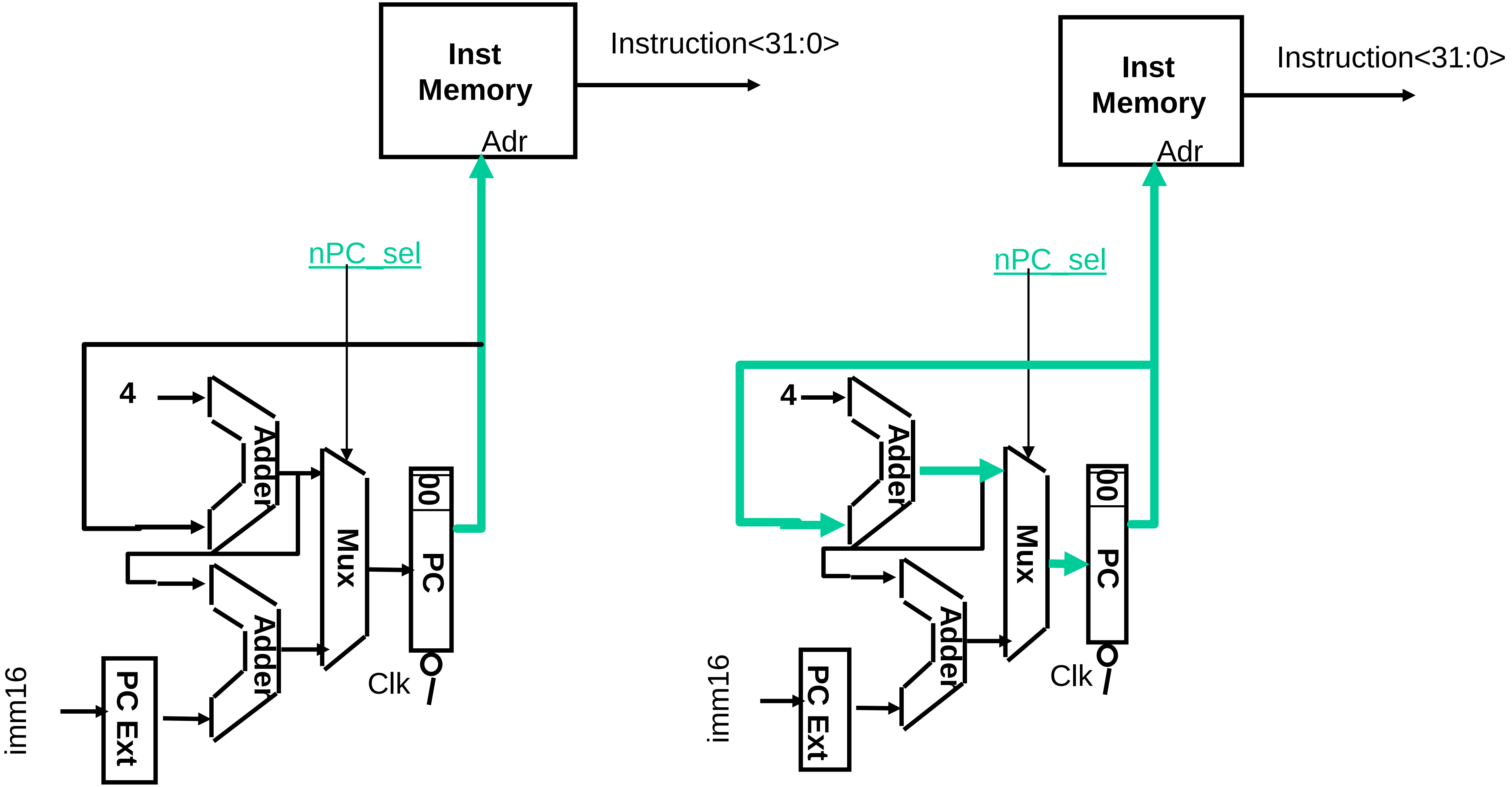
Single-cycle Datapath

❑ Today’s lecture will show you how to generate the control signals (underline)



* Slide is courtesy of Dave Patterson

Instruction Fetch Unit



Instruction ⇐ mem[PC]

same for all instructions

PC ⇐ PC + 4

same for all instructions except: Branch & Jump

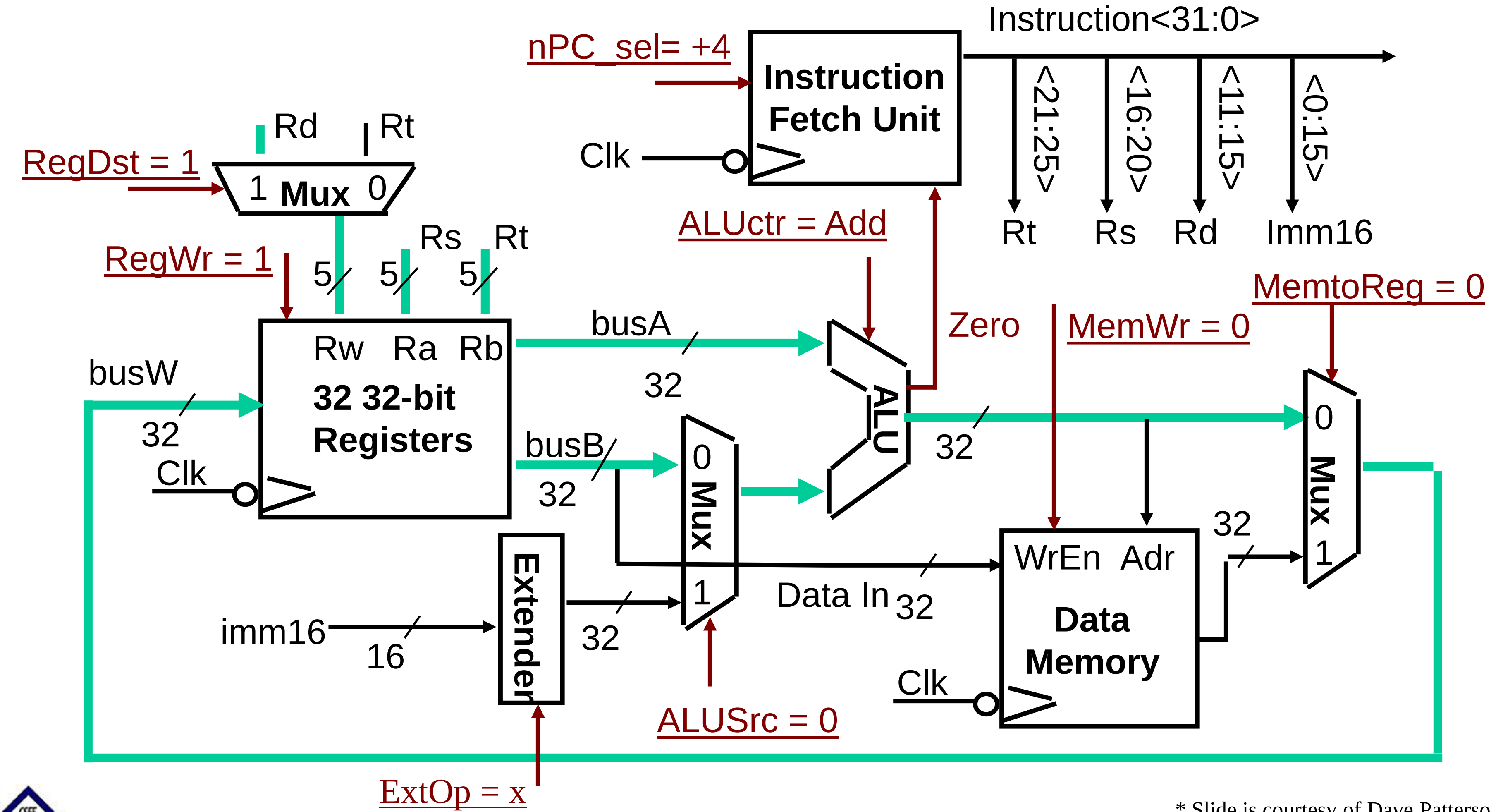
* Figures are courtesy of Dave Patterson

