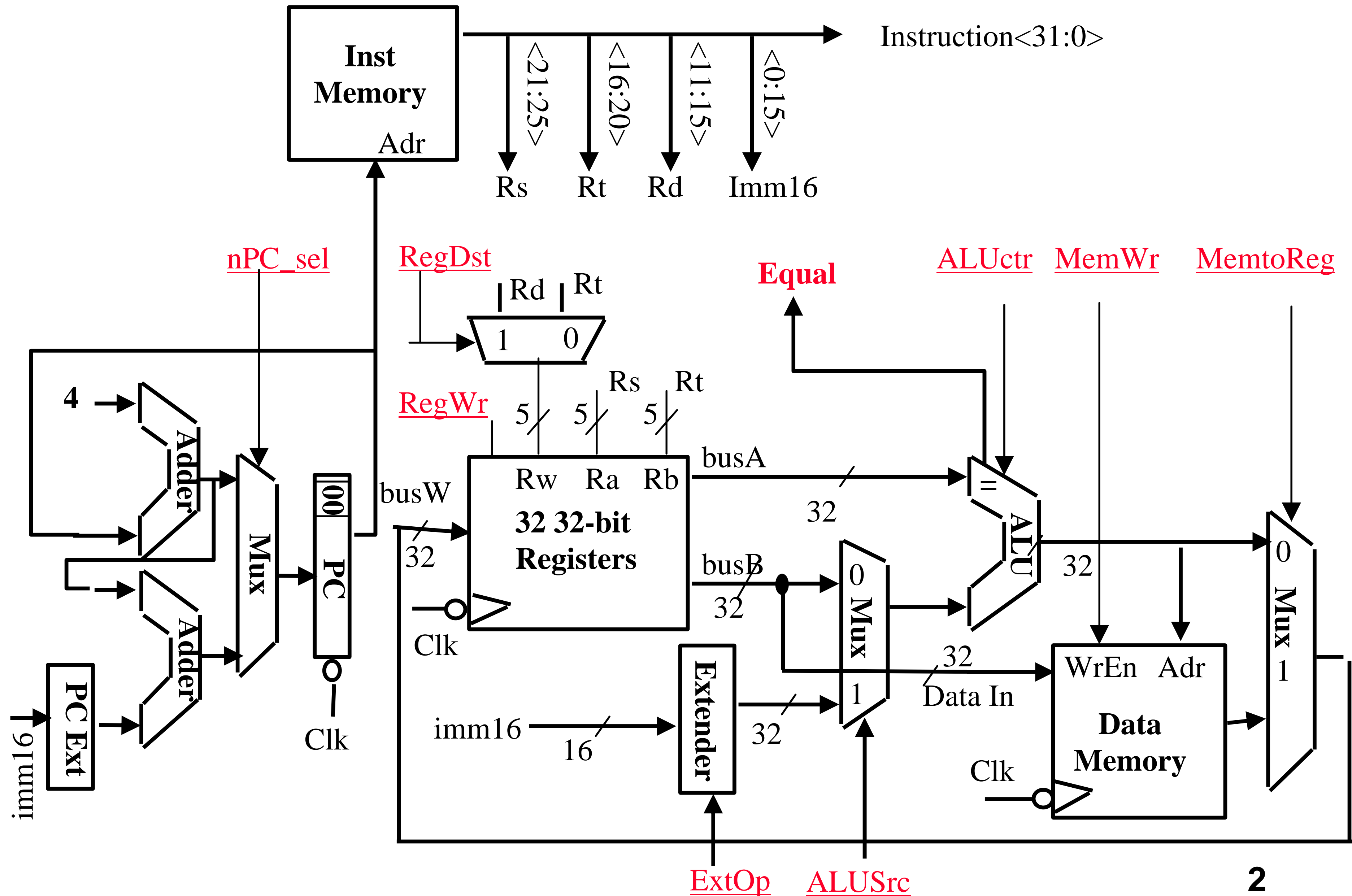
# Processor Design - 2

Computer Architecture

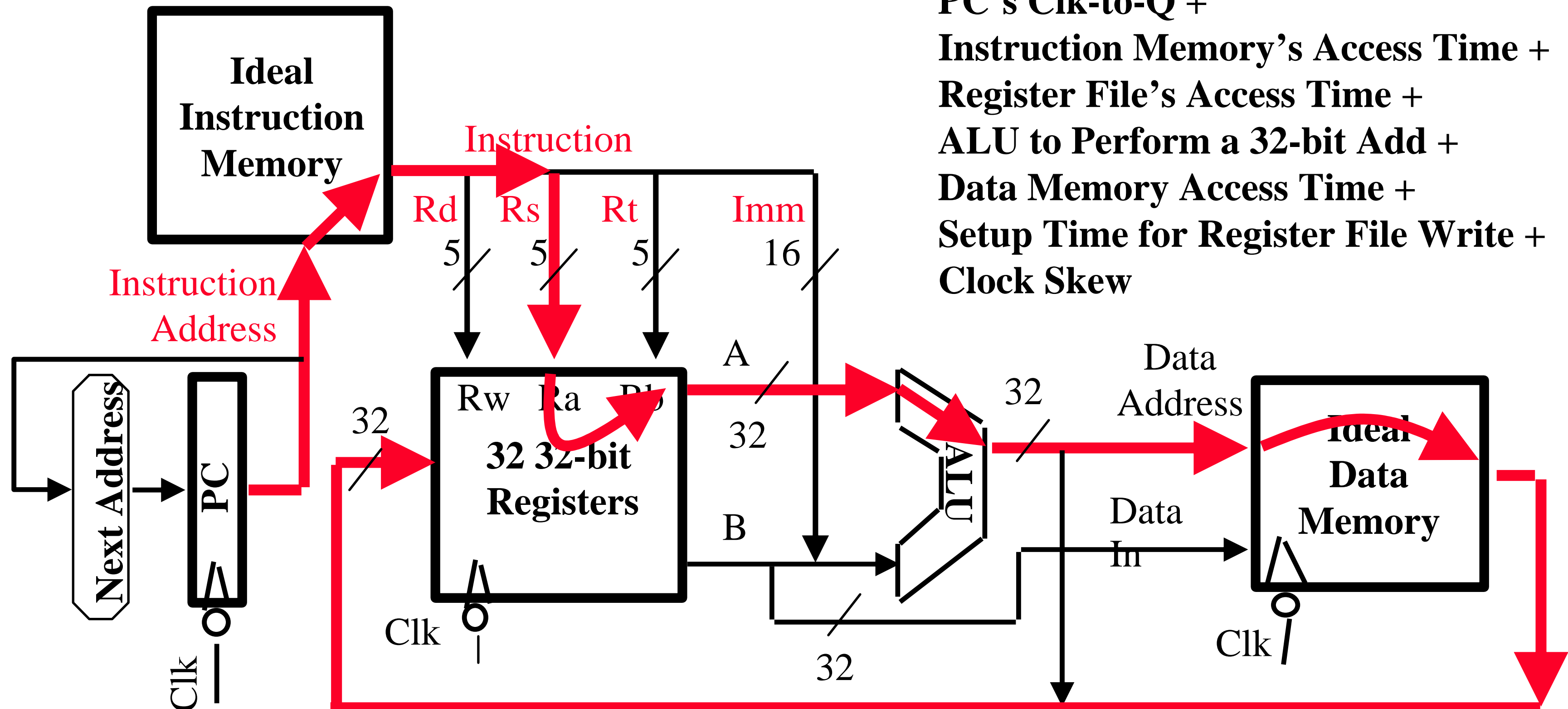
# Summary: A Single Cycle Datapath



# An Abstract View of the Critical Path

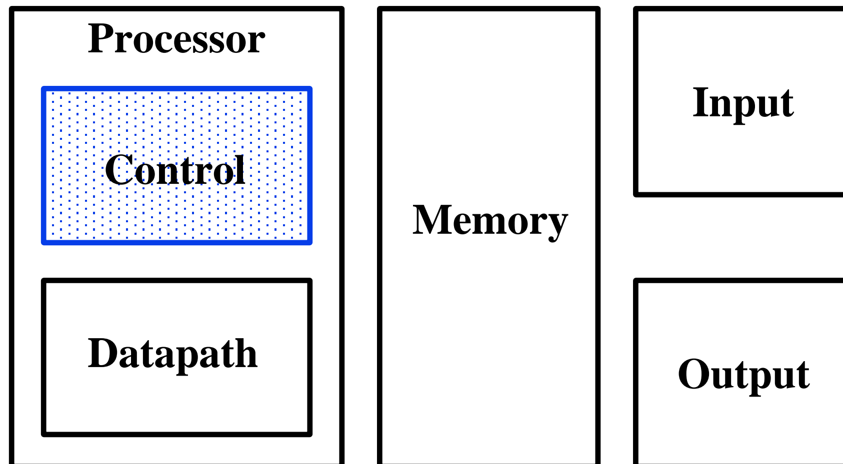
- **Register file and ideal memory:**
  - The CLK input is a factor ONLY during write operation
  - During read operation, behave as combinational logic:
    - Address valid => Output valid after "access time."

