Timing Models

Suresh Purini, IIIT-H

When we study the design of logic circuits and understand their behavior by looking at their timing diagrams, we take an idealized view of things. We treat each circuit component as a mathematical object with well-defined behavioral model as given by its truth table. We also have a well-defined model for interconnections between various circuit components. For example refer the Timing Diagram for the positive edge triggered D-FlipFlop in the Figure 1.

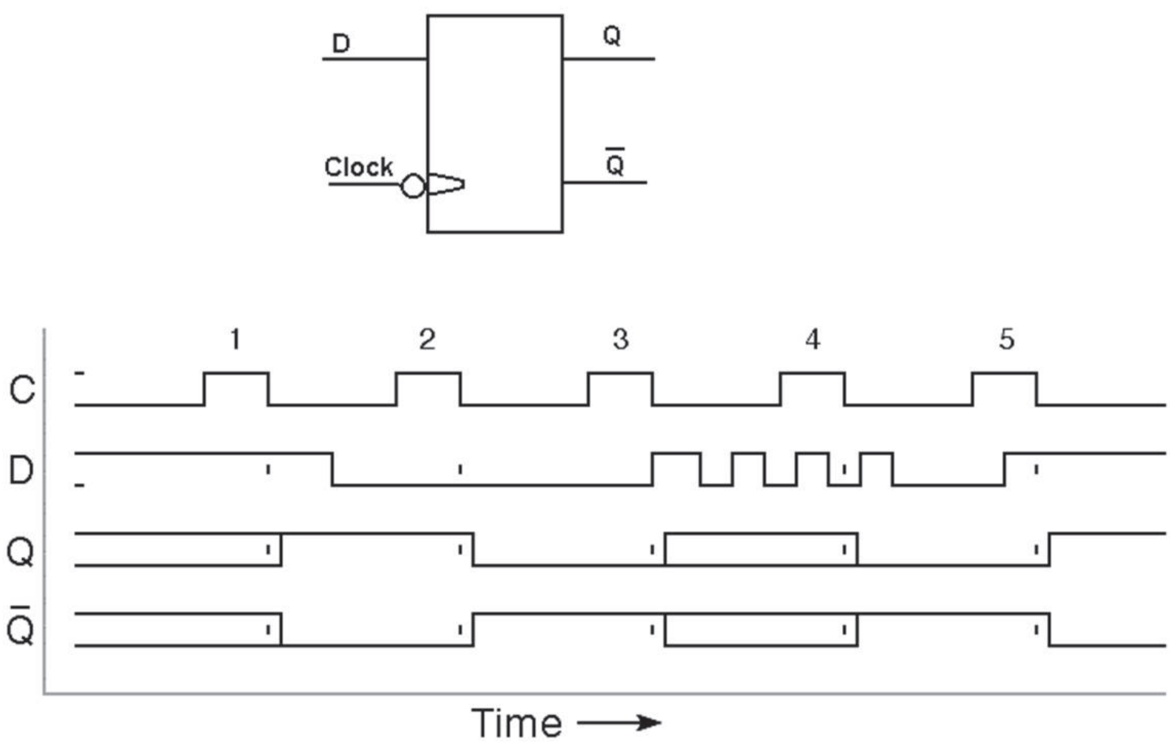


Figure 1: Timing Diagram for an Ideal Negative Edge Triggered D-Flip Flop

At the falling clock edge of the Clock Pulse 1, the D-Flip Flop input is equal to 1 and the D-Flip Flop reads the value and the output Q changes from 0 to 1 instantaneously. But that is not how it happens in any physical realization of the D-Flip Flop as against its mathematical model. In any physical realization of the D-flop the output Q will reflect the new value after certain delay. The following are three important characteristic parameters of the circuit components which are technology dependent (refer Figure 2).

1. **Set Up Time:** Setup time (denoted as t_{su}) is the minimum amount of time the data signal

should be held steady before the clock event so that the data are reliably sampled by the clock.

2. **Hold Time:** Hold time (denoted as t_h) is the minimum amount of time the data signal should be held steady after the clock event so that the data are reliably sampled.

3. **Clock-to-Q Time:** Clock-to-Q Time (denoted as t_{cq}) is the time it takes for the flip-flop to change its output after the clock edge.

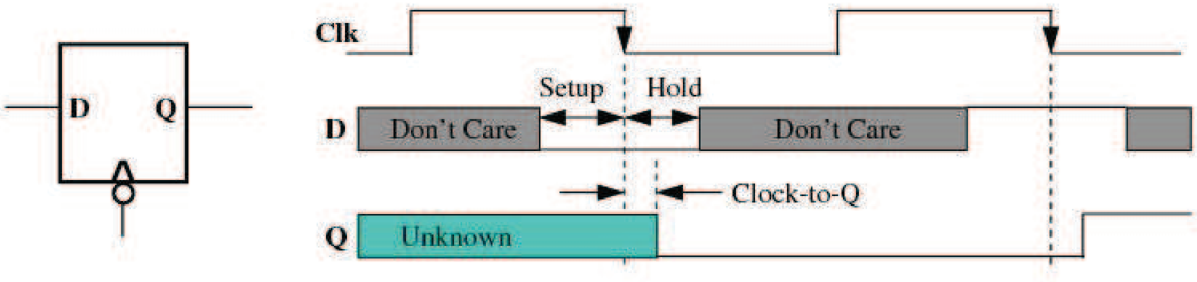


Figure 2: Diagram illustration Setup, Hold and Clock-to-Q Times.

You might wonder whether there is any relationship between the three parameters t_{su}, t_h and t_{cq}. Especially between the parameters t_h and t_{cq}, intuitively we feel there could be a relationship[1] For the sake of our discussion we can safely assume that all the three parameters are independent and their values are dictated by the underlying device physics. We shall now see how we have to factor in these parameters while designing logic circuits. Consider the Figure 3 where the output of the register R_1 is connected to the input of the register R_2 through some combinational logic circuit.

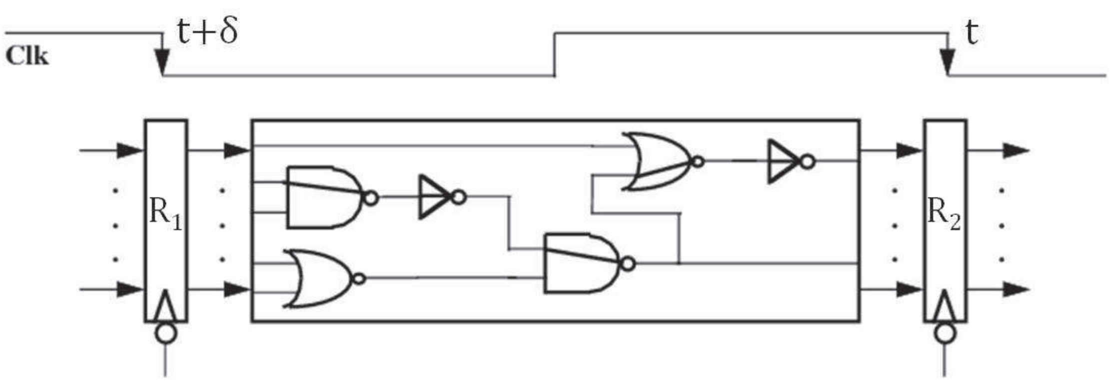


Figure 3: Two registers connected through intermediate combinational logic

At time instant t a falling clock edge arrives at both the registers R_1 and R_2 simultaneously (assuming no clock skew). The registers R_1 and R_2 sample their respective input signals at that falling clock edge and update their states accordingly and the updated state is available on the

[1] I am not aware of any such relation. However I have to double check before I give a definite answer to you all.

output after the Clock-2-Q time which would be at the time instant $t + t_{cq}$. Remember the input signals to the registers R_1 and R_2 should be stable during the interval $[t - t_{su}, t + t_h]$ for the proper update of the registers. Now let us focus on how the signals flow between the registers R_1 and R_2 alone. The updated output signals of the register R_1 which are available at the time instant $t + t_{cq}$ flow through the combination logic and reach the input of the register R_2. The combinational logic would induce some delay before the input signals to the register R_2 gets stabilized. This delay value is equal to the sum of the logic gate delays along the longest path through the combinational logic. This longest path is also called as the critical path. Let t_{cl} be the delay along this critical path. So the new input signals to the register R_2 will be available by the time instant $t + t_{cq} + t_{cl}$. It is important to note that during the time interval $[t + t_{cq}, t + t_{cq} + t_{cl}]$, the input signals to the register R_2 may oscillate (why?).

We expect that the newly available input signals to be latched onto the register R_2 at the next falling clock edge which arrives at the time instant $t + \delta$ where δ is the clock cycle length. If we factor in the setup time requirements for the proper update of the register R_2, we will get the following necessary condition on the clock cycle time.

$$\begin{aligned} t + t_{cq} + t_{cl} + t_{su} &\leq t + \delta \\ \delta &\geq t_{cq} + t_{cl} + t_{su} \end{aligned}$$

If the above condition is not satisfied, then we say there is a **setup time violation**. In general if t_{cl}^{ij} is the critical path length in the combinational logic connecting two registers R_i and R_j, then the following condition has to be satisfied.

$$\delta \geq t_{cq} + t_{cl}^{ij} + t_{su}$$

Assuming that the Clock-to-Q and setup times are same for all the registers, the clock cycle length is governed by the largest crictical path length. For example if the largest critical path correponds to the combinational logic between two registers R_m and R_n then we can choose the clock cycle length as $\delta = t_{cq} + t_{cl}^{mn} + t_{su}$.