Single Cycle Datapath during Add

31 26 21 16 11 6 0

op	rs	rt	rd	shamt	funct

R[rd] ⇐ R[rs] + R[rt]

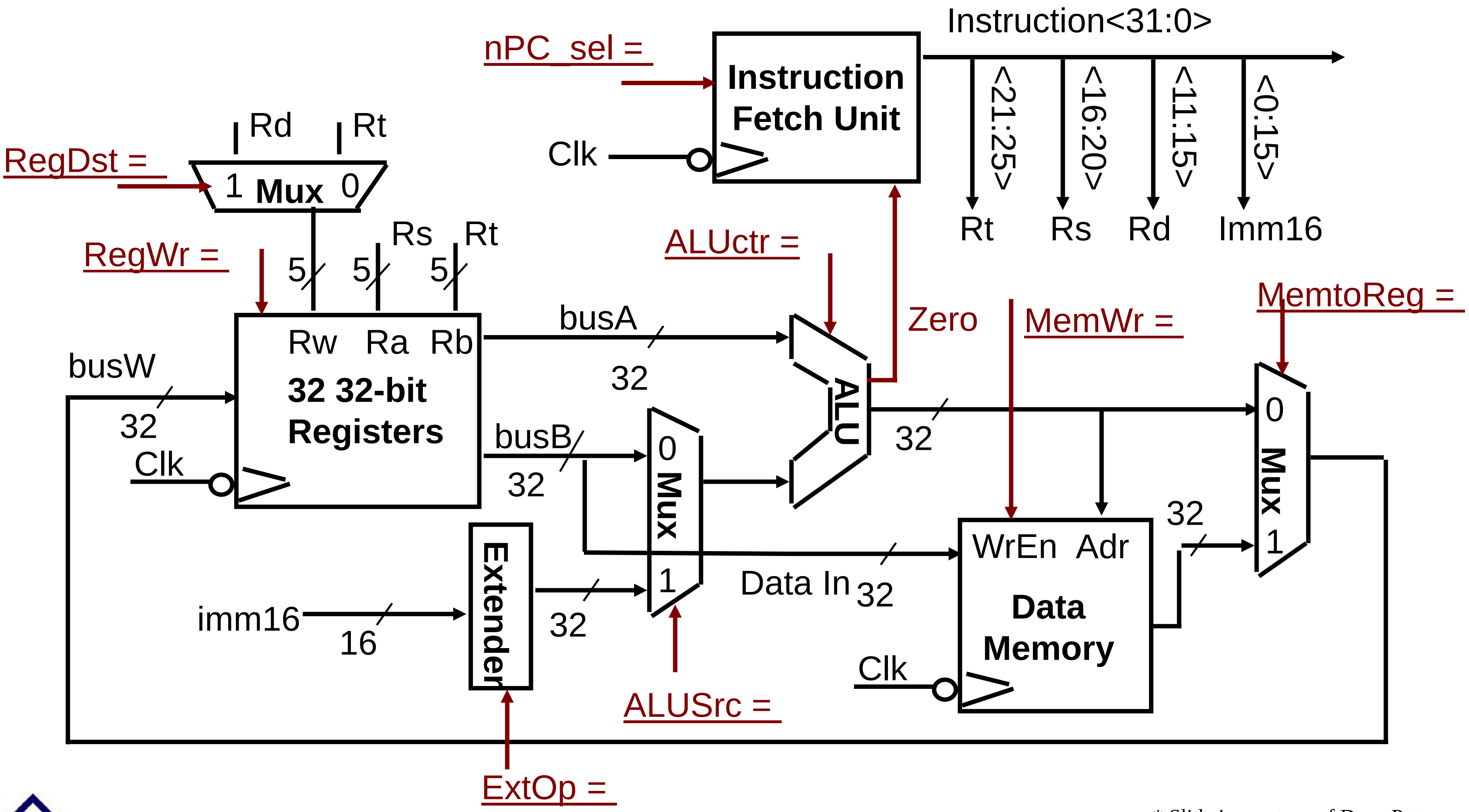


* Slide is courtesy of Dave Patterson

Datapath during Or Immediate

31	26	21	16	0
op	rs	rt	immediate	

R[rt] ⇐ R[rs] or ZeroExt[Imm16]

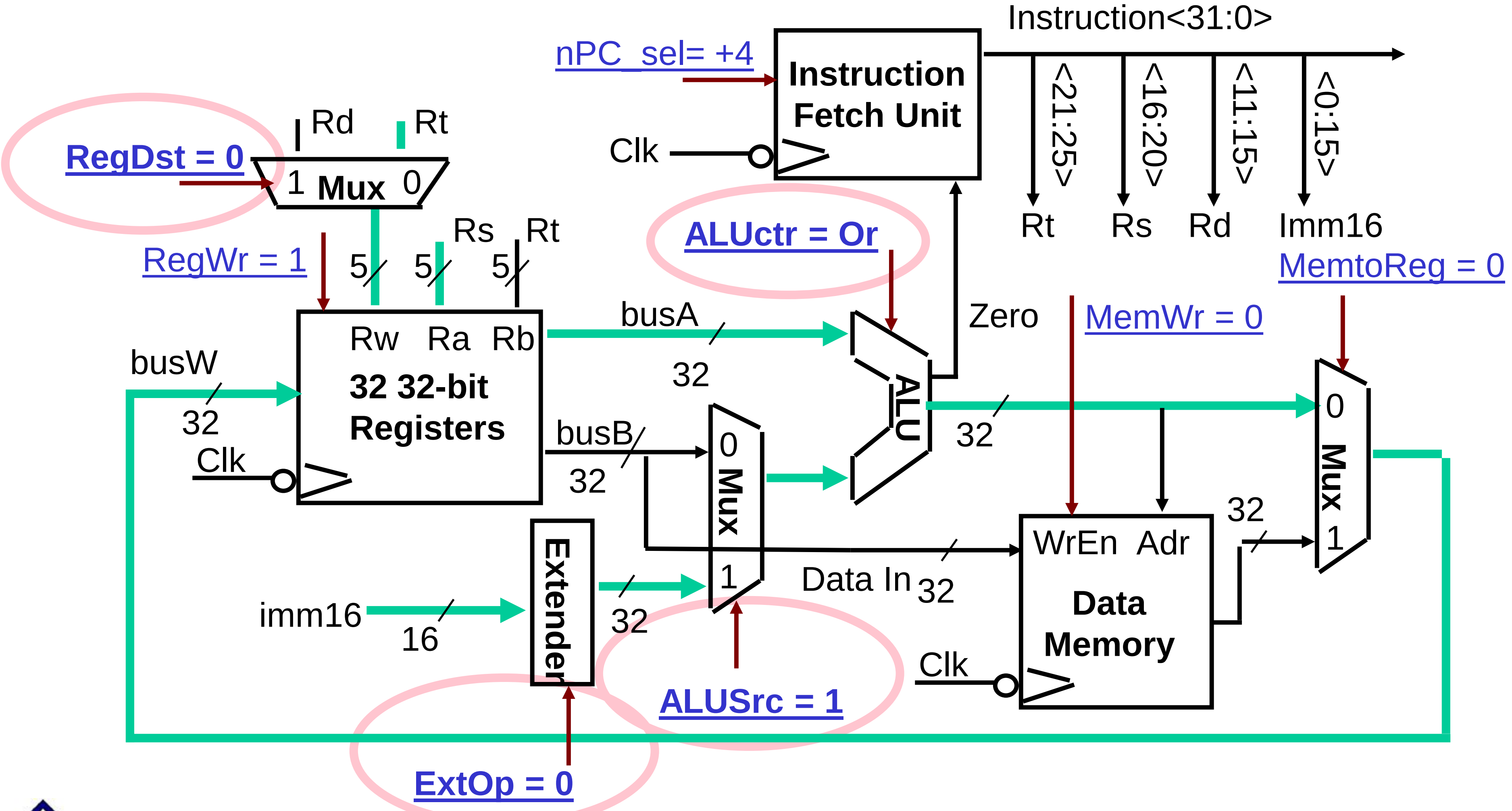


* Slide is courtesy of Dave Patterson

Datapath during Or Immediate

31	26	21	16	0
op	rs	rt	immediate	

R[rt] ⇐ R[rs] or ZeroExt[Imm16]