**Critical Path (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**



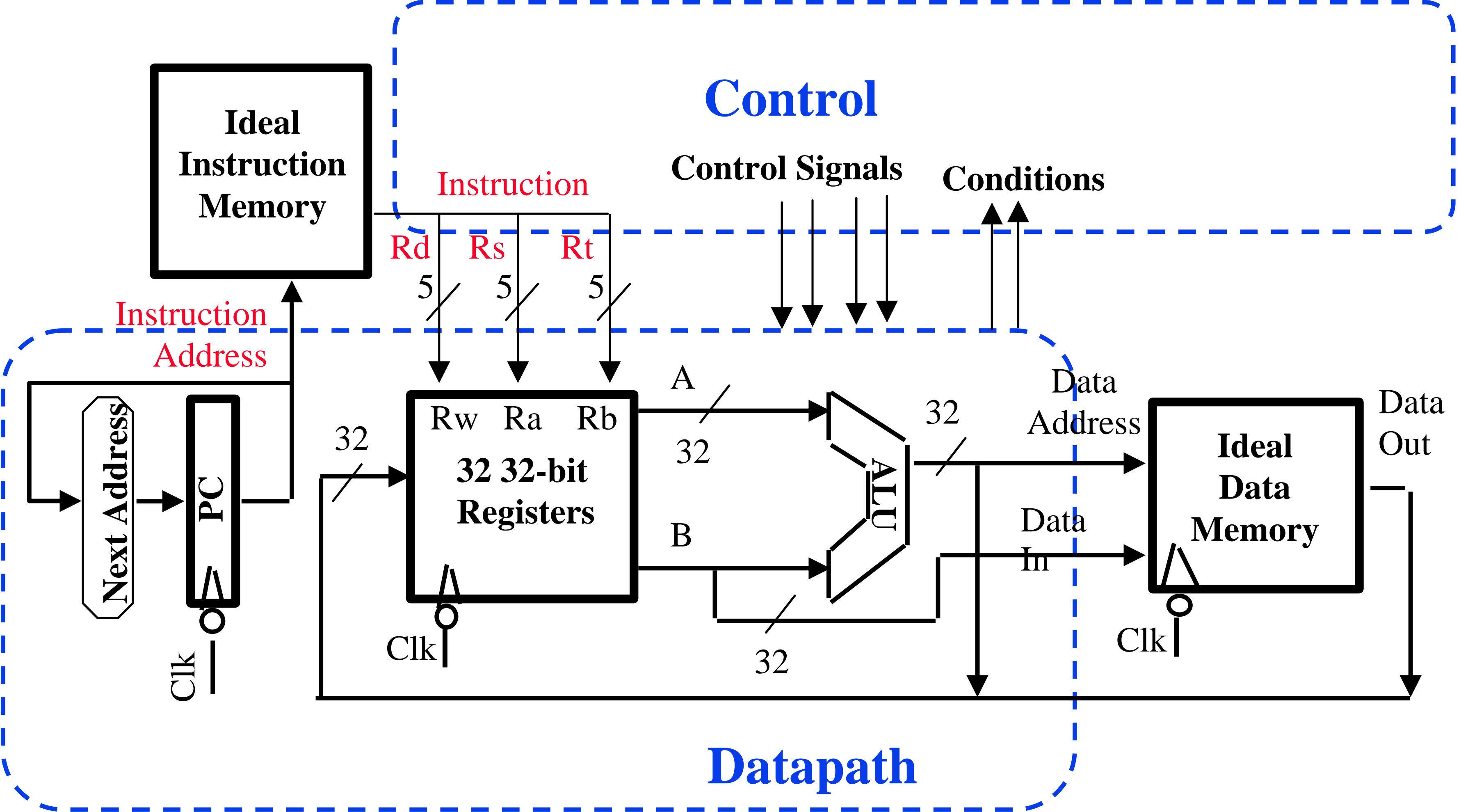
# The Big Picture: Where are We Now?