There is another interesting problem that we can face during logic circuit design. Let us consider a path in the combinational logic circuit connecting the registers R_1 and R_2. Let t_p be the delay along this path. So after the falling clock edge occuring at time instant t, a signal can reach the input of the register R_2 by time $t + t_{cq} + t_p$. If it turns out that $t + t_{cq} + t_p \in [t - t_{su}, t + t_h]$ then the new signal value will over write the old input signals to register R_2 causing an instability of the input. We call this as **hold time violation**. To prevent hold time violation we need to make sure that for every path in the combinational logic circuit with delay t_p the following condition is true.

$$t + t_{cq} + t_p > t + t_h$$

If there is a path in the combinational circuit which does not satisfy the aforementioned condition, then the delay along the path has to be increased by adding buffer elements.

In the previous discussion we assumed that a following clock edge arrives at both the registers exactly at the same time instant. However in practice due to wire delays arising out of wire resistance and other physical characteristics of the wires, the same falling clock edge could reach different registers at different time instants (refer Figure 4). Let t_{cs} be the clock skew. Due to the clock skew the falling clock edge which is arriving at register R_2 at time instant t would now arrive at time instant $t + t_{cs}$. However it can so happen that during this time interval $[t, t + t_{cs}]$, along some path in the combination logic the new signal would flow and overwrite the old input signal to the register R_2. This can happen if

$$t + t_{cq} + t_p \leq t + t_{cs} + t_h$$

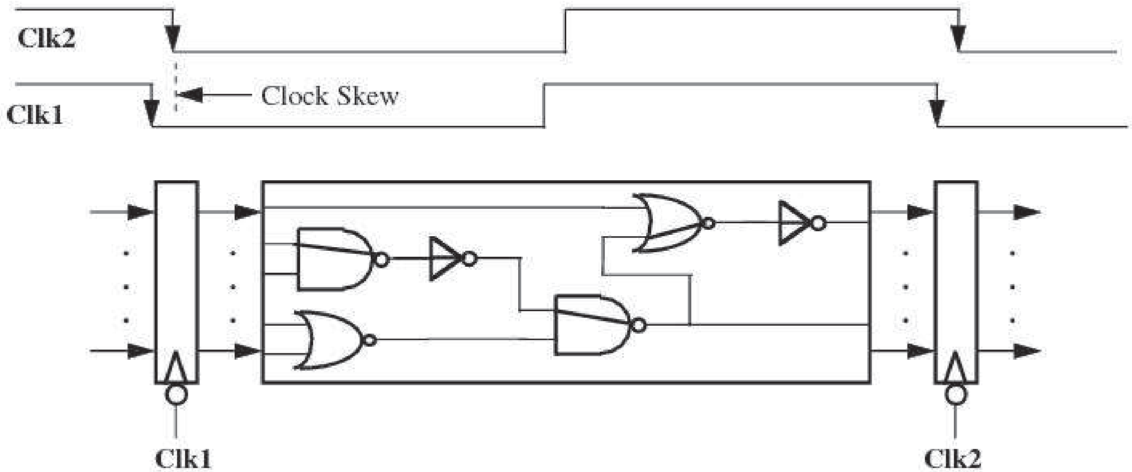


Figure 4: Clock skew due to clock signal propagation delays

where t_p is delay along the combination logic path under consideration. Refer to the Figures 5 and 6 which illustrate this problem.

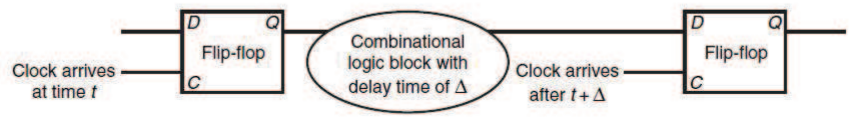


Figure 5: Potential problems due to clock skew

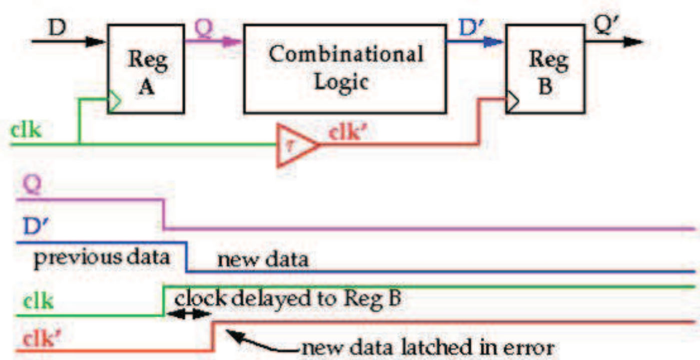


Figure 6: Potential problems due to clock skew

Representation of Integers and their Arithmetic

Suresh Purini, IIIT-H

What does the 8-bit string 11100000 represent? It could mean 224, -96, -31 and -32 when treated as an unsigned integer, sign-magnitude integer, one's complement integer and two's complement integer respectively. Or it could be mean the ASCII character α. So what a bit string means depends on the semantics or the definition we associate with it. In this write-up we shall study binary representation of unsigned and signed integers which is a primitive data structure supported by all modern processors.

1 Unsigned Integers

Consider the bijective function $B2U_w : \{0,1\}^w \rightarrow \{0, \cdots, 2^w - 1\}$ which maps w-bit binary strings to unsigned integers as follows.

$$B2U_w(\vec{b}) = \sum_{i=0}^{w-1} b_i 2^i$$

For example $B2U_4(0101) = 5$ and $B2U_4(1101) = 13$. You can observe that the function $B2U_w$ and its inverse are efficiently computable. In other words, we can easily compute the binary representation of an unsigned integer in the range of the function making it a viable representation.

In C-language all variables of type unsigned integers are allocated a fixed number of bytes (or equivalent number of bits) for storage which is typically 4 bytes or 32 bits. You can check this by running the following C-program on your machine.

```
#include <stdio.h>
main()
{
   printf("Size of Unsigned Integer: %d\n", sizeof(unsigned int));
}
```

Having represented unsigned integers in binary, we would like to figure out how to perform addition and multiplication operations. Let us just focus on addition operation in our discussion and the relevant ideas can be applied to multiplication operations too with suitable modifications. We presume that you know the algorithm to add two binary numbers as illustrated in the Figure 1 [1]. We also know the analogous algorithm for addition in the unsigned integer domain. Now the beauty of the mapping function $B2U_w$ is that it shows the isomorphic structure between the unsigned integers and their binary representation with respect to the addition operation (and also multiplication operation). To elaborate more on this idea let us define the w-bit addition of two numbers as the regular binary addition except that we ignore the carry-out bit from the MSB if at all there is one. With this definition, when we add two w-bit numbers, the result is always a w-bit number. The key claim here is whether we do addition of two unsigned integers in decimal notation

[1] Recall the w-bit ripple carry adder circuit.

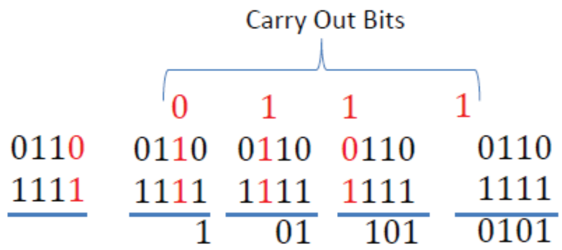


Figure 1: Addition of binary numbers

Row No	3-bit Binary	Unsigned Integer
R_0	000	0
R_1	001	1
R_2	010	2
R_3	011	3
R_4	100	4
R_5	101	5
R_6	110	6
R_7	111	7

Table 1: Isomorphic structure of 3-bit binary numbers and their unsigned intrepetation

which we are familiar with or we do addition of their respective w-bit binary representations, the net result is just the same except for the difference in their notational representation. This claim is true so far as the result of the addition operation does cause an overflow, in other words the result would fit into w-bits. Consider the following Table 1 with three columns. If we want to add 1 and 4, whether we carry out addition in column 2 or in column 3, the respective results would fall in Row 5. However if we want to add 4 and 5, then the result wouldn't fall in the range in the column 2 and the result in the column 3 would fall in Row-1 (recall how w-bit addition is defined). It can be observed though that there is isomorphism between ($mod\ 2^w$) addition of decimal numbers and w-bit addition of binary numbers without worrying about overflow at all since it would never happen in modular arithmetic. It has to be noted here that we can use any other function (preferably bijective) from the w-bit strings to unsigned numbers and create a isomorphism between the decimal domain and the binary domain by appropriately defining the addition operations in the binary domain. We leave it to you to ponder whether such an alternate function has any utility. It is easy to note here that the addition of two w-bit unsigned numbers would cause an overflow if and only if the carry-out bit is 1.

2 Signed Integers

The following are three different ways of representing signed integers.