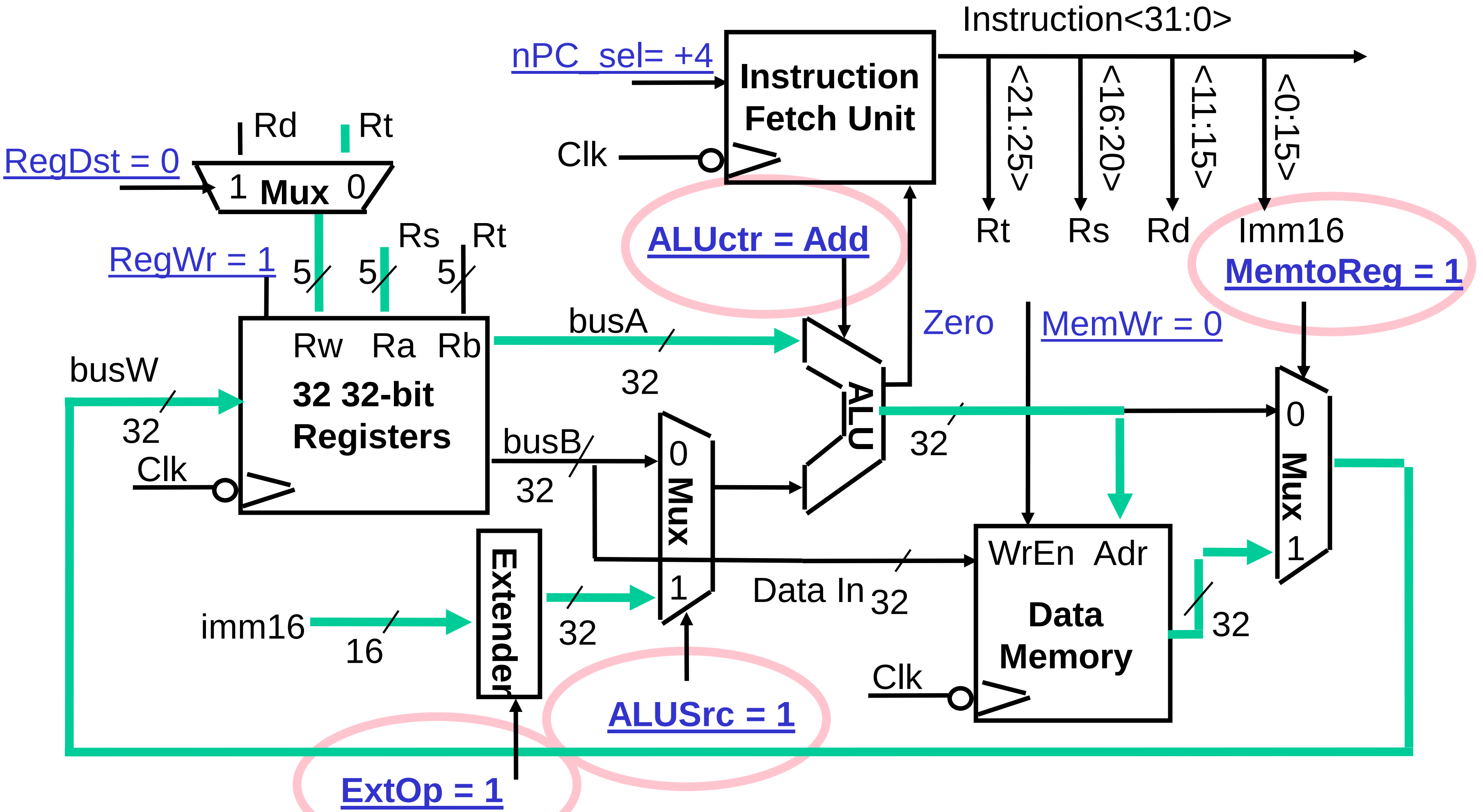


* Slide is courtesy of Dave Patterson

Single Cycle Datapath during Load

31 26	26 21	21 16	16 0
op	rs	rt	immediate

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

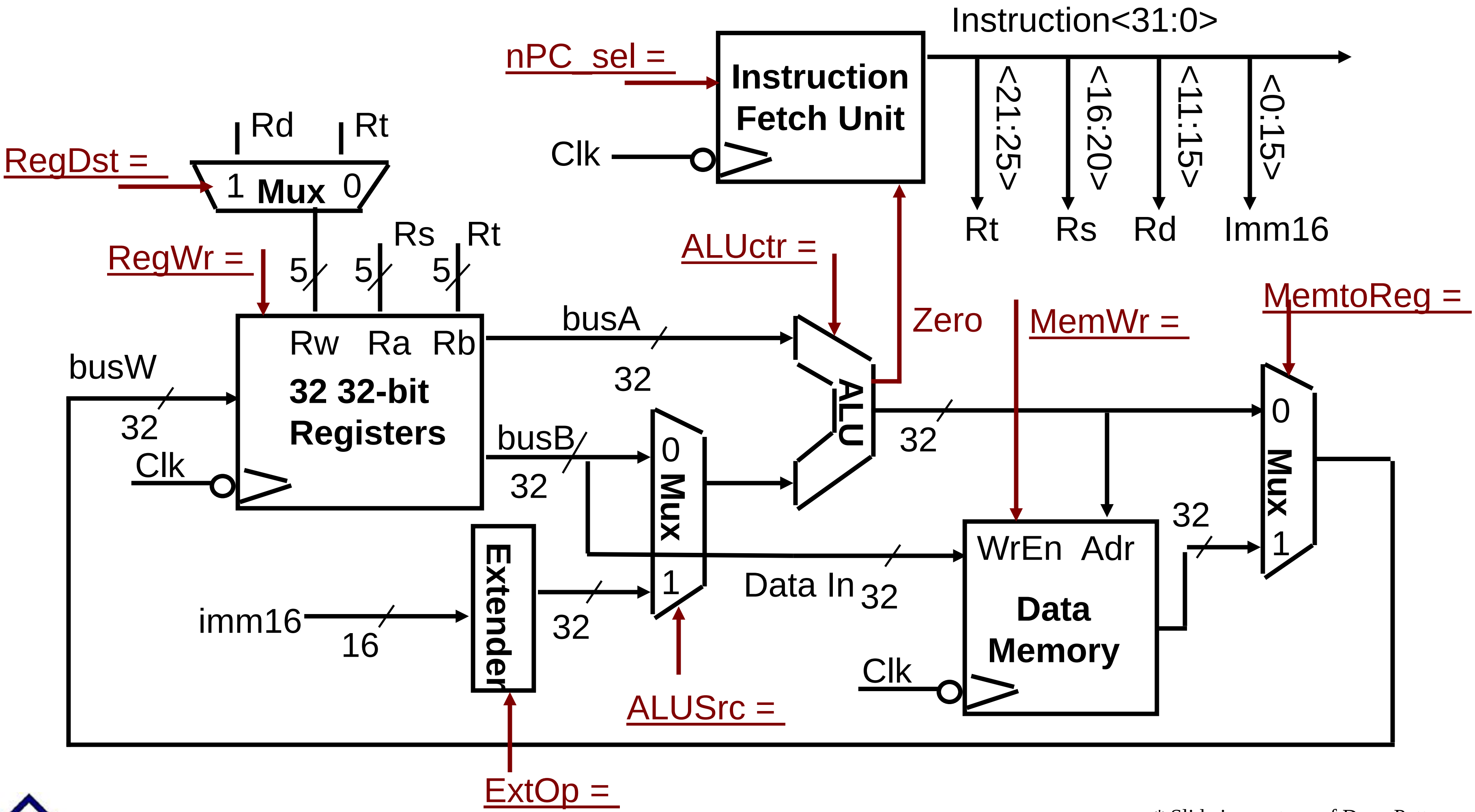


* Slide is courtesy of Dave Patterson

Single Cycle Datapath during Store

31 26	26 21	21 16	16 0
op	rs	rt	immediate