- The Five Classic Components of a Computer



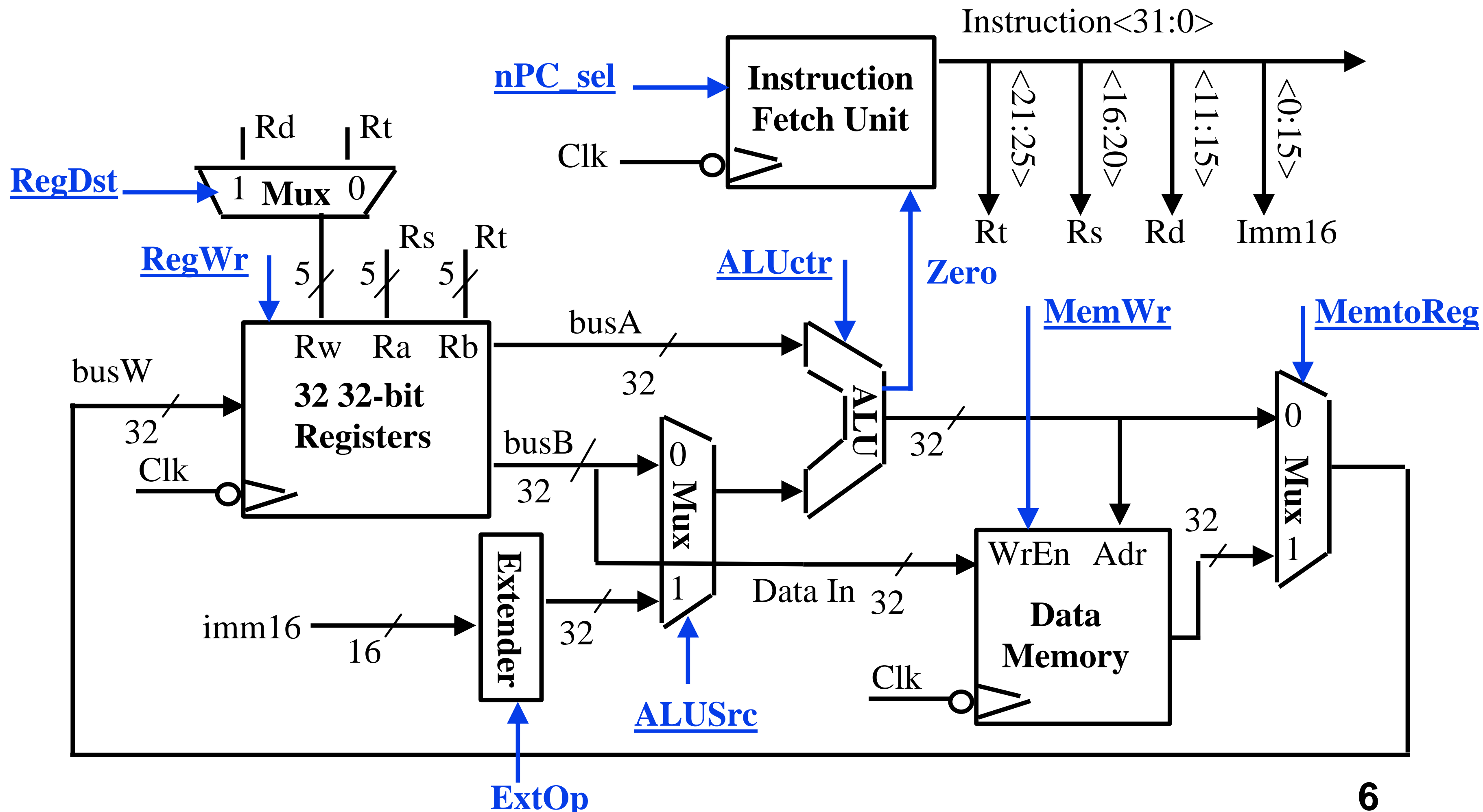

- Next Topic: Designing the Control for the Single Cycle Datapath

# An Abstract View of the Implementation



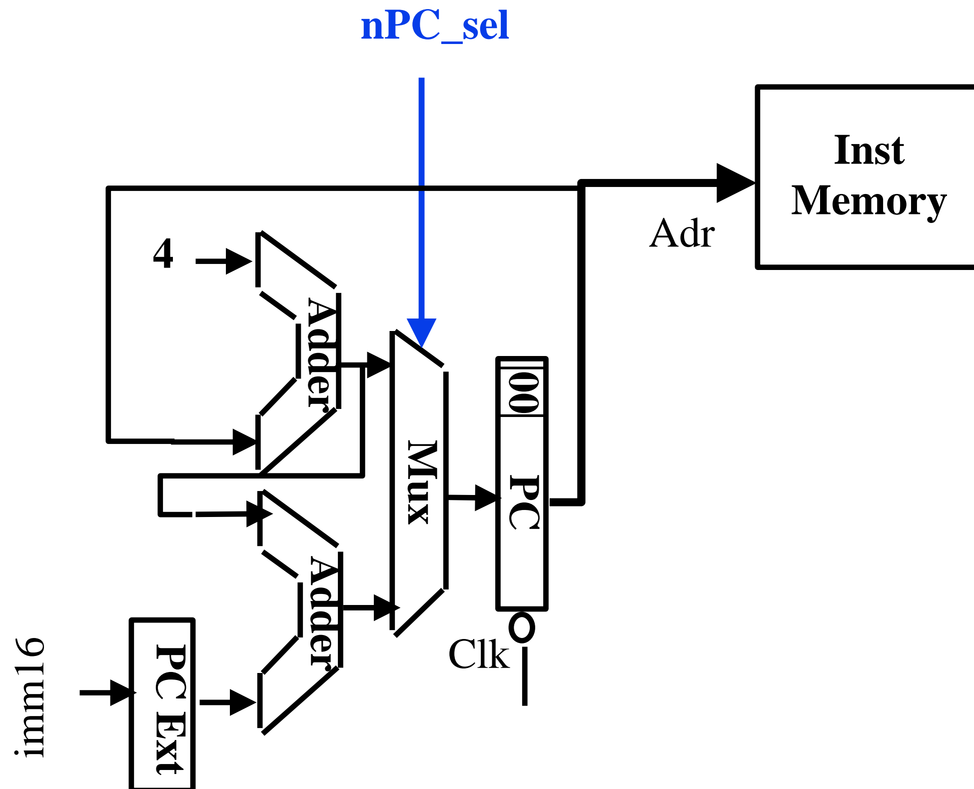
# Recap: A Single Cycle Datapath

- **Rs, Rt, Rd and Imed16 hardwired into datapath from Fetch Unit**
- **We have everything except control signals (underline)**
  – Today's lecture will show you how to generate the control signals



# Recap: Meaning of the Control Signals

- **nPC_sel:** $0 \Rightarrow PC \leftarrow PC + 4$
  $1 \Rightarrow PC \leftarrow PC + 4 + SignExt(Im16) \,||\, 00$
- **Later in lecture: higher-level connection between mux and branch cond**

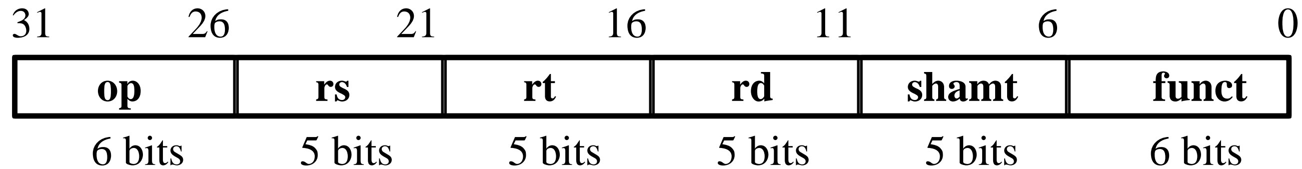


# Recap: Meaning of the Control Signals

- **ExtOp:** "zero", "sign"
- **ALUsrc:** 0 ⇒ regB; 1 ⇒ immed
- **ALUctr:** "add", "sub", "or"

- **MemWr:** 1 ⇒ write memory
- **MemtoReg:** 0 ⇒ ALU; 1 ⇒ Mem
- **RegDst:** 0 ⇒ "rt"; 1 ⇒ "rd"
- **RegWr:** 1 ⇒ write register

# The `add` Instruction

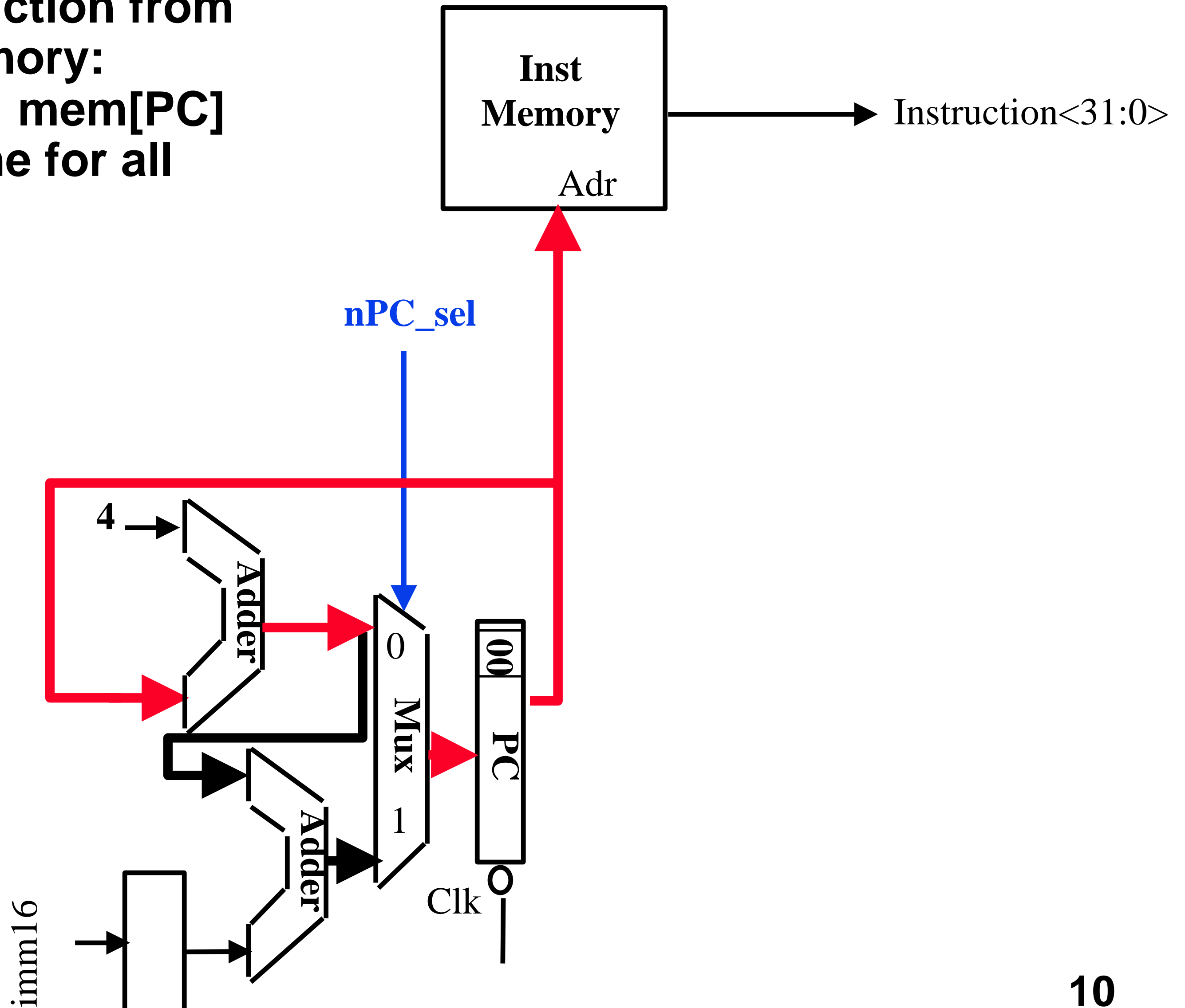



- `add rd, rs, rt`

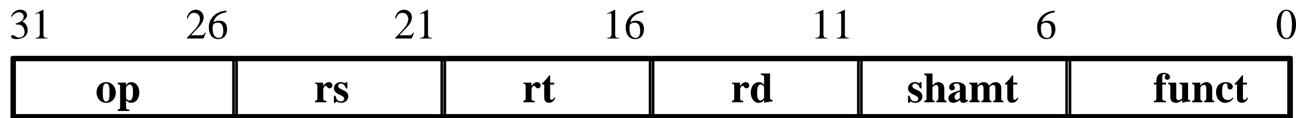
| | |
|---|---|
| **mem[PC]** | **Fetch the instruction from memory** |
| **R[rd] ← R[rs] + R[rt]** | **The actual operation** |
| **PC ← PC + 4** | **Calculate the next instruction's address** |

# Fetch Unit at the Beginning of `add`

- **Fetch the instruction from Instruction memory: Instruction ← mem[PC] (This is the same for all instructions)**