1. Sign-Magnitude Representation. The mapping function here is:

$$B2S_w(b_{w-1}...b_0) = (-1)^{b_{w-1}} * (2^{w-2} * b_{w-2} + ... + 2^0 * b_0)$$

Row No	3-bit String	Sign-Magnitue	1's Complement	2's Complement
R_0	000	0	0	0
R_1	001	1	1	1
R_2	010	2	2	2
R_3	011	3	3	3
R_4	100	-0	-3	-4
R_5	101	-1	-2	-3
R_6	110	-2	-1	-2
R_7	111	-3	0	-1

Table 2: Isomorphic structure of 3-bit binary numbers and 2's complement signed integers

2. 1's Complement Representation. The mapping function here is:

$$B2O_w(b_{w-1}...b_0) = -b_{w-1} * (2^{w-1} - 1) + b_{w-2} * 2^{w-2} + ... + b_0 * 2^0$$

3. 2's Complement Representation. The mapping function here is:

$$B2T_w(b_{w-1}...b_0) = -b_{w-1} * 2^{w-1} + b_{w-2} * 2^{w-2} + ... + b_0 * 2^0$$

Pretty much all systems use 2's complement representation for signed integers. We shall see the rationale behind such a choice in the following discussion. First you can verfiy that among the 3 mapping functions only the $B2T_w$ function corresponding to 2's complement representation is bijective. Let us stick to our definition of w-bit addition of binary numbers and we shall see that there is an isomorphic structure between signed integers and their 2's complement representation with respect to addition. It has to be noted that this isomorphism holds if and only if the results of addition does not cause overflow or underflow. Sign-magnitude and 1's complement representation of signed integers doesn't carry this isomorphic structure with respect to the canonical binary addition rules. It is worth noting that we can create an isomorphic structure even with these representations by suitable modifying the rules of binary addition. To understand these ideas consider the Table 2. For example if we add Row3 with Row4, the resulting binary number is 111 which lies in Row 7, whereas if we perform the addition on Sign-Magnitude numbers in Column 2, we get a value in Row 3 indicating the lack of isomorphic structure with respect to addition between the binary and sign-magnitude representation of numbers. It can be verified that there is no isomorphic structure between binary and one's complement representation of numbers by adding elements in Row 5 and Row 6. In binary addition we get an element in Row 3, whereas in the one's complement representation we get an element in Row 4 in Column 3. However it can be verified that as long as there is no overflow there is a perfect isomorphism with respect to addition between binary and two's complement representation of numbers.

3 Unsigned versus 2's Complement Addition

From the previous discussion it could have been noted that the rules of binary addition for both Unsigned and 2's Complement Addition is exactly the same. It means that we could use the same k-bit ripple carry to add any 2 unsigned or 2's complement numbers and we need not tell the k-bit ripple adder whether we are doing signed arithmetic or unsigned arithmetic. To illustrate this point further let us that I have a k-bit adder circuit with me, some of the students in the class want to do 2's complement addition and some of you may want to perform unsigned addition over k-bit numbers using my k-bit adder circuit. But you don't want to reveal me whether you are performing signed or unsigned arithmetic for whatever reasons you have. It is no big deal for my k-bit adder circuit as the rules of addition remains the same for both signed and unsigned numbers. However there is a catch here. The catch is that overflow conditions for signed and unsigned arithmetic are different.

Predication in ARM ISA

Suresh Purini, IIIT-H

January 29, 2012

In the ARM ISA, for almost all instructions we can add a predicate as a suffix which tells the processor to execute the corresponding instruction only if the respective conditions associated with the predicate are satisfied (refer Table)1. For example when we add the predicate MI to an ADD instruction to get an instruction of the form ADDMI, then the following things happen:

- The assembler packs the bits 0100 in the most significant nibble while construction constructing the 32-bit opcode of the ADDMI instruction. This is an *offline* task that happens at the program assembly time.

- When the processor encounters the ADDMI instruction, it executes it if and only if the N flag is equal to 1 (or set in other words). This is an *online* task that happens at the program execution time.

Usually we use this predicate table in conjunction with the CMP instruction [1]. A brief description of the CMP instruction follows. When the processor encounters CMP r1, r2 instruction, it will subtract the register $r2$ from $r1$ and uses the result to affect the N, Z, C, V bits of the CPSR register as follows. Let `alu_out = r1 - r2`.

- `N = alu_out[31]`
- `Z = if alu_out == 0 then 1 else 0`
- `C = if r1 >= r2 then 1 else 0 (assuming r1 and r2 as unsigned integers)`
- `V = 1 if the result of subtraction does not fit into 32 bits else 0 (assuming r1 and r2 as signed integers)`

Finally `alu_out` will be discarded without storing it in any register.

After the CMP instruction, we have to use the right predicate suffix to indicate the condition we would like to test. For example if we treat r1 and r2 as unsigned integers and want to check if $r1 <= r2$, then we have to use the suffix LS (for example BLS). However if treat r1 and r2 as signed integer to test the same condition we use the suffix LE (for example BLE). As assembly language programmers this much knowledge is sufficient for us to write our programs correctly. However as processor designers and computer scientists, we are interested in checking whether the condtions mentioned in the Column 3 and Column 4 of the Table 1 are equivalent or not. For example is the following statement true after the exectuion of the CMP r1, r2 instruction.

$$r1 \geq r2 \ \ \text{(signed)} \iff N = V$$

Question: Check the equivalence of the conditions in Column 3 and Column 4 of the Table 1 are equivalent (after the execution of the CMP instruction).

[1]We can use these predicates in conjunction with other instructions also. For example SUBS instruction followed by ADDEQ.

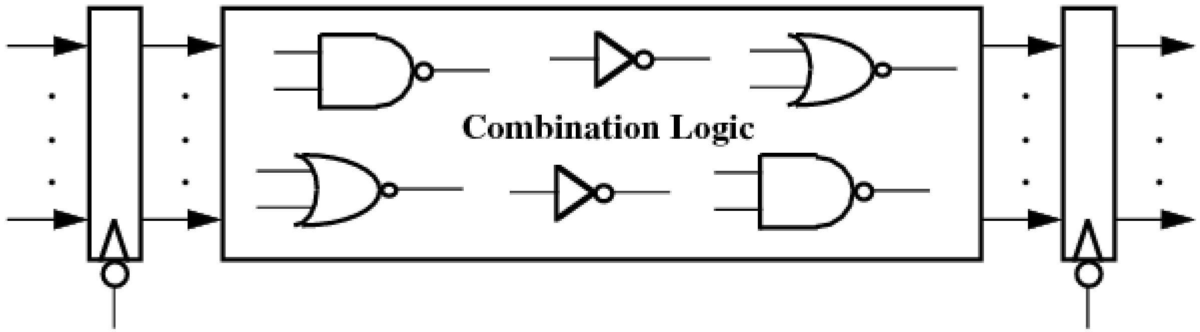
Opcode [31:28]	Extension	Interpretation	Status flag state for execution
0000	EQ	Equal/equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsgined lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

Table 1: ARM Predicate Table

Acknowledgment: Almost all of these slides are based on Dave Patterson’s CS152 Lecture Slides at UC, Berkeyley.

COMPUTER SYSTEMS ORGANIZATION

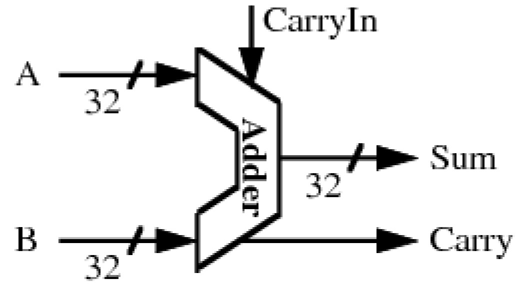
Sequential and Combinational Circuits



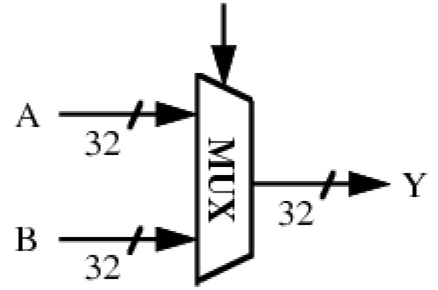
- What's the difference between sequential and combinational circuits?