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

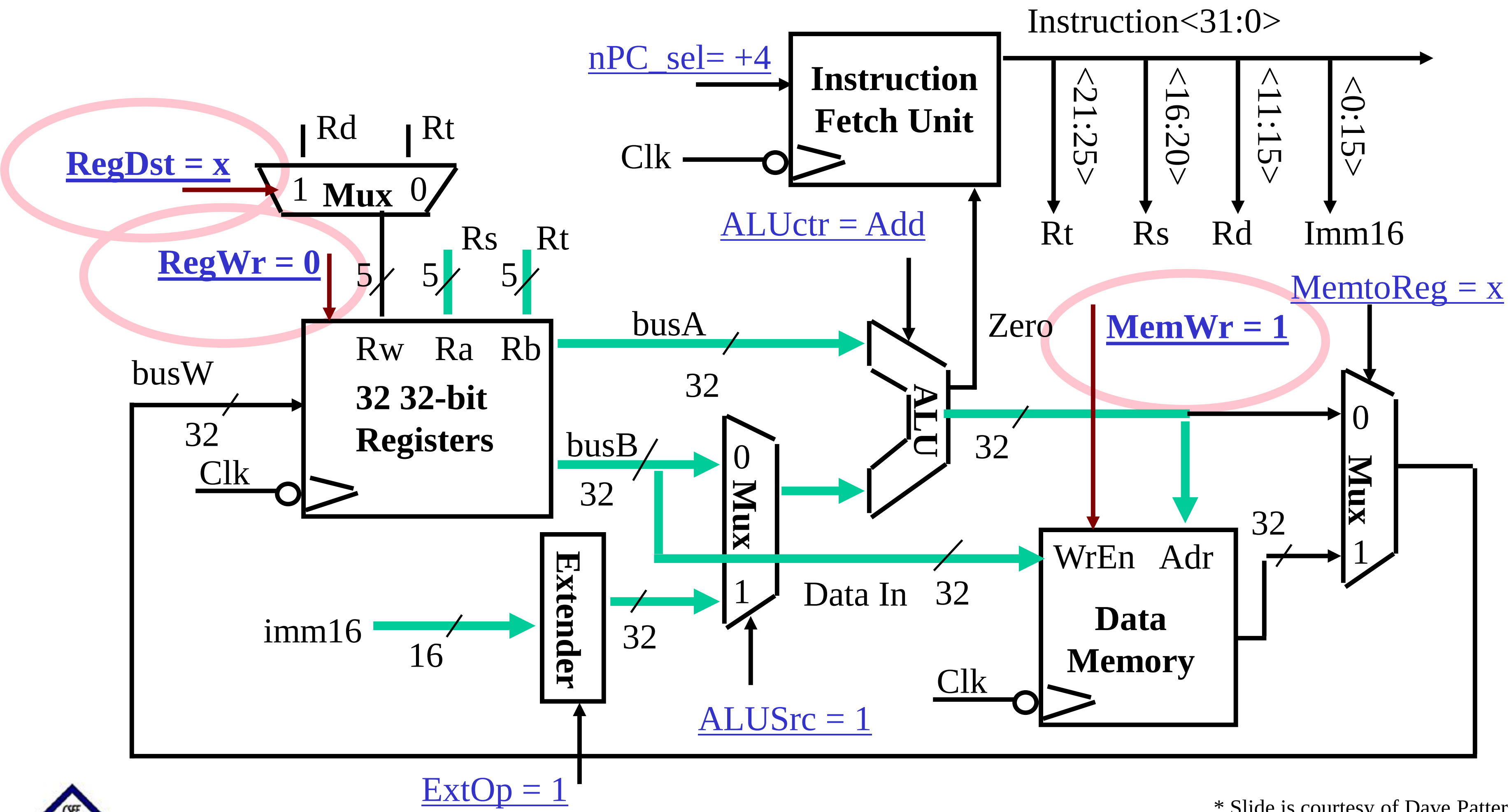


* Slide is courtesy of Dave Patterson

Single Cycle Datapath during Store

31	26	21	16	0
op	**rs**	**rt**	**immediate**	

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

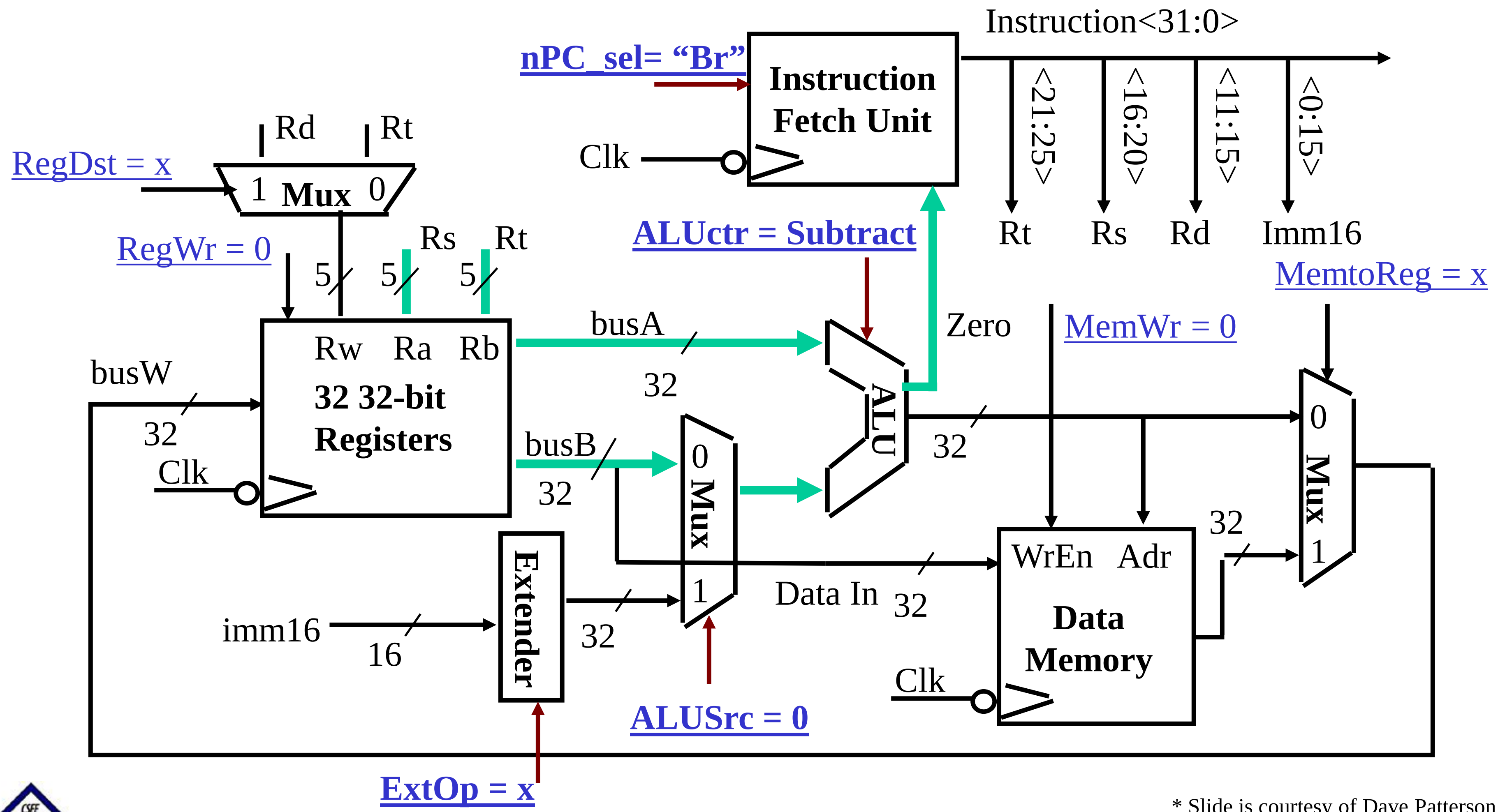


* Slide is courtesy of Dave Patterson

Single Cycle Datapath during Branch