# The Single Cycle Datapath during `add`

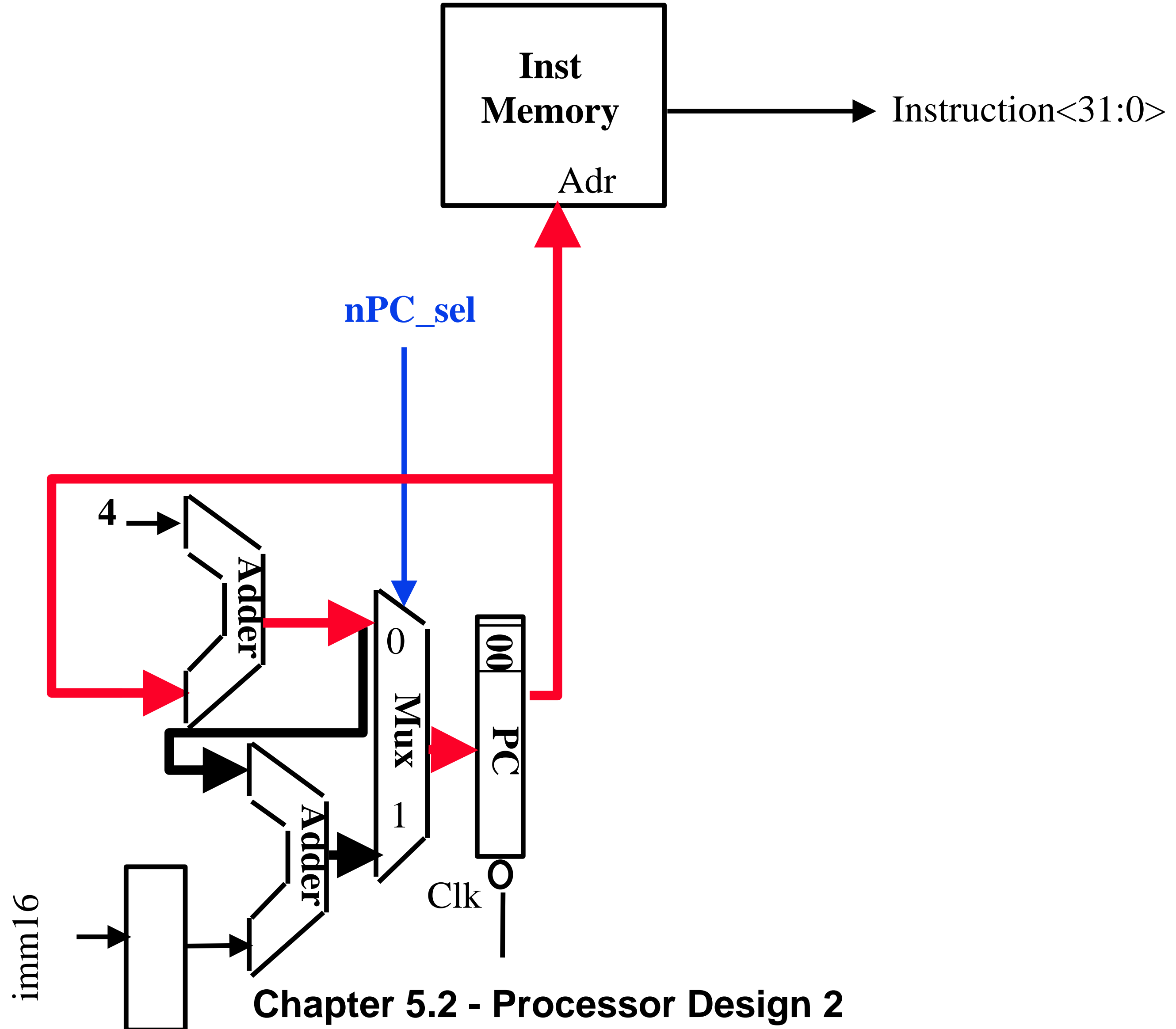



- `R[rd] ← R[rs] + R[rt]`

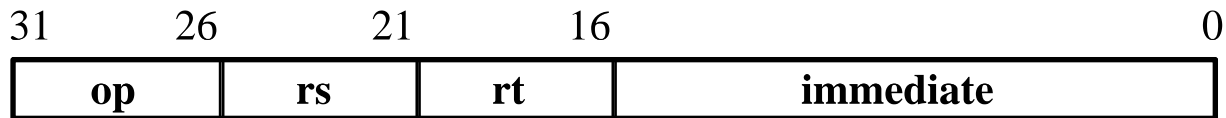
Instruction<31:0>
nPC_sel= +4
Instruction Fetch Unit
Clk
<21:25>
<16:20>
<11:15>
<0:15>
Rt
Rs
Rd
Imm16
RegDst = 1
Rd
Rt
1 Mux 0
ALUctr = Add
RegWr = 1
5
5
5
Rs
Rt
Rw Ra Rb
32 32-bit Registers
busW
32
Clk
busA
32
busB
32
Zero
MemWr = 0
MemtoReg = 0
ALU
0
Mux
1
32
imm16
16
Extender
32
Data In
32
WrEn Adr
Data Memory
Clk
ALUSrc = 0
ExtOp = x

# Instruction Fetch Unit at the End of `add`

- **PC $\leftarrow$ PC + 4**
  - This is the same for all instructions except: Branch and Jump



# The Single Cycle Datapath during Or Immediate




- `R[rt] ← R[rs] or ZeroExt(Imm16)`

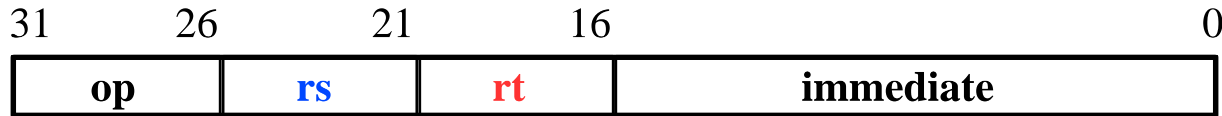
Instruction<31:0>
nPC_sel =
Instruction Fetch Unit
Clk
<21:25>
<16:20>
<11:15>
<0:15>
Rt
Rs
Rd
Imm16
RegDst =
Rd
Rt
1 Mux 0
RegWr =
5
5
5
Rs
Rt
ALUctr =
Zero
MemWr =
MemtoReg =
busA
32
Rw Ra Rb
32 32-bit Registers
busW
32
Clk
busB
32
0
Mux
1
ALU
32
0
Mux
1
32
WrEn Adr
Data Memory
Data In
32
Clk
imm16
16
Extender
32
ALUSrc =
ExtOp =

# The Single Cycle Datapath during Or Immediate

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

- R[rt] ← R[rs] or ZeroExt(Imm16)

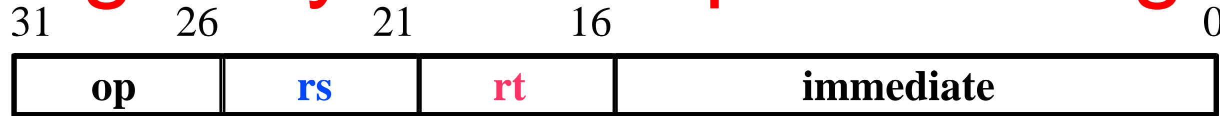
# The Single Cycle Datapath during Load




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

Instruction<31:0>
nPC_sel= +4
Instruction Fetch Unit
Clk
<21:25>
<16:20>
<11:15>
<0:15>
Rt
Rs
Rd
Imm16
MemtoReg = 1
RegDst = 0
Rd
Rt
1 Mux 0
ALUctr = Add
RegWr = 1
Rs
Rt
5
5
5
Zero
MemWr = 0
busA
Rw Ra Rb
32
busW
32 32-bit Registers
32
ALU
32
0
Mux
1
Clk
busB
32
0
Mux
1
WrEn Adr
Data In
32
Data Memory
32
imm16
16
Extender
32
Clk
ALUSrc = 1
ExtOp = 1

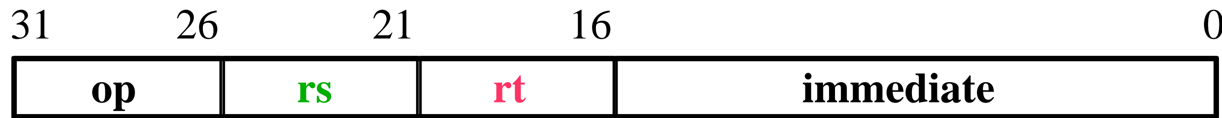
# The Single Cycle Datapath during Store