Combinational Logic Elements

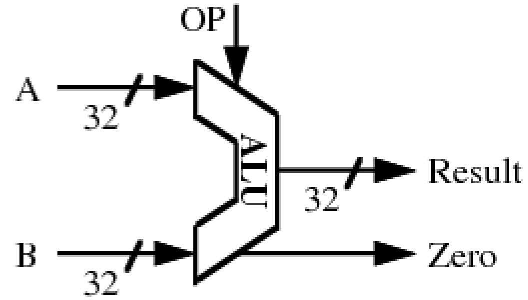
- **Adder**



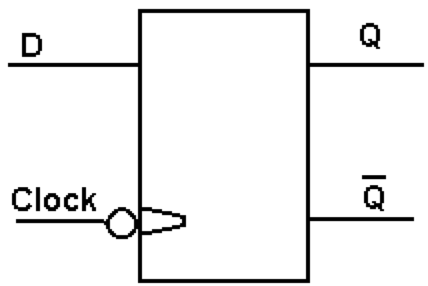
- **MUX**

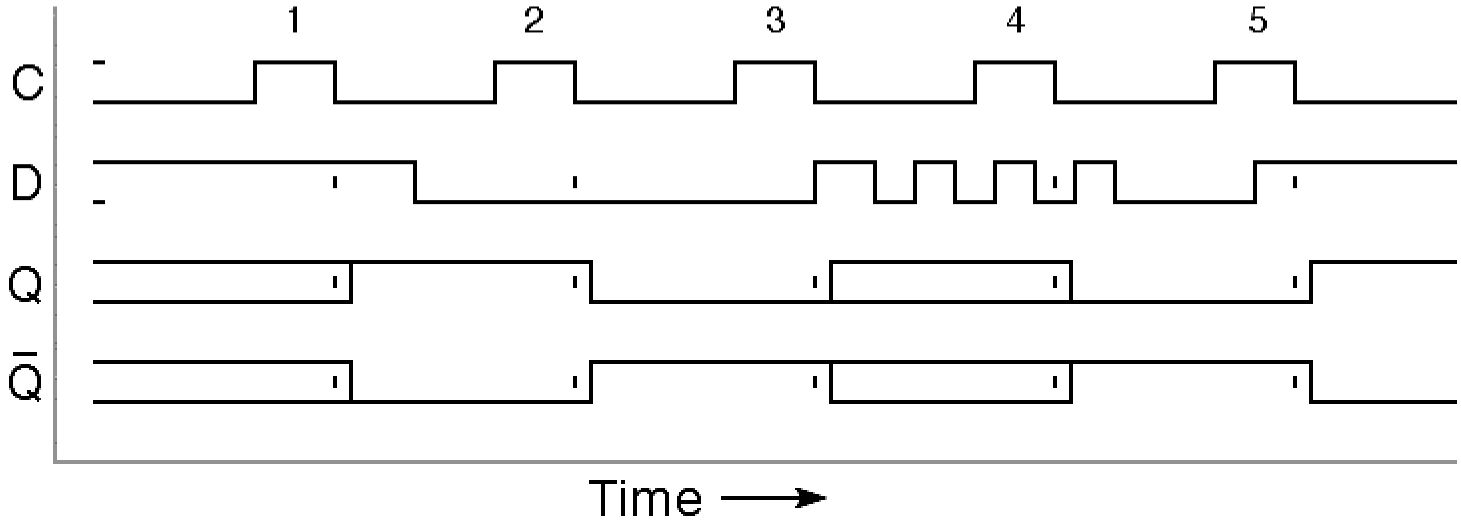


- **ALU**

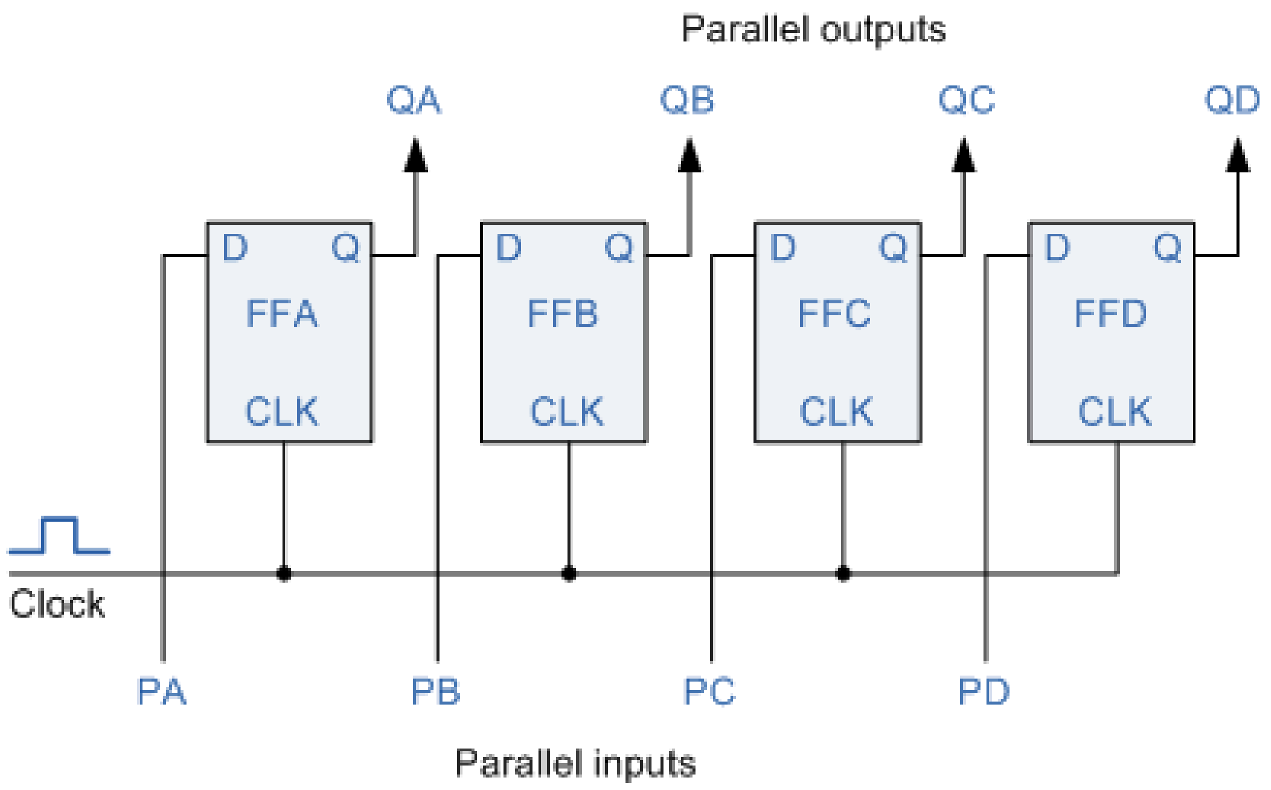


Sequential Element: Negative Edge Triggered D-Flip Flop



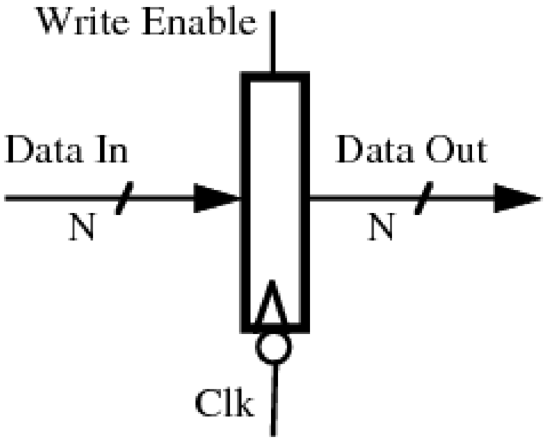


4-bit Register

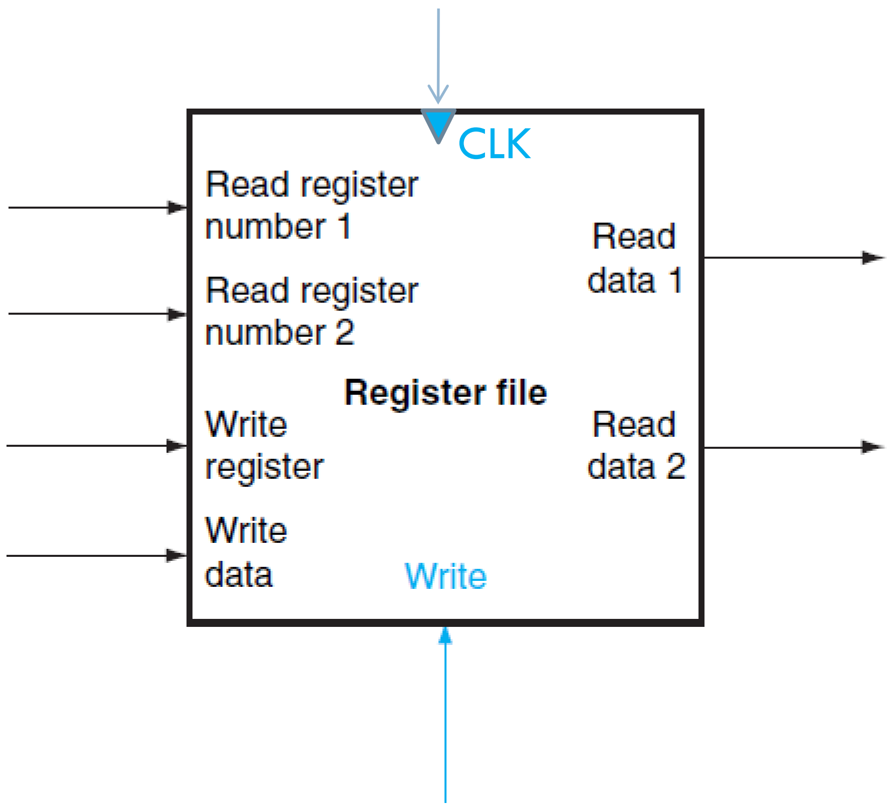


Sequential Element: Register

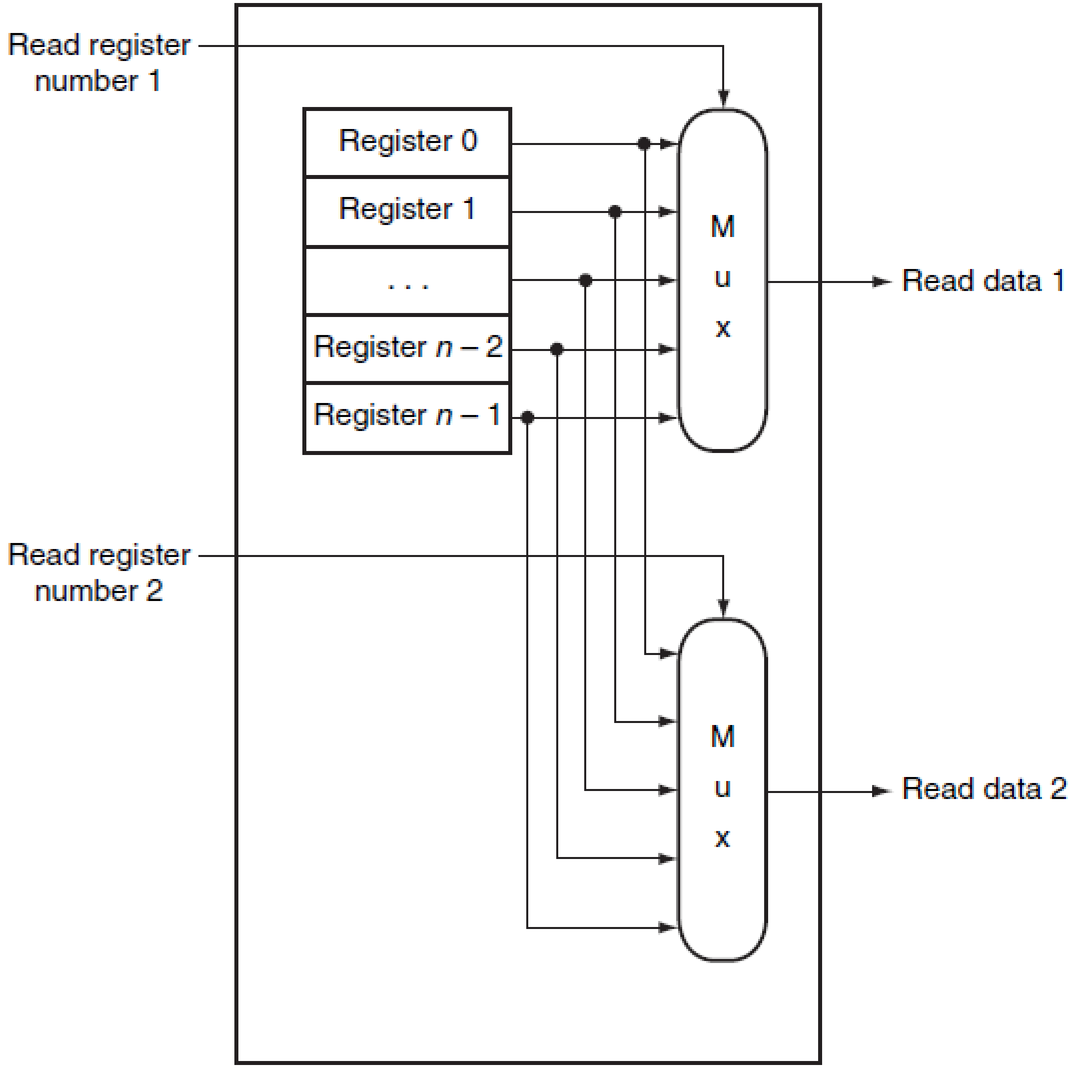
- Register
  - Similar to D Flip Flop except
    - N bit input and output
    - Write Enable input
  - Write Enable