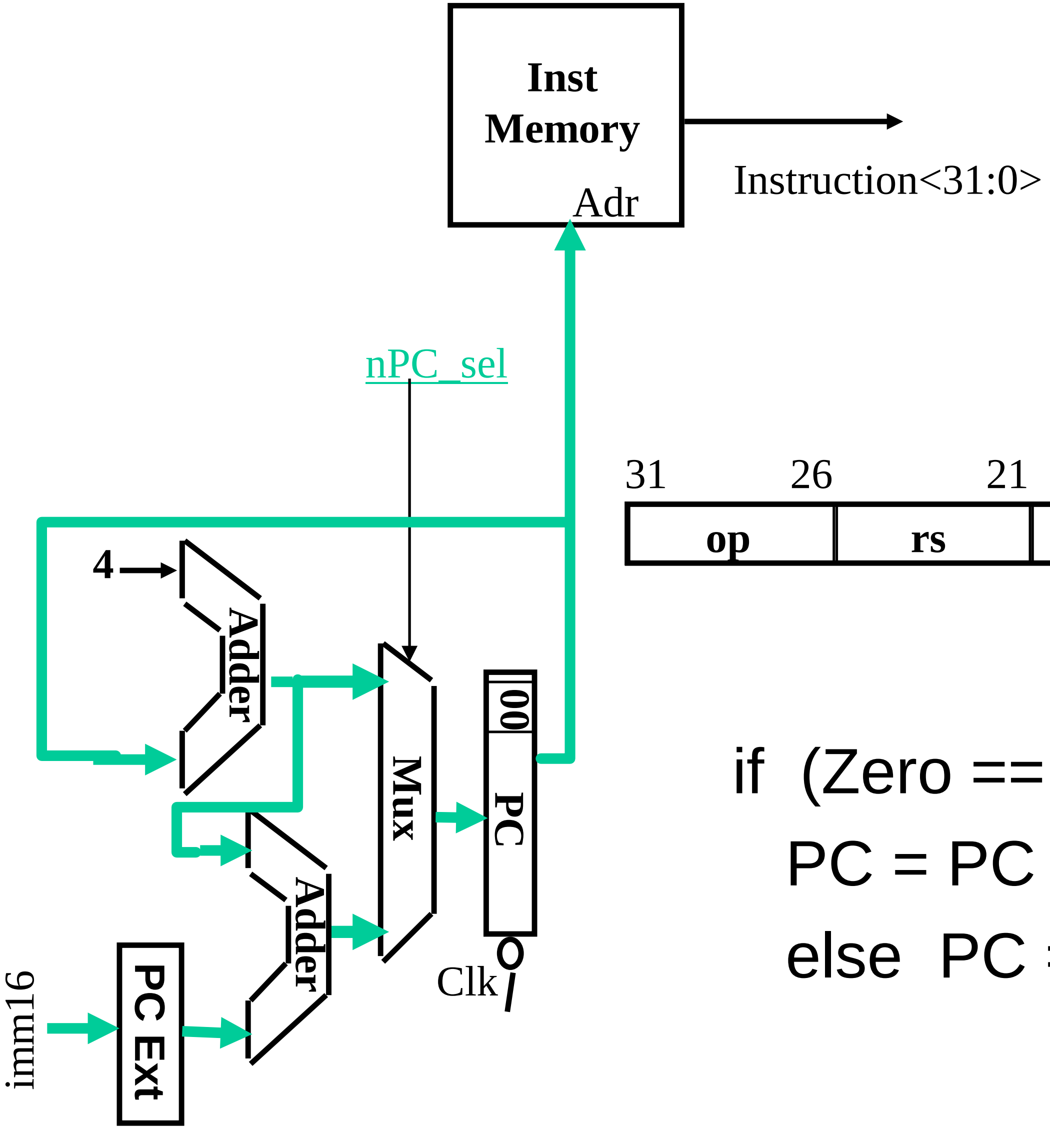
31	26	21	16	0
op	rs	rt	immediate	

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



* Slide is courtesy of Dave Patterson

Instruction Fetch Unit at End of Branch



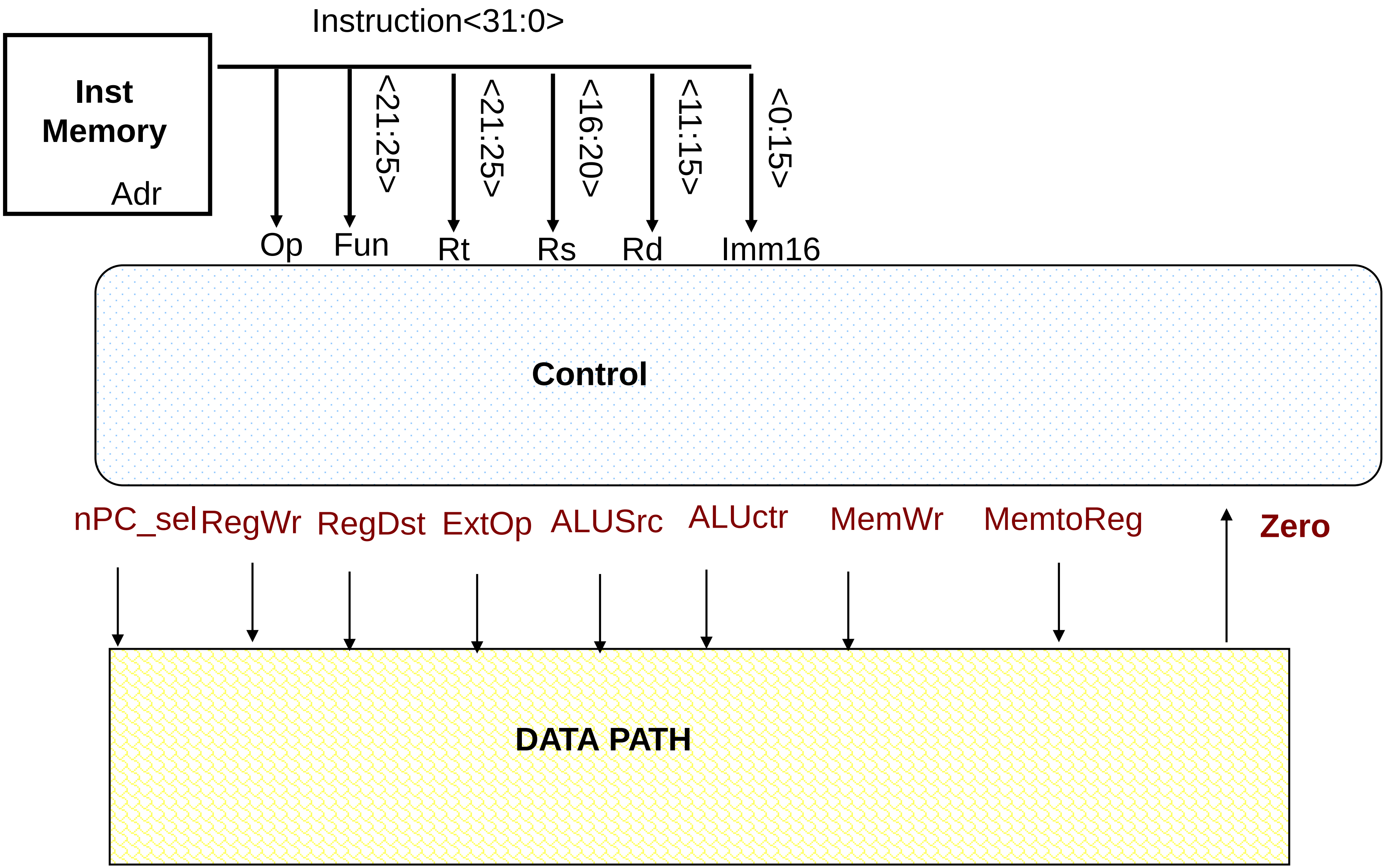
31 26 21 16 0

op	rs	rt	immediate

```
if  (Zero == 1)   then
    PC = PC + 4 + SignExt[imm16]*4 ;
    else  PC = PC + 4
```