- `Data Memory {R[rs] + SignExt[imm16]} ← R[rt]`

Instruction<31:0>
nPC_sel =
Instruction Fetch Unit
Clk
<21:25>
<16:20>
<11:15>
<0:15>
Rt
Rs
Rd
Imm16
RegDst =
Rd
Rt
1 Mux 0
RegWr =
5
5
Rs
5
Rt
ALUctr =
Zero
MemWr =
MemtoReg =
busA
32
Rw Ra Rb
32 32-bit Registers
busW
32
Clk
busB
32
0 Mux 1
ALU
32
0 Mux 1
Extender
imm16
16
32
Data In
32
WrEn Adr
Data Memory
Clk
32
ALUSrc =
ExtOp =

# The Single Cycle Datapath during Store




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

nPC_sel= +4
Instruction Fetch Unit
Clk
Instruction<31:0>
<21:25>
<16:20>
<11:15>
<0:15>
Rt
Rs
Rd
Imm16
RegDst = x
Rd
Rt
1 Mux 0
RegWr = 0
5
5
5
Rs
Rt
ALUctr = Add
Zero
MemWr = 1
MemtoReg = x
busW
32
Clk
Rw Ra Rb
32 32-bit Registers
busA
32
busB
32
0 Mux 1
ALU
32
0 Mux 1
32
WrEn Adr
Data In
32
Data Memory
Clk
imm16
16
Extender
32
ALUSrc = 1
ExtOp = 1

# The 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`

nPC_sel= "Br"

Instruction Fetch Unit

Clk

Instruction<31:0>

<21:25> <16:20> <11:15> <0:15>

Rt Rs Rd Imm16

RegDst = x

Rd Rt

1 Mux 0

RegWr = 0

ALUctr =Sub

Rs Rt

5 5 5

Rw Ra Rb

32 32-bit Registers

busW 32 Clk

busA 32

busB 32

Zero

MemWr = 0

MemtoReg = x

ALU

32

0 Mux 1

Data In 32

WrEn Adr

Data Memory

Clk

imm16 16

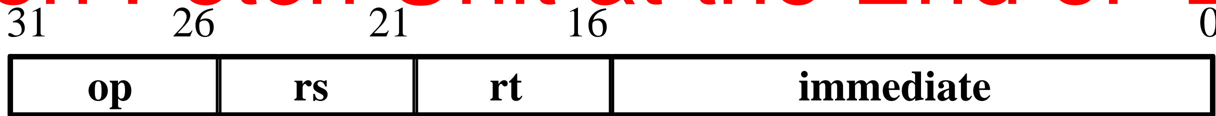
Extender

32

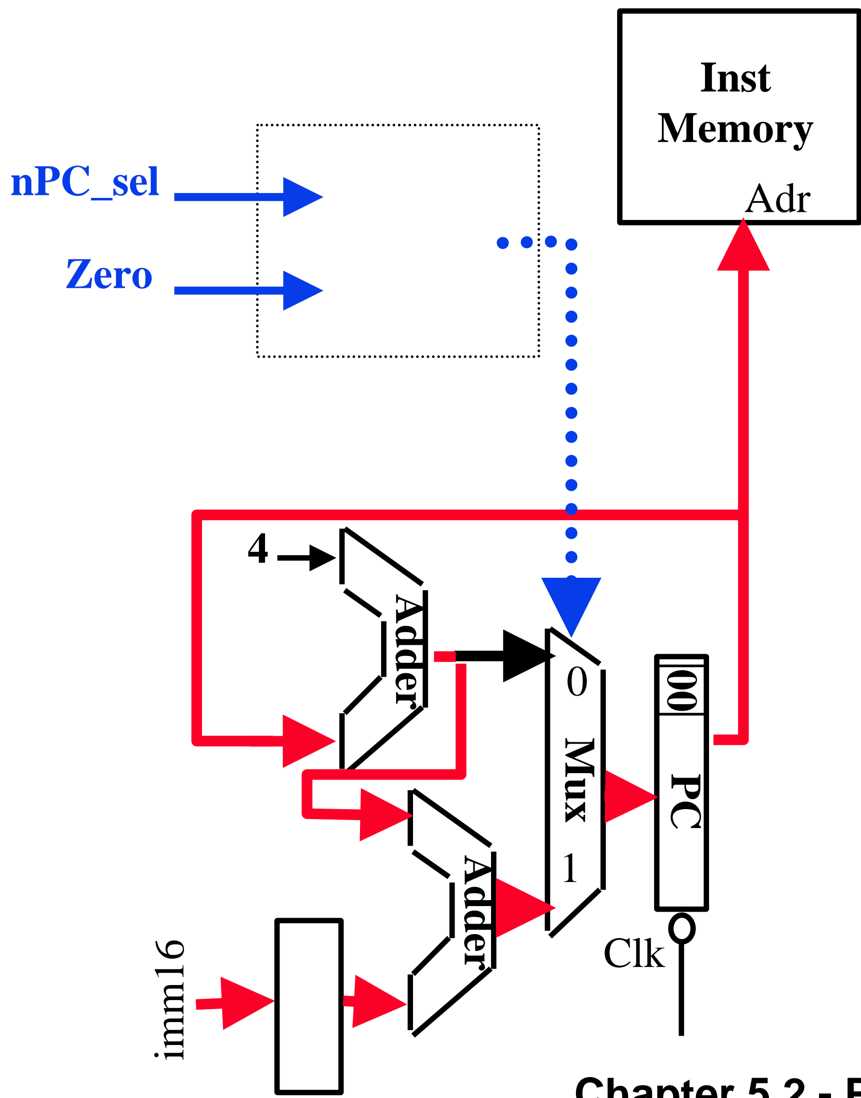
ALUSrc = 0

ExtOp = x

# Instruction Fetch Unit at the End of Branch



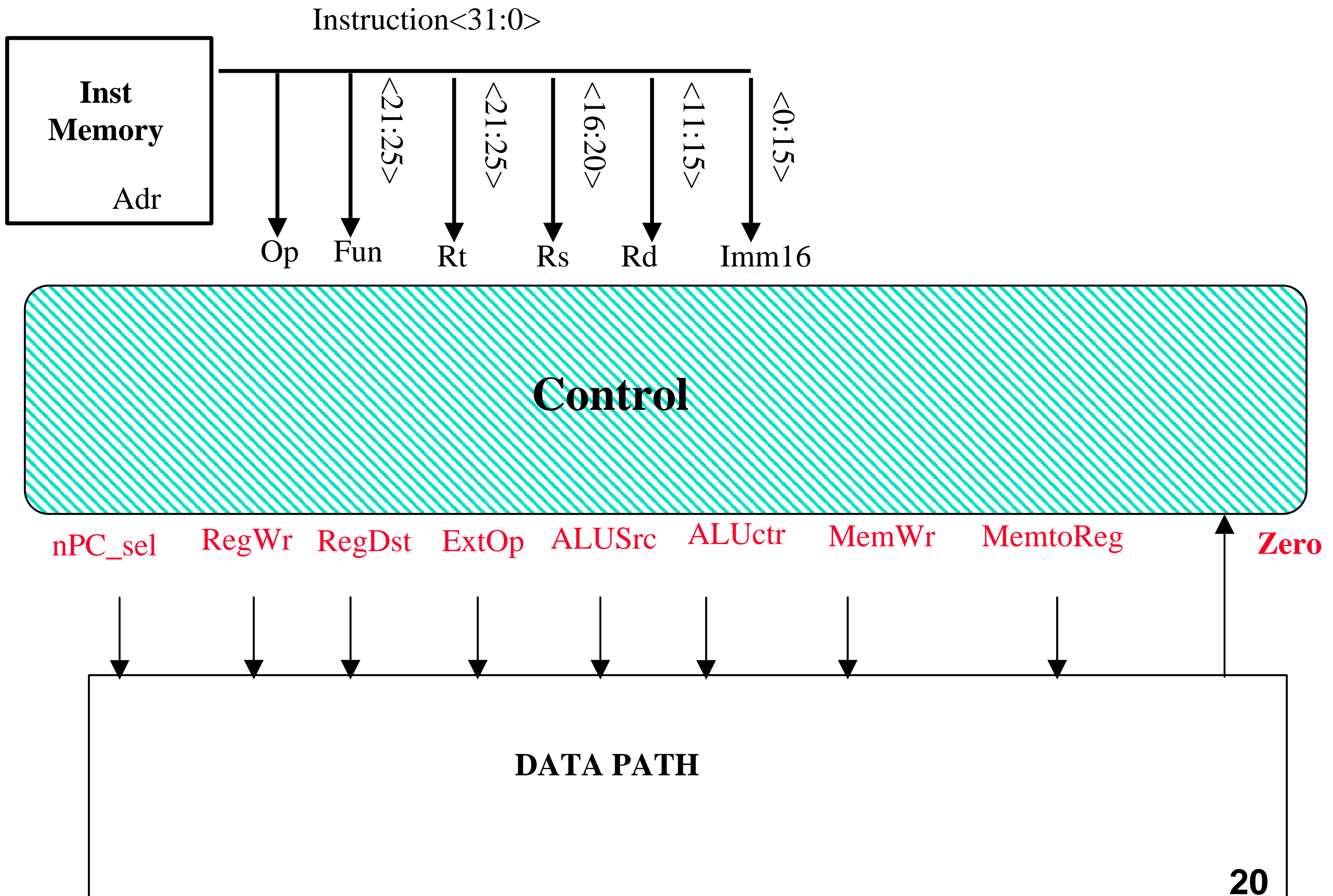

- if (Zero == 1) then PC = PC + 4 + SignExt(imm16)×4 ; else PC = PC + 4




- What is encoding of nPC_sel?
  - **Direct MUX select?**
  - **Branch / not branch**
- Let's choose second option

| nPC_sel | zero? | MUX |
|---|---|---|
| 0 | x | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