    - 0: Data out will not change
    - 1: Data out will become Data In



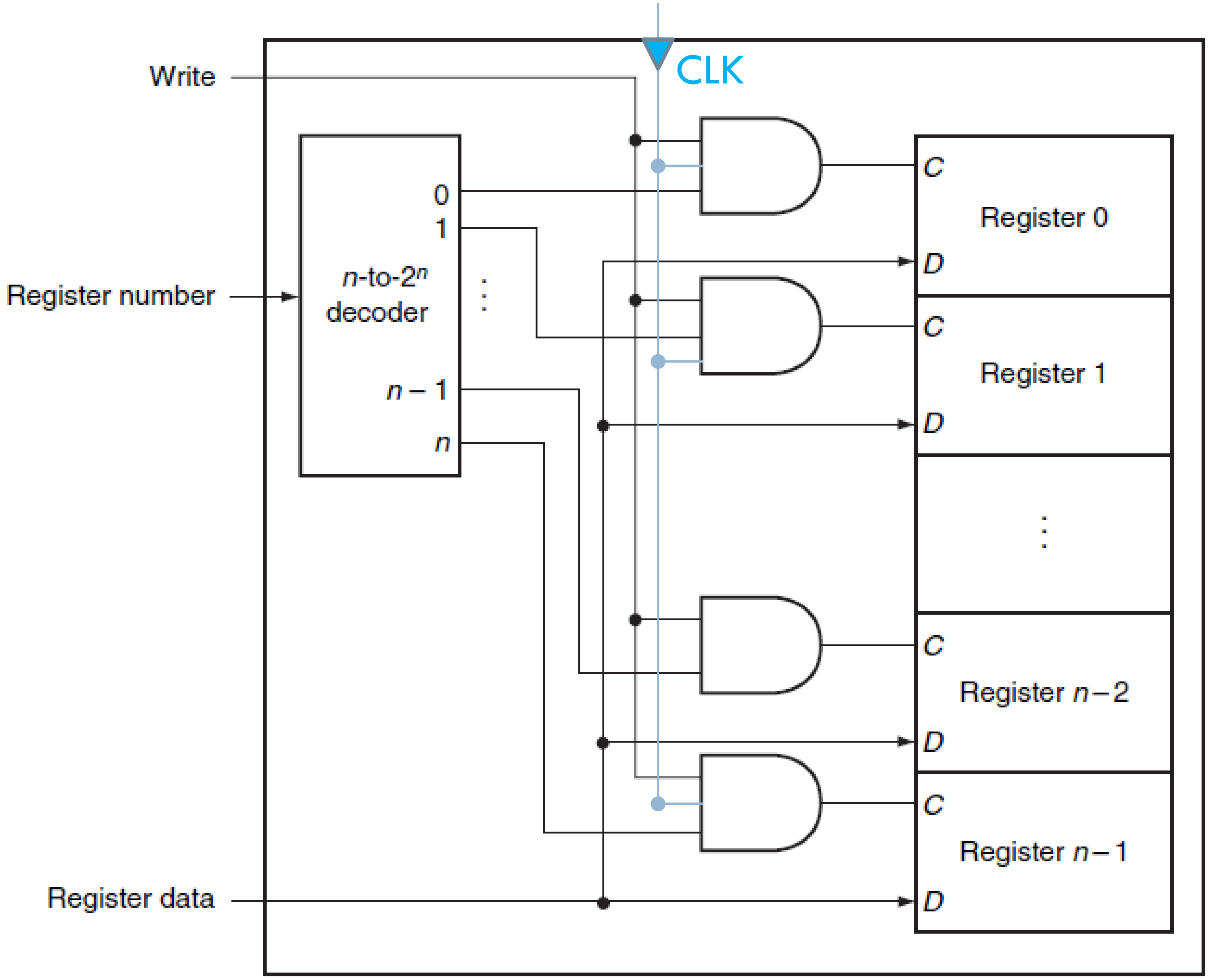
Register File



Register File

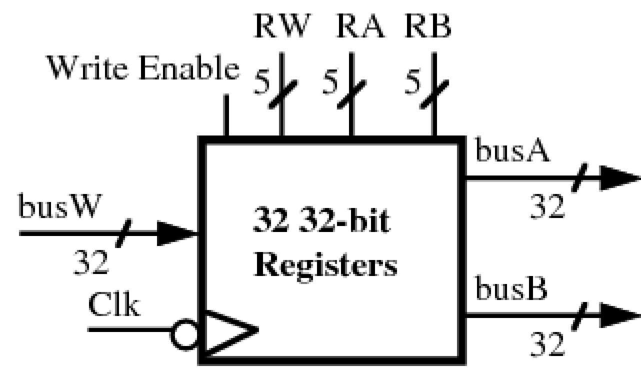


Register File

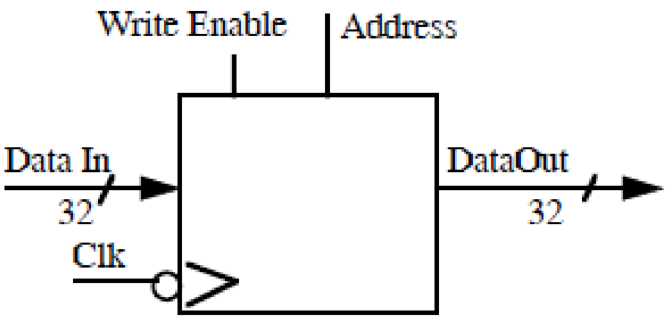


Storage Element: Register File

- Register File consists of 32 registers
  - Two 32-bit output busses: busA and busB
  - one 32-bit input bus: busW
- Register is selected by
  - RA selects the register to put on busA
  - RB selects the register to put on busB
  - RW selects the register to be written via busW 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