* Slide is courtesy of Dave Patterson

Step 4: Given Datapath: RTL ⇒ Control



* Slide is courtesy of Dave Patterson

Value of Control Signals

inst **Register Transfer**

ADD R[rd] ⇐ R[rs] + R[rt]; PC ⇐ PC + 4

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

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

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

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

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

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

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

STORE MEM[R[rs] + sign_ext(Imm16)] ⇐ R[rs]; PC ⇐ PC + 4

ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemWr, 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”

* Slide is courtesy of Dave Patterson

A Summary of the Control Signals

See Appendix A

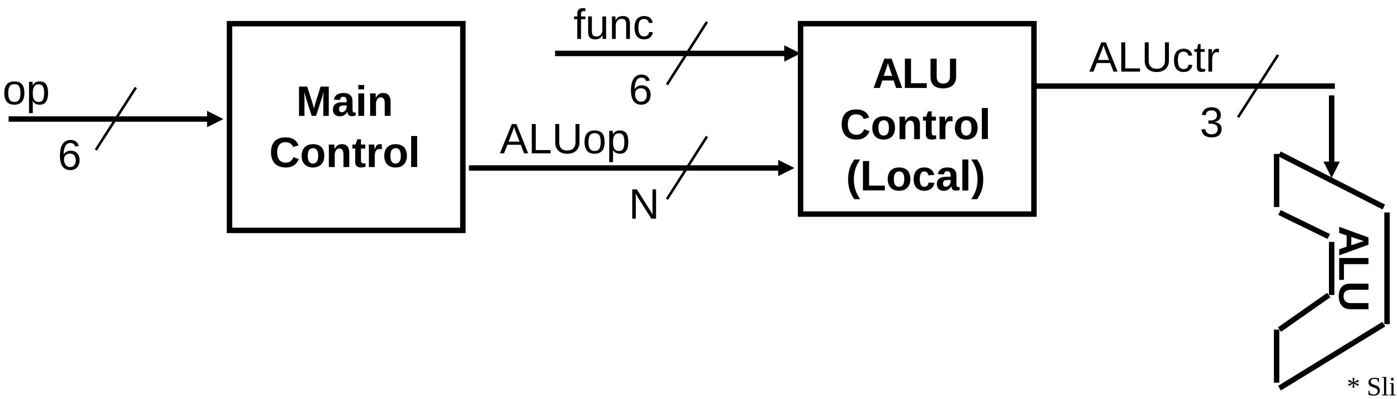
func	10 0000	10 0010	We Don't Care :-)				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	**sub**	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
nPCsel	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr<2:0>	Add	Subtract	Or	Add	Add	Subtract	xxx

	31	26	21	16	11	6	0	
R-type	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**		**add, sub**
I-type	**op**	**rs**	**rt**	**immediate**				**ori, lw, sw, beq**
J-type	**op**	**target address**						**jump**

* Slide is courtesy of Dave Patterson

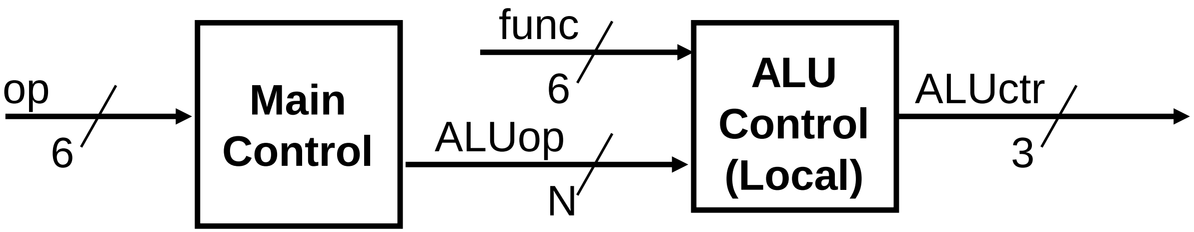
The Concept of Local Decoding

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



* Slide is courtesy of Dave Patterson

Encoding of ALUop



- ❑ In this exercise, ALUop has to be 2 bits wide to represent:
  - ➔ (1) “R-type” instructions
  - ➔“I-type” instructions that require the ALU to perform:

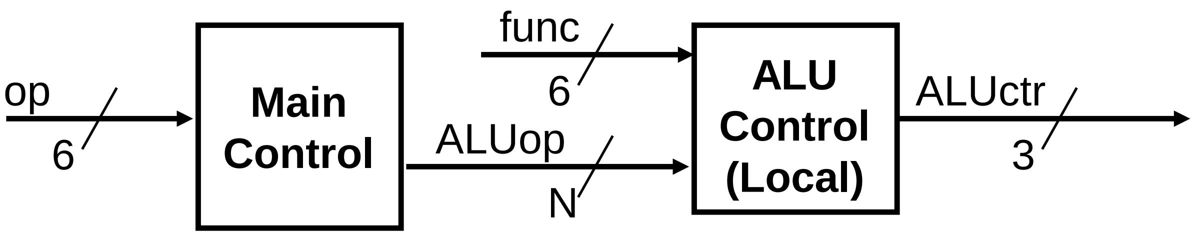
    (2) Or, (3) Add (address calculation), and (4) Subtract (BEQ instruction)
- ❑ To implement the full MIPS ISA, ALUop has to be 3 bits for:
  - ➔ (1) “R-type” instructions
  - ➔ “I-type” instructions that require the ALU to perform:

    (2) Or, (3) Add, (4) Subtract, and (5) And (Example: 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

* Slide is courtesy of Dave Patterson

Decoding of the “func” Field

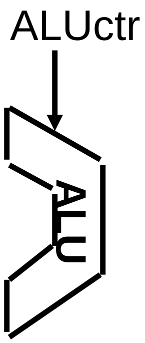


	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx

31 26 21 16 11 6 0

R-type	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**

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

* Slide is courtesy of Dave Patterson

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

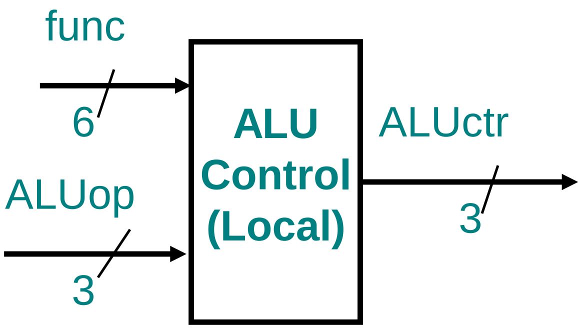
ALUop			func				ALU Operation	ALUctr		
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>		bit<2>	bit<1>	bit<0>
0	0	0	x	x	x	x	Add	0	1	0
0	x	1	x	x	x	x	Subtract	1	1	0
0	1	x	x	x	x	x	Or	0	0	1
1	x	x	0	0	0	0	Add	0	1	0
1	x	x	0	0	1	0	Subtract	1	1	0
1	x	x	0	1	0	0	And	0	0	0
1	x	x	0	1	0	1	Or	0	0	1
1	x	x	1	0	1	0	Set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

* Slide is courtesy of Dave Patterson

The Logic Equation for ALUctr

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



ALUctr<2> = !ALUop<2> & ALUop<0> +
ALUop<2> & !func<2> & func<1> & !func<0>

Similarly:

ALUctr<1> = !ALUop<2> & !ALUop<1> +
ALUop<2> & !func<2> & !func<0>

ALUctr<0> = !ALUop<2> & ALUop<0>
+ ALUop<2> & !func<3> & func<2> & !func<1> & func<0>
+ ALUop<2> & func<3> & !func<2> & func<1> & !func<0>