# Step 4: Given Datapath: RTL → Control



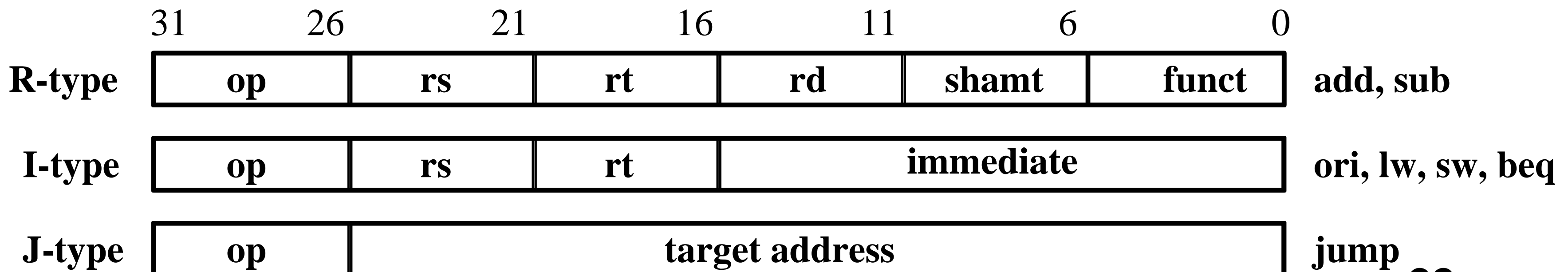
# A Summary of Control Signals

| 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" | |

# A Summary of Control Signals

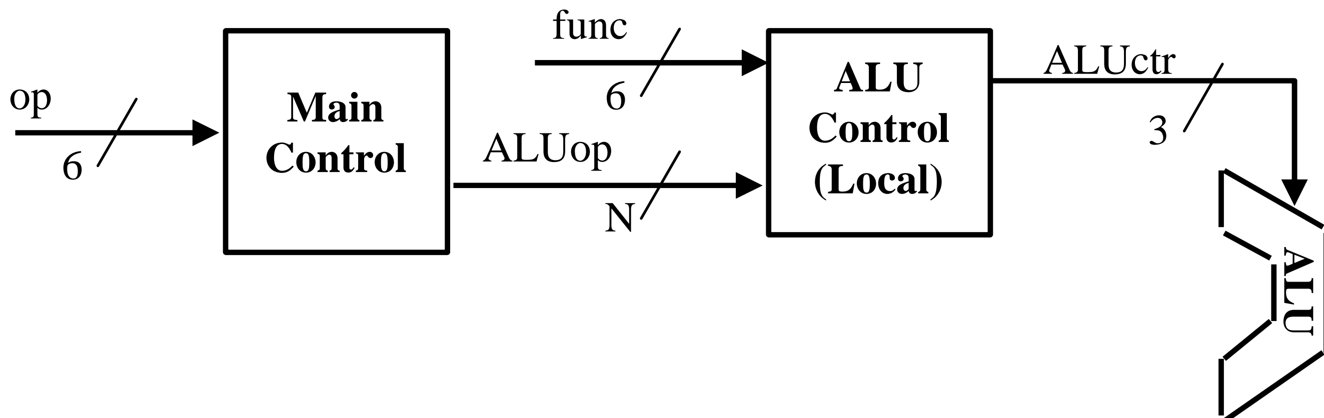
See Appendix A

| 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 |

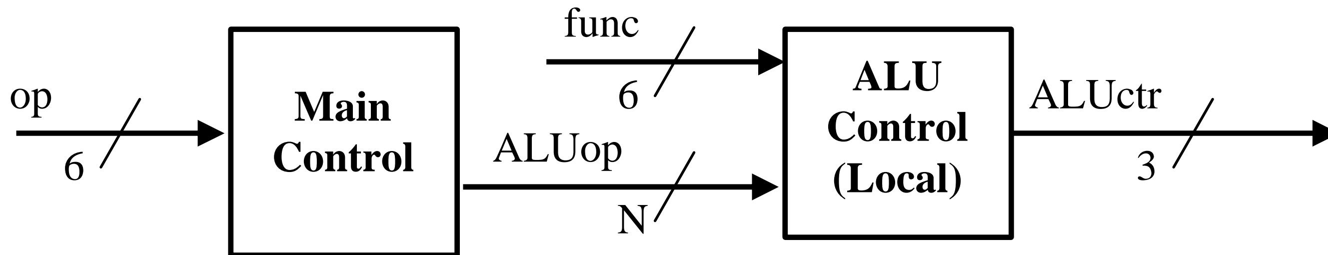


# The Concept of Local Decoding

| 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 |



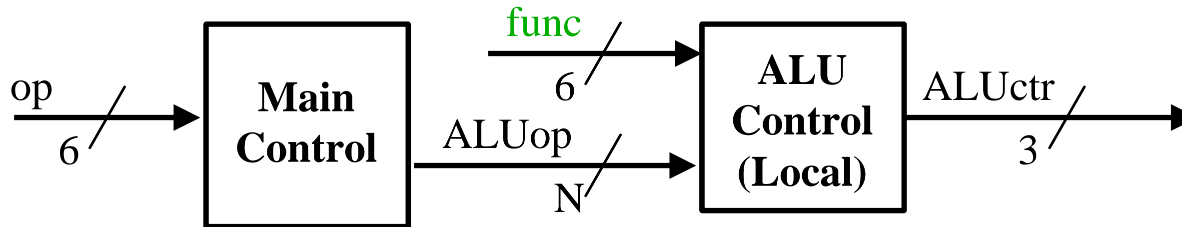
# The Encoding of ALUop



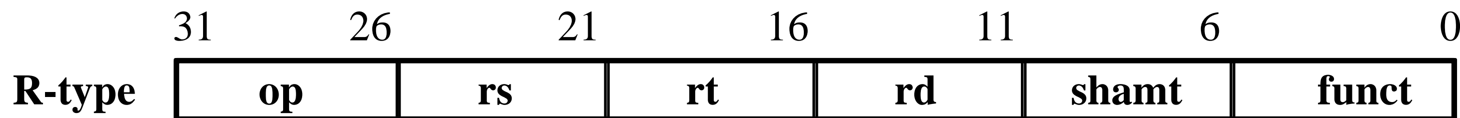

- **In this exercise, ALUop has to be 2 bits wide to represent:**
  - "R-type" instructions (1)
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, and (4) Subtract
- **To implement the full MIPS ISA, ALUop has to be 3 bits to represent:**
  - "R-type" instructions (1)
  - "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 |

# The Decoding of the “func” Field

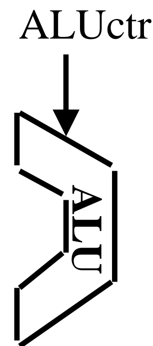



| | R-type | ori | lw | sw | beq | jump |
|---|---|---|---|---|---|---|
| **ALUop (Symbolic)** | “R-type” | Or | Add | Add | Subtract | xxx |
| **ALUop<2:0>** | 1 00 | 0 10 | 0 00 | 0 00 | 0 01 | xxx |




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

**P. 286 text:**




| ALUctr<2:0> | ALU Operation |
|---|---|
| 000 | And |
| 001 | Or |
| 010 | Add |
| 110 | Subtract |
| 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 bit<2> | ALUop bit<1> | ALUop bit<0> | func bit<3> | func bit<2> | func bit<1> | func bit<0> | ALU Operation | ALUctr bit<2> | ALUctr bit<1> | ALUctr 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