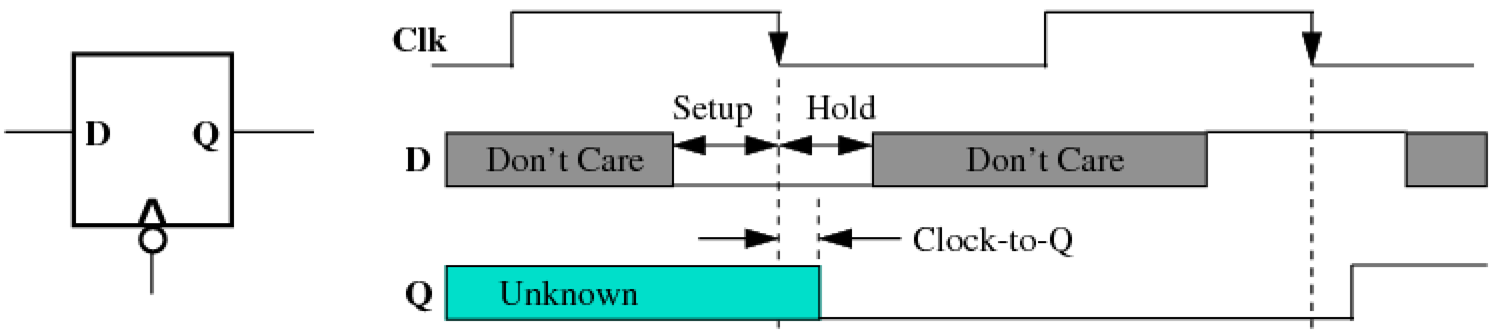


Storage Element: Memory



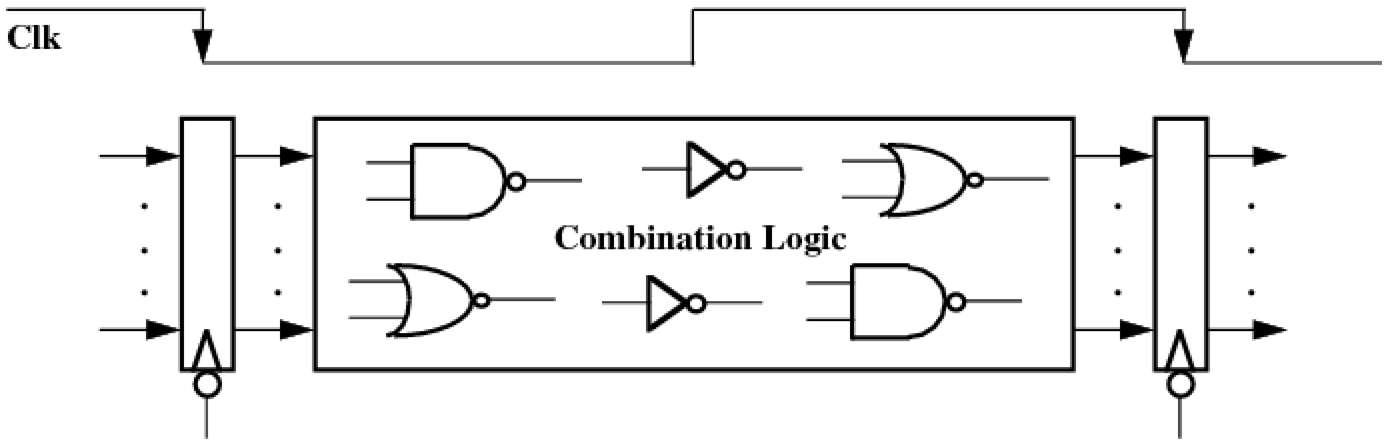
- Memory
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is selected by:
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - Address valid => Data Out valid after "access time."

Storage Element Timing Model – Negative Edge Triggered D-Flip Flop



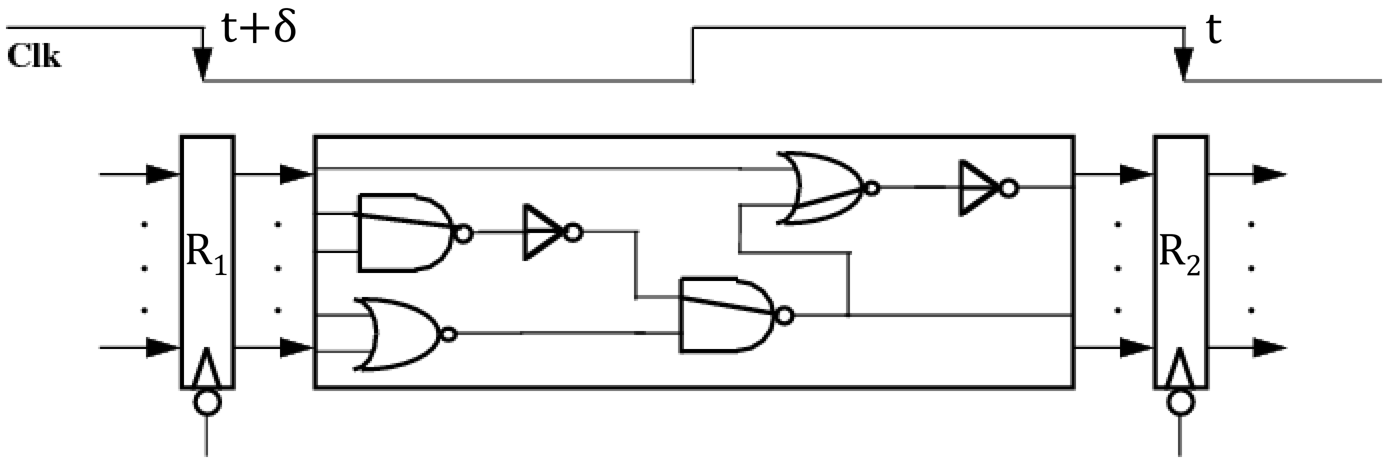
- Setup Time: Input must be stable BEFORE the trigger clock edge.
- Hold Time: Input must be stable AFTER the trigger clock edge.
- Clock-to-Q time: Output cannot change instantaneously at the trigger clock edge.
  - Similar to delay in logic gates.

Clocking Methodology



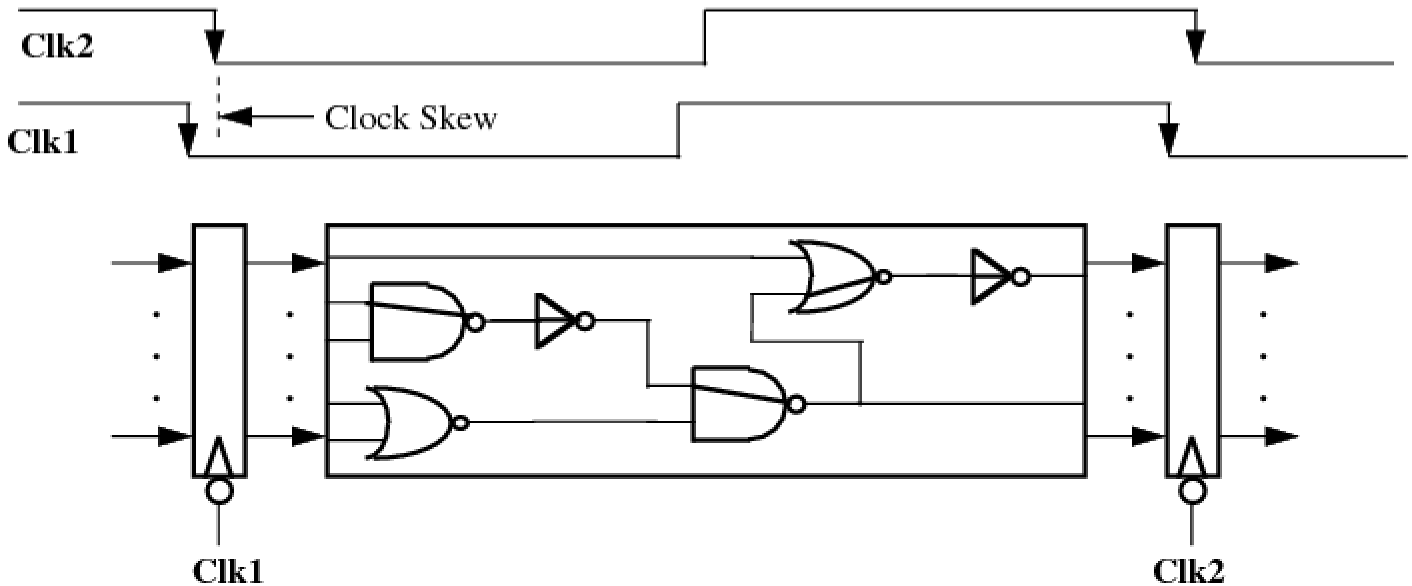
- All storage elements are clocked by the same clock edge
- The combination logic block's:
  - Inputs are updated at each clock tick
  - All outputs MUST be stable before the next clock tick