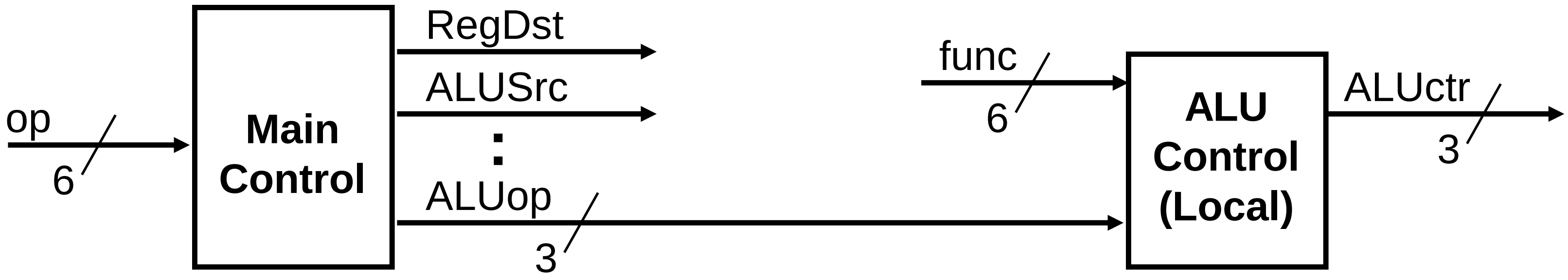
* Slide is courtesy of Dave Patterson

Step 5: Logic for each control signal

- ❑ nPC_sel ⇐ if (OP == BEQ) then ZERO else 0
- ❑ ALUsrc ⇐ if (OP == “Rtype”) then “regB” else “immed”
- ❑ ALUctr ⇐ if (OP == “Rtype”) 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

* Slide is courtesy of Dave Patterson

"Truth Table" for the 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
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 (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

* Slide is courtesy of Dave Patterson

The “Truth Table” for RegWrite

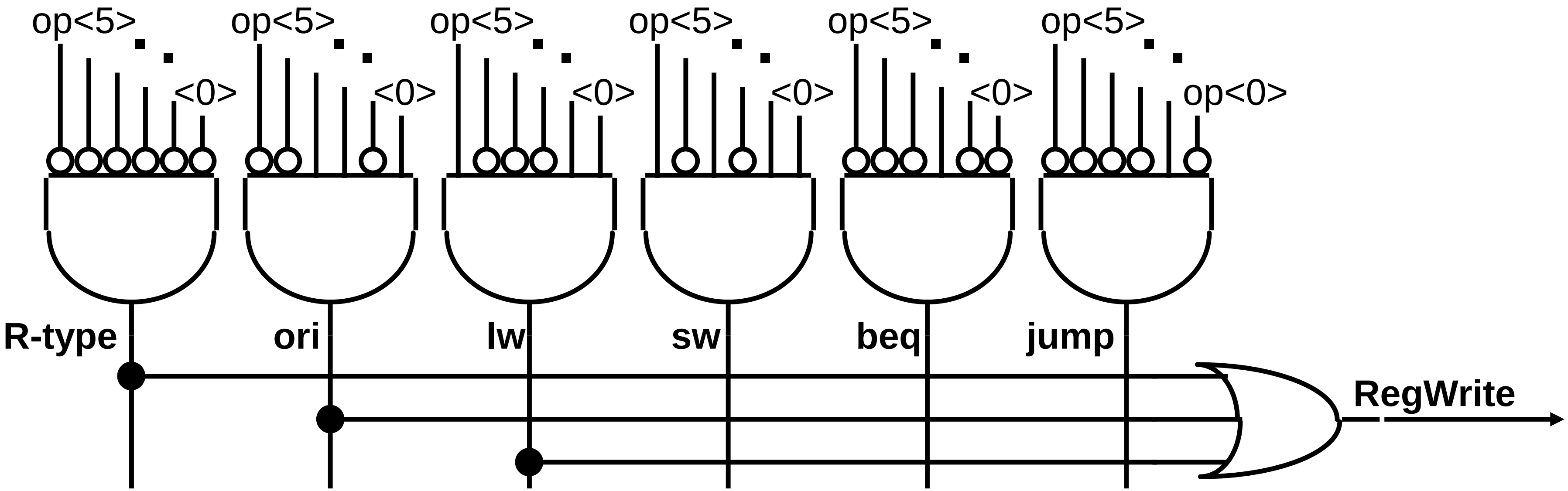
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegWrite	1	1	1	0	0	0

RegWrite = R-type + ori + lw

= !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)

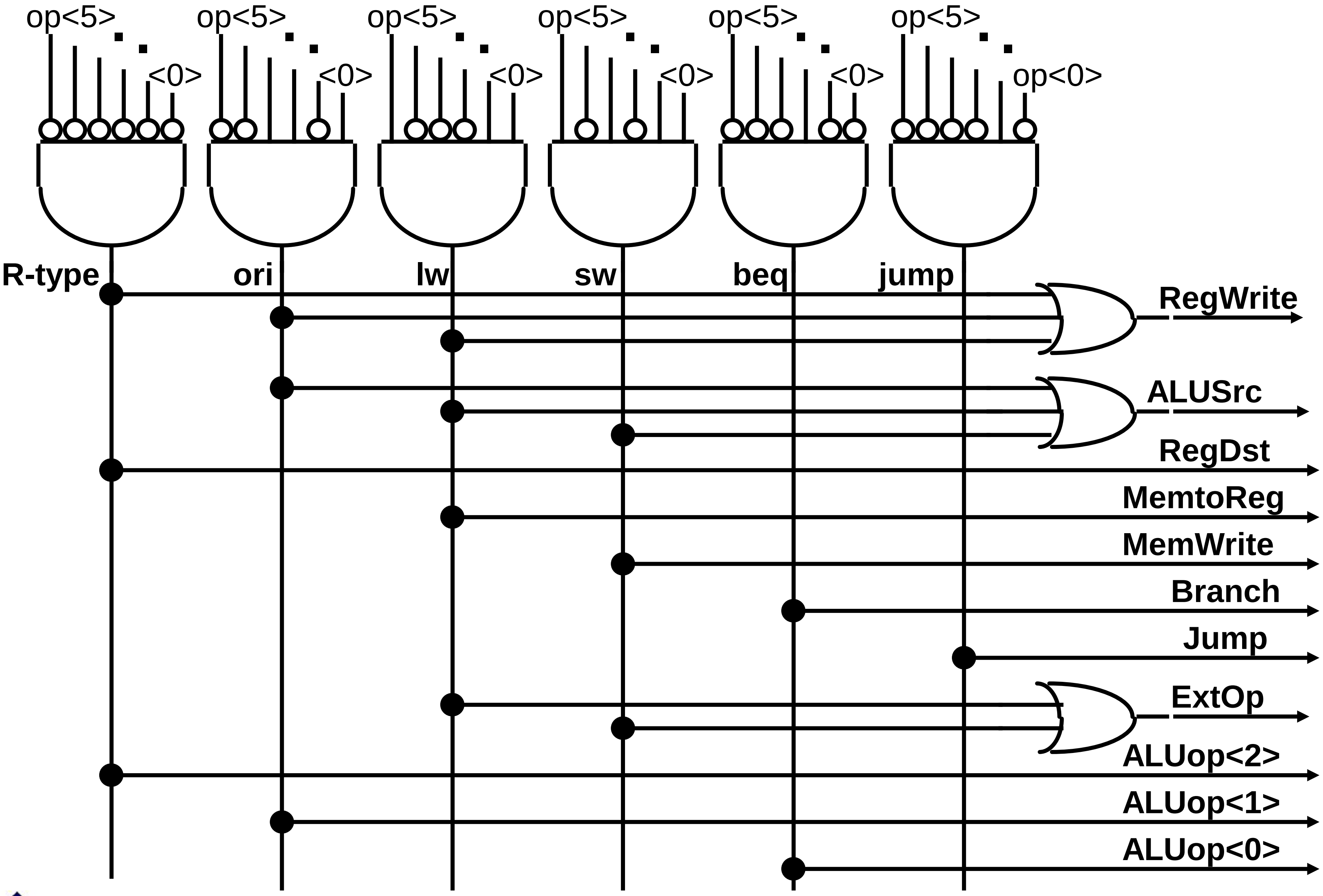
+ !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)