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

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

# 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 |

- **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>**

# The ALU Control Block

func
6
ALUop
3
ALU Control (Local)
ALUctr
3

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

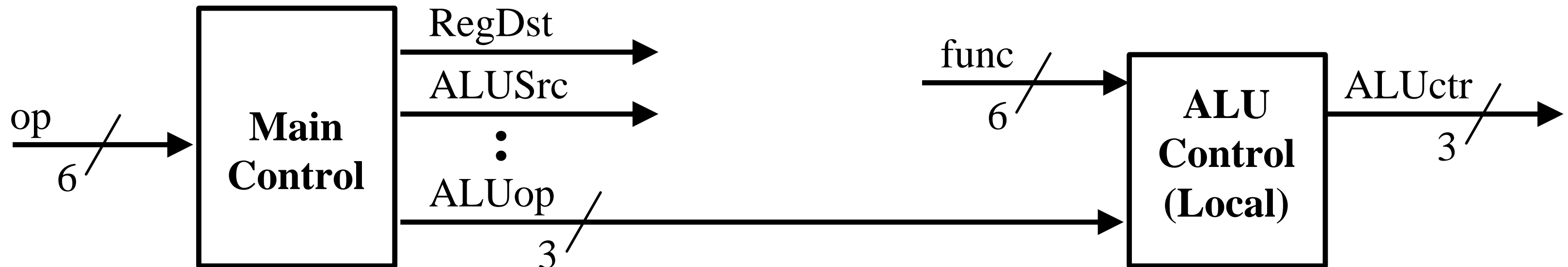
# Step 5: Logic for Each Control Signal

- **nPC_sel** <= if (OP == BEQ) then "Br" else "+4"
- **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** <= ________________
- **MemWr** <= ________________
- **MemtoReg** <= ________________
- **RegWr:** <=________________
- **RegDst:** <= ________________

# Step 5: Logic for each control signal

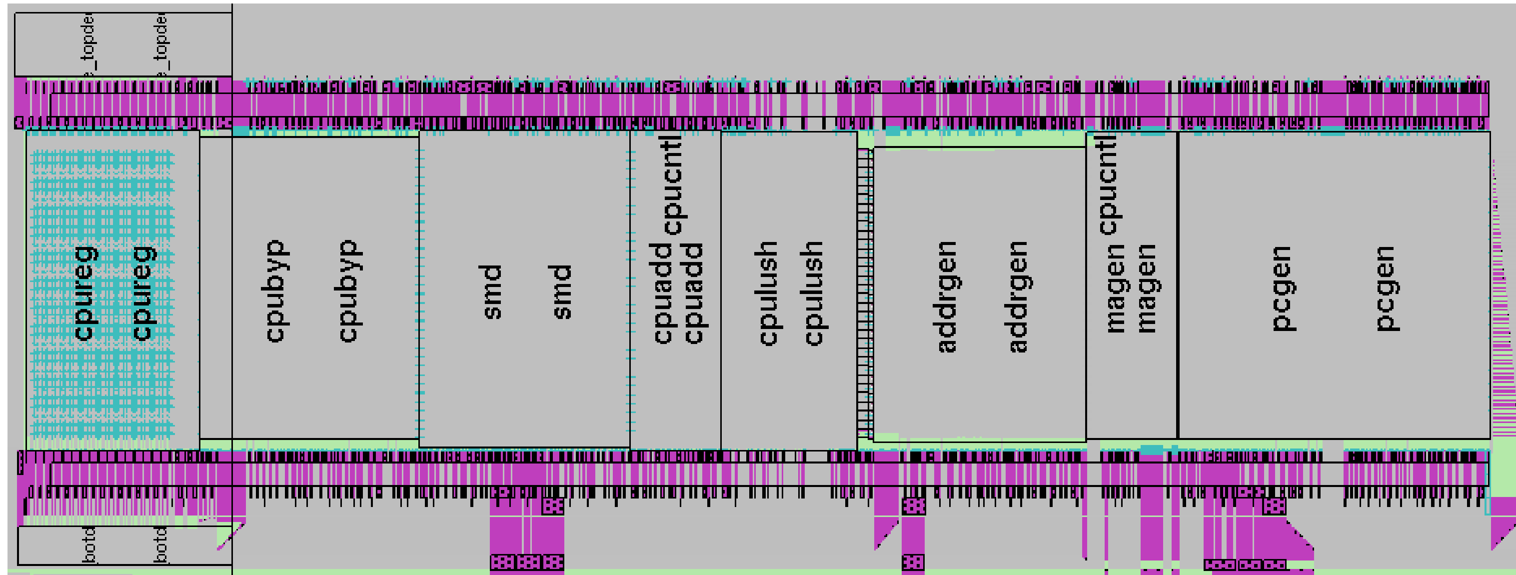
- **nPC_sel** **<= if (OP == BEQ) then "Br" else "+4"**
- **ALUsrc** **<= if (OP == "Rtype") then "regB" else "immed"**
- **ALUctr** **<= if (OP == "Rtype") then** funct
  **elseif (OP == ORi) then "OR"**
  **elseif (OP == BEQ) then "sub"**
  **else "add"**
- **ExtOp** **<= if (OP == ORi) then "zero" 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**

# The "Truth Table" for the Main Control

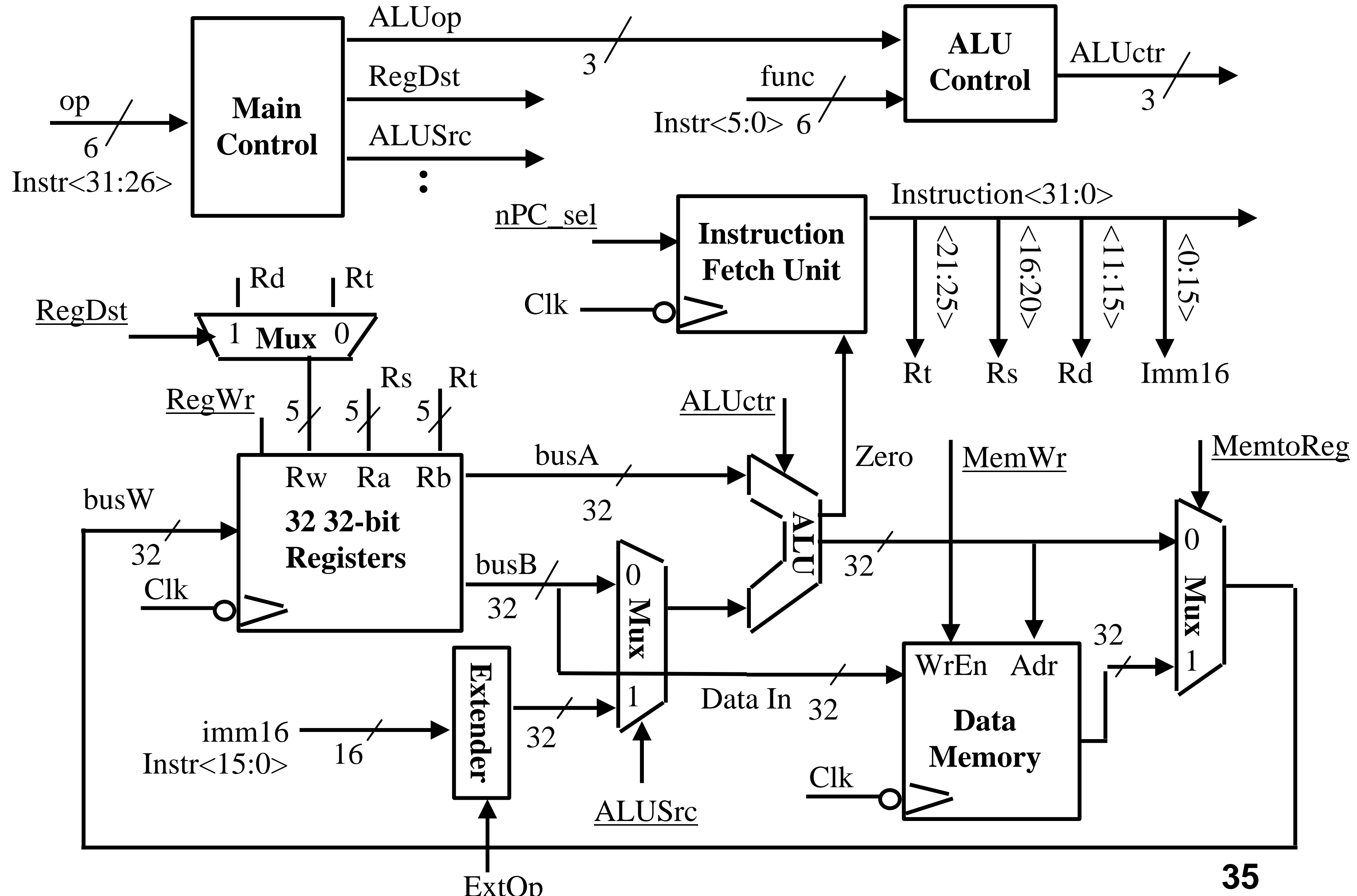



| op | 00 0000 | 00 1101 | 10 0011 | 10 1011 | 00 0100 | 00 0010 |
|---|---|---|---|---|---|---|
| | **R-type** | **ori** | **lw** | **sw** | **beq** | **jump** |
| **RegDst** | 1 | 0 | 0 | x | x | x |
| **ALUSrc** | 0 | 1 | 1 | 1 | 0 | x |
| **MemtoReg** | 0 | 0 | 1 | x | x | x |
| **RegWrite** | 1 | 1 | 1 | 0 | 0 | 0 |
| **MemWrite** | 0 | 0 | 0 | 1 | 0 | 0 |
| **nPC_sel** | 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 |