Critical Path and Cycle Time

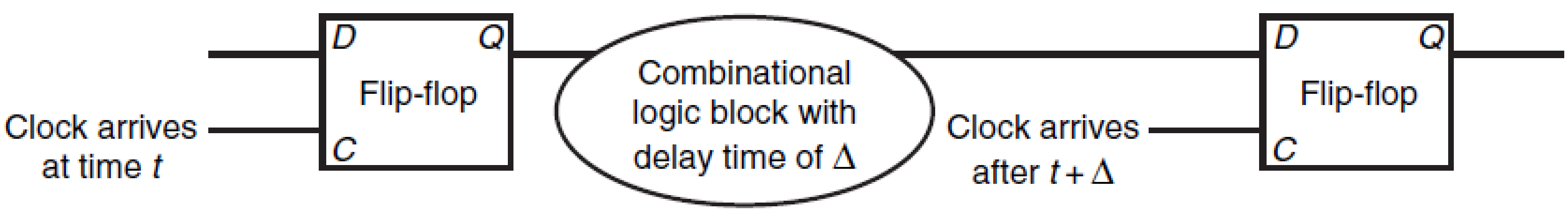


- **Critical Path:** Slowest path between any two storage devices
- Cycle time is a function of critical path
- More specifically, the cycle time must be greater than:
  - Clock-to-Q + Longest Path through the Combinational Logic + Setup Time

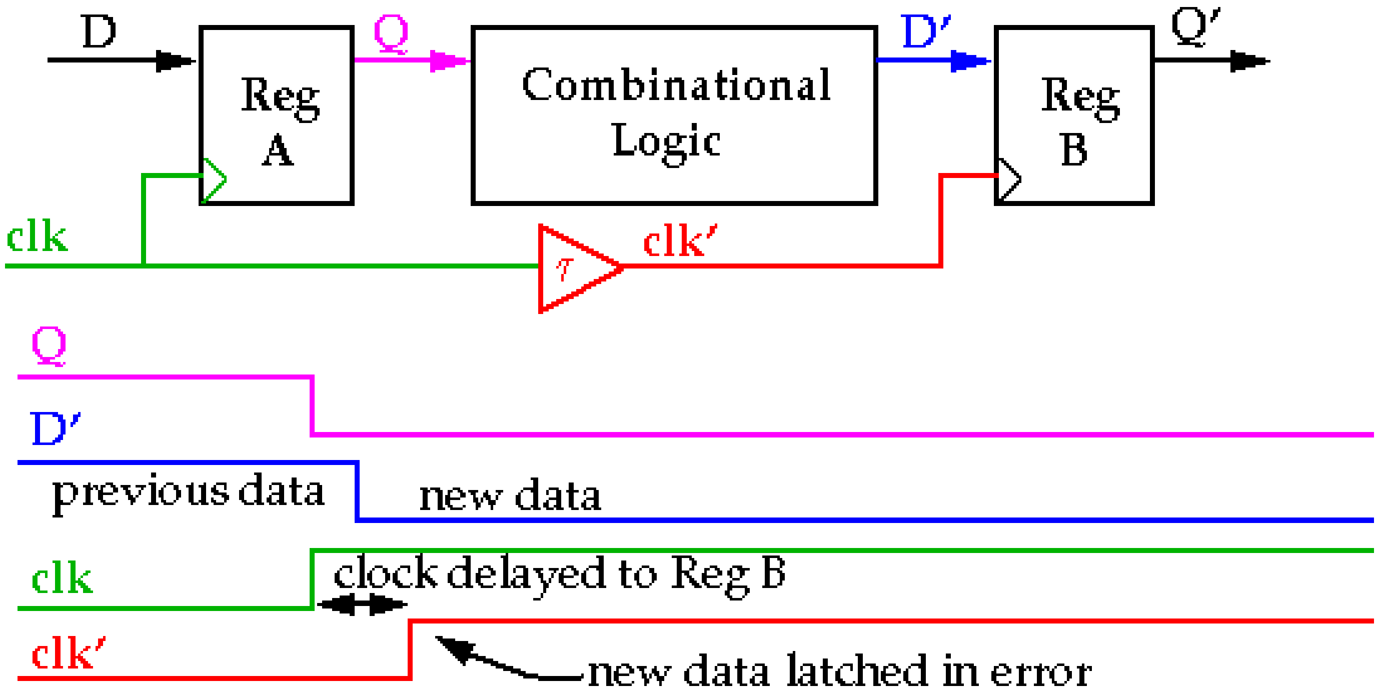
Clock Skew's Effect on Cycle Time



- How to take care of Clock Skew?
- We shall assume there is not Clock Skew.

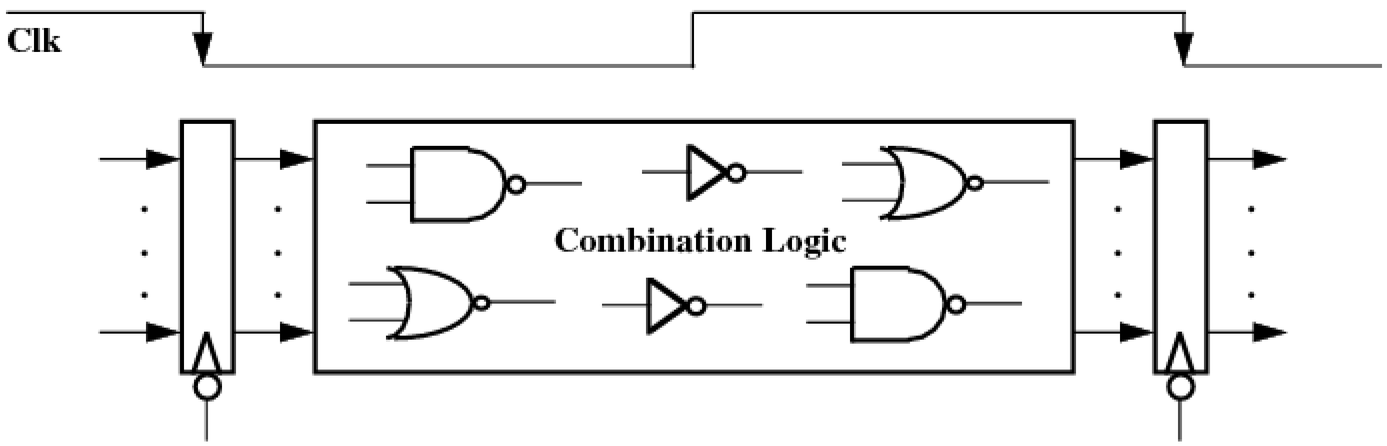


Clock Skew



Source: http://www.ece.unm.edu/~jimp/vlsi/slides/chap5_2.html

How to Avoid Hold Time Violation?



- Hold time requirement:
  - Input to register must NOT change immediately after the clock tick.
- CLK-to-Q + Shortest Delay Path must be greater than Hold Time

MIT 6.004 ISA Reference Card: Instructions

Instruction	Syntax	Description	Execution
LUI	**lui** rd, immU	Load Upper Immediate	reg[rd] <= immU << 12
JAL	**jal** rd, immJ	Jump and Link	reg[rd] <= pc + 4 pc <= pc + immJ
JALR	**jalr** rd, rs1, immI	Jump and Link Register	reg[rd] <= pc + 4 pc <= {(reg[rs1] + immI)[31:1], 1'b0}
BEQ	**beq** rs1, rs2, immB	Branch if $=$	pc <= (reg[rs1] == reg[rs2]) ? pc + immB : pc + 4
BNE	**bne** rs1, rs2, immB	Branch if $\neq$	pc <= (reg[rs1] != reg[rs2]) ? pc + immB : pc + 4
BLT	**blt** rs1, rs2, immB	Branch if $<$ (Signed)	pc <= (reg[rs1] $<_s$ reg[rs2]) ? pc + immB : pc + 4
BGE	**bge** rs1, rs2, immB	Branch if $\geq$ (Signed)	pc <= (reg[rs1] $>=_s$ reg[rs2]) ? pc + immB : pc + 4
BLTU	**bltu** rs1, rs2, immB	Branch if $<$ (Unsigned)	pc <= (reg[rs1] $<_u$ reg[rs2]) ? pc + immB : pc + 4
BGEU	**bgeu** rs1, rs2, immB	Branch if $\geq$ (Unsigned)	pc <= (reg[rs1] $>=_u$ reg[rs2]) ? pc + immB : pc + 4
LW	**lw** rd, immI(rs1)	Load Word	reg[rd] <= mem[reg[rs1] + immI]
SW	**sw** rs2, immS(rs1)	Store Word	mem[reg[rs1] + immS] <= reg[rs2]
ADDI	**addi** rd, rs1, immI	Add Immediate	reg[rd] <= reg[rs1] + immI
SLTI	**slti** rd, rs1, immI	Compare $<$ Immediate (Signed)	reg[rd] <= (reg[rs1] $<_s$ immI) ? 1 : 0
SLTIU	**sltiu** rd, rs1, immI	Compare $<$ Immediate (Unsigned)	reg[rd] <= (reg[rs1] $<_u$ immI) ? 1 : 0
XORI	**xori** rd, rs1, immI	Xor Immediate	reg[rd] <= reg[rs1] ^ immI
ORI	**ori** rd, rs1, immI	Or Immediate	reg[rd] <= reg[rs1] | immI
ANDI	**andi** rd, rs1, immI	And Immediate	reg[rd] <= reg[rs1] & immI
SLLI	**slli** rd, rs1, immI	Shift Left Logical Immediate	reg[rd] <= reg[rs1] << immI
SRLI	**srli** rd, rs1, immI	Shift Right Logical Immediate	reg[rd] <= reg[rs1] $>>_u$ immI
SRAI	**srai** rd, rs1, immI	Shift Right Arithmetic Immediate	reg[rd] <= reg[rs1] $>>_s$ immI
ADD	**add** rd, rs1, rs2	Add	reg[rd] <= reg[rs1] + reg[rs2]
SUB	**sub** rd, rs1, rs2	Subtract	reg[rd] <= reg[rs1] - reg[rs2]
SLL	**sll** rd, rs1, rs2	Shift Left Logical	reg[rd] <= reg[rs1] << reg[rs2]
SLT	**slt** rd, rs1, rs2	Compare $<$ (Signed)	reg[rd] <= (reg[rs1] $<_s$ reg[rs2]) ? 1 : 0
SLTU	**sltu** rd, rs1, rs2	Compare $<$ (Unsigned)	reg[rd] <= (reg[rs1] $<_u$ reg[rs2]) ? 1 : 0
XOR	**xor** rd, rs1, rs2	Xor	reg[rd] <= reg[rs1] ^ reg[rs2]
SRL	**srl** rd, rs1, rs2	Shift Right Logical	reg[rd] <= reg[rs1] $>>_u$ reg[rs2]
SRA	**sra** rd, rs1, rs2	Shift Right Arithmetic	reg[rd] <= reg[rs1] $>>_s$ reg[rs2]
OR	**or** rd, rs1, rs2	Or	reg[rd] <= reg[rs1] | reg[rs2]
AND	**and** rd, rs1, rs2	And	reg[rd] <= reg[rs1] & reg[rs2]