+ op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



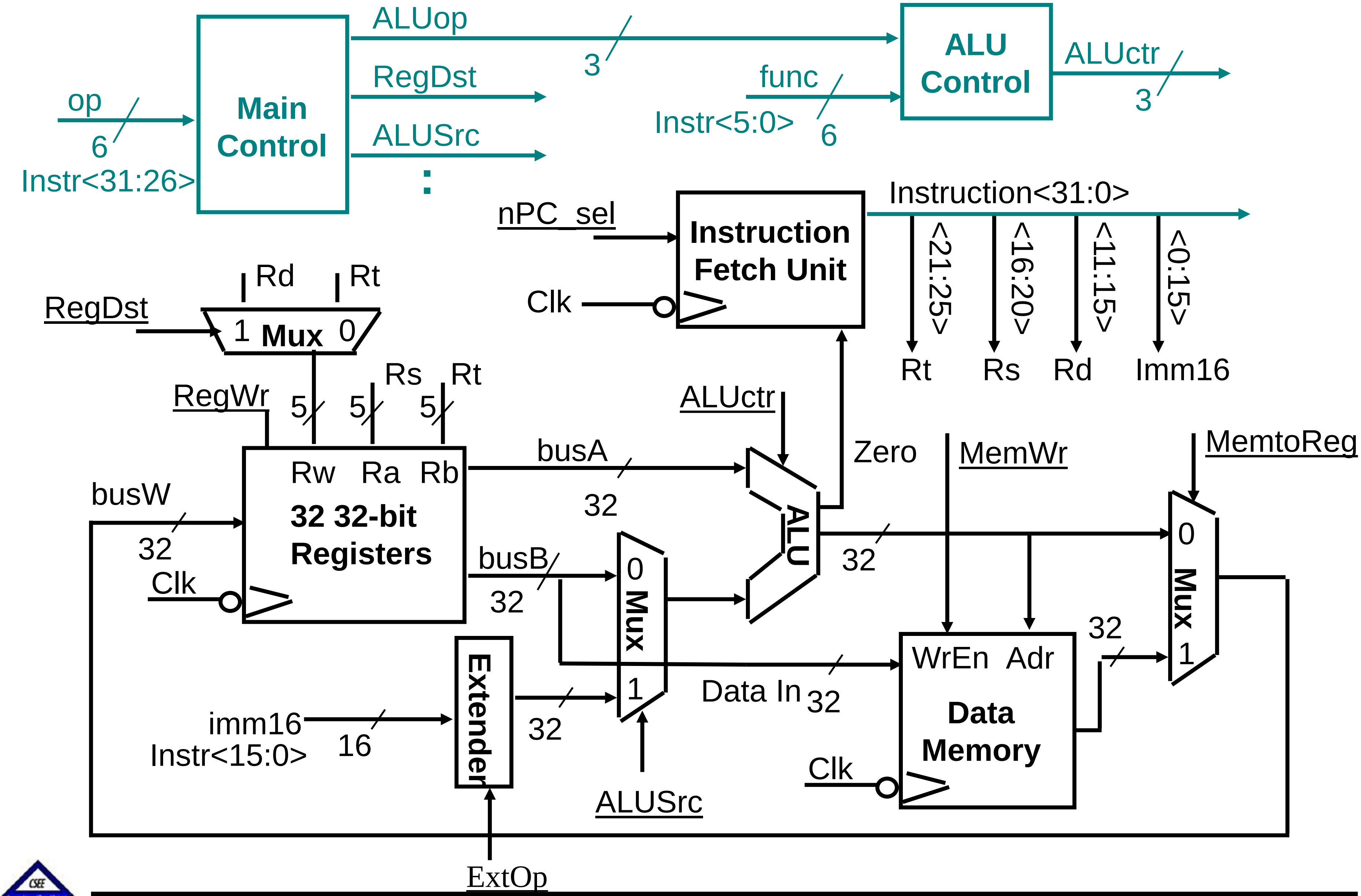
* Slide is courtesy of Dave Patterson

Implementation of the Main Control

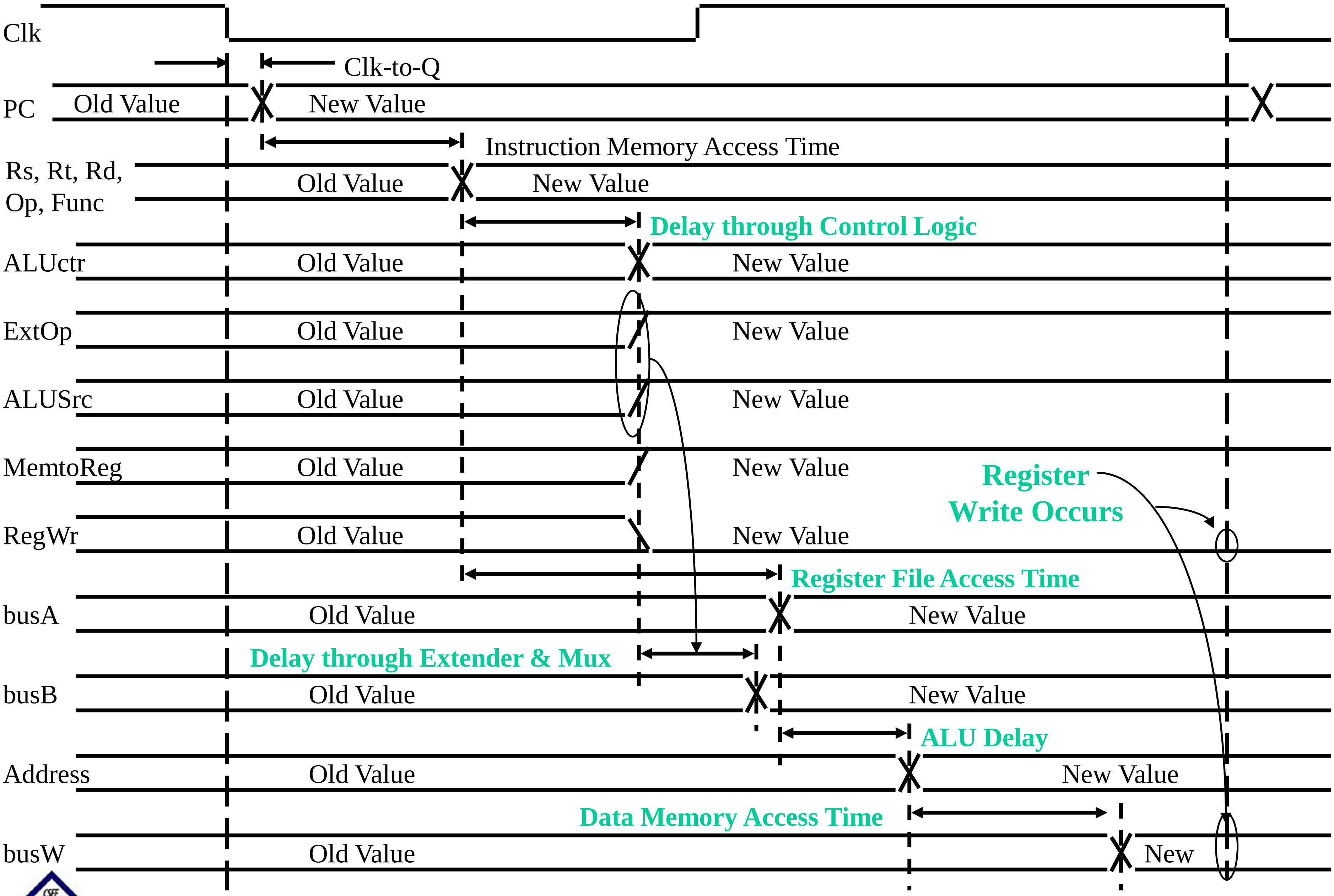


* Slide is courtesy of Dave Patterson

A Single Cycle Processor



Worst Case Timing (Load)



Conclusion

❑ ***Summary***

- ➔ Processor design steps

  (ISA analysis, component selection, datapath assembly, control unit)

- ➔Building a datapath

  (Instruction fetch, register transfer requirements)

- ➔ Control unit design

  (Steps of control design, register transfer logic)

- ➔ Single cycle processor

  (Advantage and disadvantage, integration of datapath and control)

- ➔ Circuit implementation of control unit

  (Logic equations, truth tables, combinational circuit)

❑ ***Next Lecture***

- ➔ Multi-cycle datapath
- ➔ Multi-cycle control

Read section 4.4 in 5th Ed., or section 4.4 in 4th Ed. of the textbook