# A Real MIPS Datapath (CNS T0)

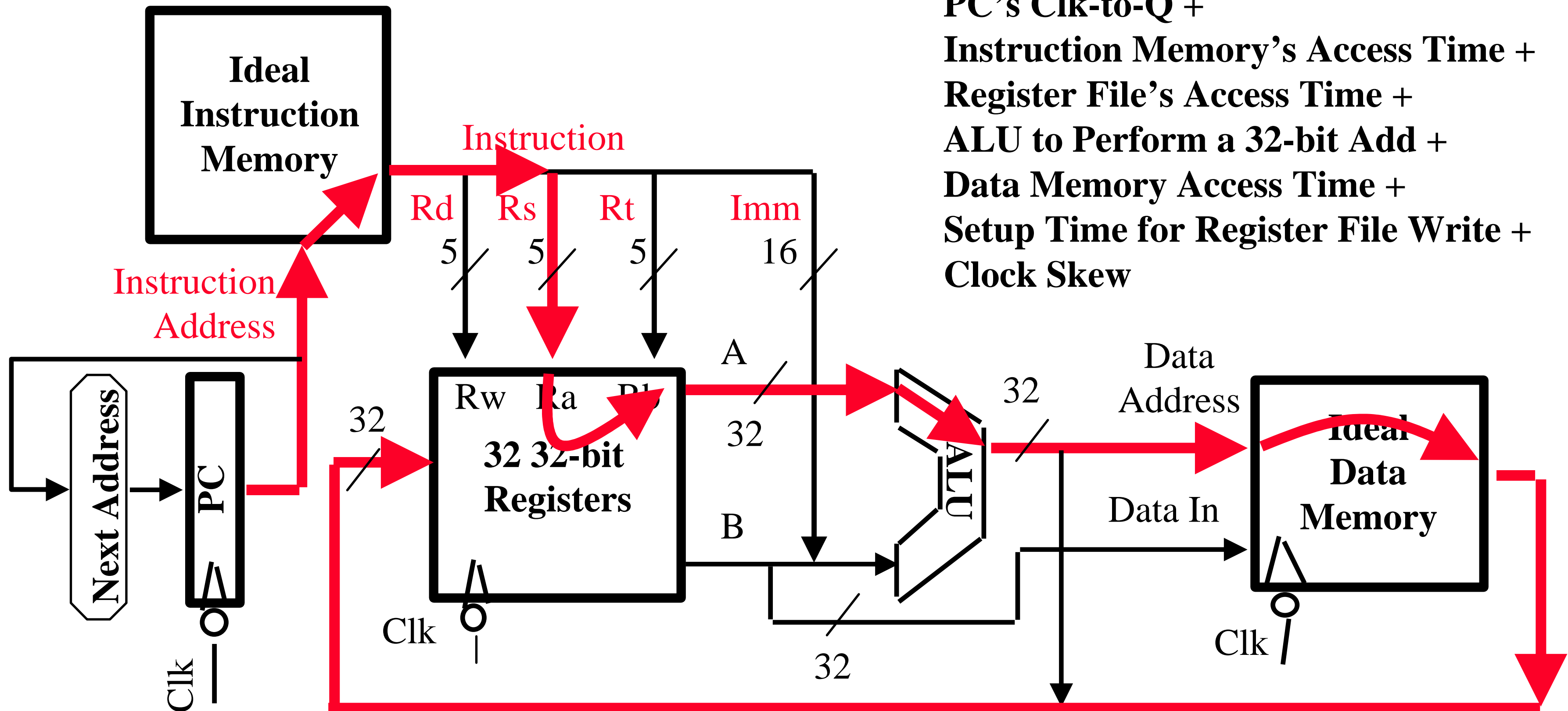


# Summary: *A Single Cycle Processor*

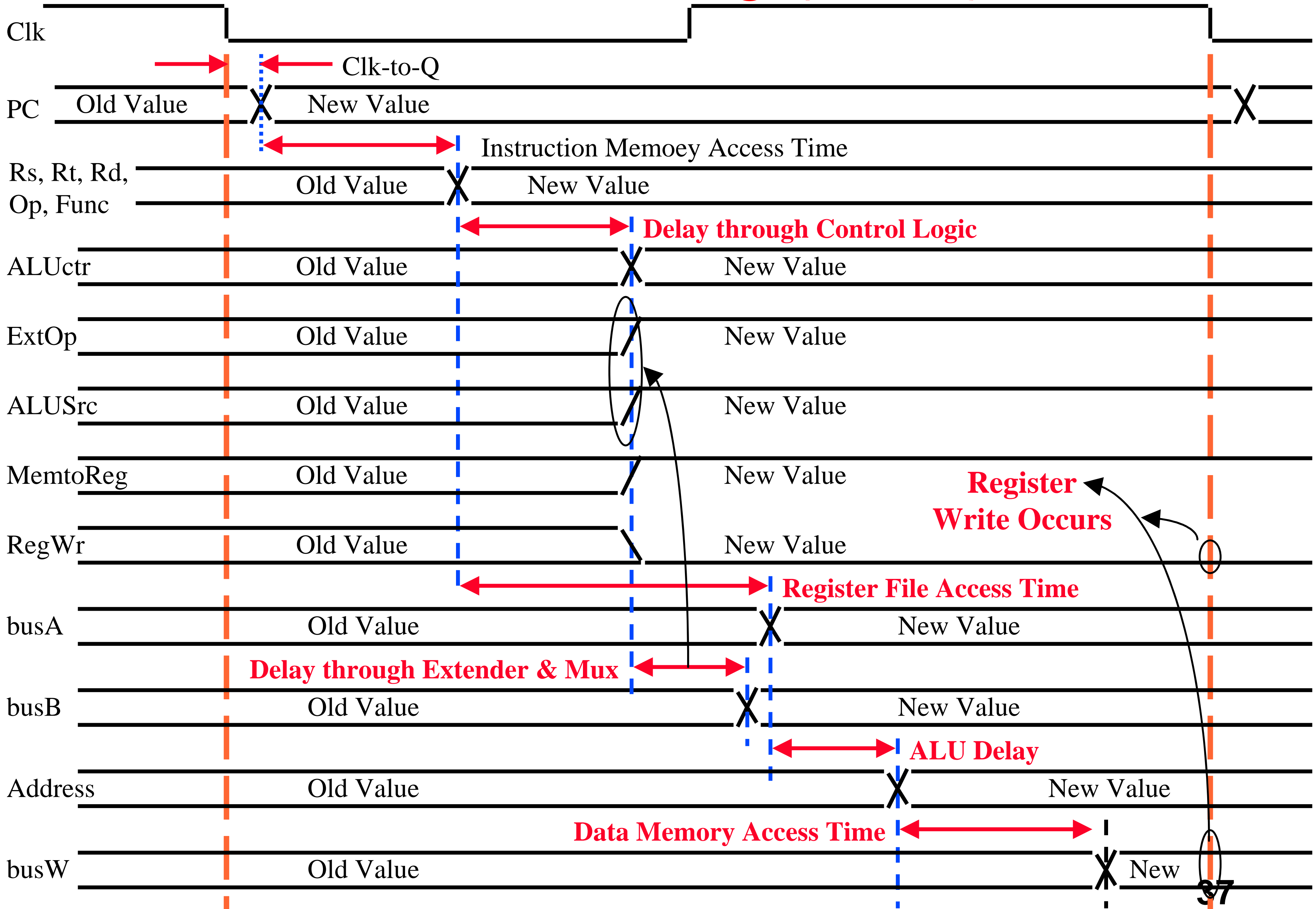


# Recap: An Abstract View of the Critical Path (Load)

- **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."



# Worst Case Timing (Load)



# Drawback of this Single Cycle Processor

- **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**

# Summary

- Single cycle datapath: CPI = 1, CCT → long
- 5 steps to design a processor
  - 1. Analyze instruction set => datapath requirements
  - 2. Select set of datapath components & establish clock methodology
  - 3. Assemble datapath meeting the requirements
  - 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  - 5. Assemble the control logic
- Control is the hard part
- MIPS makes control easier
  - Instructions same size
  - Source registers always in same place
  - Immediates same size, location
  - Operations always on registers/immediates