NOTE: All immediate values (*immU*, *immJ*, *immI*, *immB*, and *immS*) are sign-extended to 32-bits.

MIT 6.004 ISA Reference Card: Pseudoinstructions

Pseudoinstruction	Description	Execution
li rd, constant	Load Immediate	reg[rd] <= constant
mv rd, rs1	Move	reg[rd] <= reg[rs1] + 0
not rd, rs1	Logical Not	reg[rd] <= reg[rs1] ^ -1
neg rd, rs1	Arithmetic Negation	reg[rd] <= 0 - reg[rs1]
j label	Jump	pc <= label
jal label **call** label	Jump and Link (with ra)	reg[ra] <= pc + 4 pc <= label
jr rs	Jump Register	pc <= reg[rs1] & ~1
jalr rs	Jump and Link Register (with ra)	reg[ra] <= pc + 4 pc <= reg[rs1] & ~1
ret	Return from Subroutine	pc <= reg[ra]
bgt rs1, rs2, label	Branch $>$ (Signed)	pc <= (reg[rs1] $>_s$ reg[rs2]) ? label : pc + 4
ble rs1, rs2, label	Branch $\leq$ (Signed)	pc <= (reg[rs1] $<=_s$ reg[rs2]) ? label : pc + 4
bgtu rs1, rs2, label	Branch $>$ (Unsigned)	pc <= (reg[rs1] $>_s$ reg[rs2]) ? label : pc + 4
bleu rs1, rs2, label	Branch $\leq$ (Unsigned)	pc <= (reg[rs1] $<=_s$ reg[rs2]) ? label : pc + 4
beqz rs1, label	Branch $= 0$	pc <= (reg[rs1] == 0) ? label : pc + 4
bnez rs1, label	Branch $\neq 0$	pc <= (reg[rs1] != 0) ? label : pc + 4
bltz rs1, label	Branch < 0 (Signed)	pc <= (reg[rs1] $<_s$ 0) ? label : pc + 4
bgez rs1, label	Branch ≥ 0 (Signed)	pc <= (reg[rs1] $>=_s$ 0) ? label : pc + 4
bgtz rs1, label	Branch > 0 (Signed)	pc <= (reg[rs1] $>_s$ 0) ? label : pc + 4
blez rs1, label	Branch ≤ 0 (Signed)	pc <= (reg[rs1] $<=_s$ 0) ? label : pc + 4

MIT 6.004 ISA Reference Card: Calling Convention

Registers	Symbolic names	Description	Saver
`x0`	`zero`	Hardwired zero	—
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	—
`x4`	`tp`	Thread pointer	—
`x5-x7`	`t0-t2`	Temporary registers	Caller
`x8-x9`	`s0-s1`	Saved registers	Callee
`x10-x11`	`a0-a1`	Function arguments and return values	Caller
`x12-x17`	`a2-a7`	Function arguments	Caller
`x18-x27`	`s2-s11`	Saved registers	Callee
`x28-x31`	`t3-t6`	Temporary registers	Caller

MIT 6.004 ISA Reference Card: Instruction Encodings

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

RV32I Base Instruction Set (MIT 6.004 subset)

imm[31:12]				rd	0110111	LUI
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

RISC-V Reference

RISC-V Instruction Set

Core Instruction Formats

<table>
<tr><td>31 27 26 25</td><td>24 20</td><td>19 15</td><td>14 12</td><td>11 7</td><td>6 0</td><td></td></tr>
<tr><td>funct7</td><td>rs2</td><td>rs1</td><td>funct3</td><td>rd</td><td>opcode</td><td>R-type</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1</td><td>funct3</td><td>rd</td><td>opcode</td><td>I-type</td></tr>
<tr><td>imm[11:5]</td><td>rs2</td><td>rs1</td><td>funct3</td><td>imm[4:0]</td><td>opcode</td><td>S-type</td></tr>
<tr><td>imm[12|10:5]</td><td>rs2</td><td>rs1</td><td>funct3</td><td>imm[4:1|11]</td><td>opcode</td><td>B-type</td></tr>
<tr><td colspan="4">imm[31:12]</td><td>rd</td><td>opcode</td><td>U-type</td></tr>
<tr><td colspan="4">imm[20|10:1|11|19:12]</td><td>rd</td><td>opcode</td><td>J-type</td></tr>
</table>

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ˆ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ˆ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

Standard Extensions

RV32M Multiply Extension

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
mul	MUL	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
mulh	MUL High	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
mulsu	MUL High (S) (U)	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
mulu	MUL High (U)	R	0110011	0x3	0x01	rd = (rs1 * rs2)[63:32]
div	DIV	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	DIV (U)	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	Remainder (U)	R	0110011	0x7	0x01	rd = rs1 % rs2

RV32A Atomic Extension

31–27	26	25	24–20	19–15	14–12	11–7	6–0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7

Inst	Name	FMT	Opcode	funct3	funct5	Description (C)
lr.w	Load Reserved	R	0101111	0x2	0x02	rd = M[rs1], reserve M[rs1]
sc.w	Store Conditional	R	0101111	0x2	0x03	if (reserved) { M[rs1] = rs2; rd = 0 } else { rd = 1 }
amoswap.w	Atomic Swap	R	0101111	0x2	0x01	rd = M[rs1]; swap(rd, rs2); M[rs1] = rd
amoadd.w	Atomic ADD	R	0101111	0x2	0x00	rd = M[rs1] + rs2; M[rs1] = rd
amoand.w	Atomic AND	R	0101111	0x2	0x0C	rd = M[rs1] & rs2; M[rs1] = rd
amoor.w	Atomic OR	R	0101111	0x2	0x0A	rd = M[rs1] | rs2; M[rs1] = rd
amoxor.w	Atomix XOR	R	0101111	0x2	0x04	rd = M[rs1] ^ rs2; M[rs1] = rd
amomax.w	Atomic MAX	R	0101111	0x2	0x14	rd = max(M[rs1], rs2); M[rs1] = rd
amomin.w	Atomic MIN	R	0101111	0x2	0x10	rd = min(M[rs1], rs2); M[rs1] = rd

RV32F / D Floating-Point Extensions

Inst	Name	FMT	Opcode	funct3	funct5	Description (C)
flw	Flt Load Word	*				rd = M[rs1 + imm]
fsw	Flt Store Word	*				M[rs1 + imm] = rs2
fmadd.s	Flt Fused Mul-Add	*				rd = rs1 * rs2 + rs3
fmsub.s	Flt Fused Mul-Sub	*				rd = rs1 * rs2 - rs3
fnmadd.s	Flt Neg Fused Mul-Add	*				rd = -rs1 * rs2 + rs3
fnmsub.s	Flt Neg Fused Mul-Sub	*				rd = -rs1 * rs2 - rs3
fadd.s	Flt Add	*				rd = rs1 + rs2
fsub.s	Flt Sub	*				rd = rs1 - rs2
fmul.s	Flt Mul	*				rd = rs1 * rs2
fdiv.s	Flt Div	*				rd = rs1 / rs2
fsqrt.s	Flt Square Root	*				rd = sqrt(rs1)
fsgnj.s	Flt Sign Injection	*				rd = abs(rs1) * sgn(rs2)
fsgnjn.s	Flt Sign Neg Injection	*				rd = abs(rs1) * -sgn(rs2)
fsgnjx.s	Flt Sign Xor Injection	*				rd = rs1 * sgn(rs2)
fmin.s	Flt Minimum	*				rd = min(rs1, rs2)
fmax.s	Flt Maximum	*				rd = max(rs1, rs2)
fcvt.s.w	Flt Conv from Sign Int	*				rd = (float) rs1
fcvt.s.wu	Flt Conv from Uns Int	*				rd = (float) rs1
fcvt.w.s	Flt Convert to Int	*				rd = (int32_t) rs1
fcvt.wu.s	Flt Convert to Int	*				rd = (uint32_t) rs1
fmv.x.w	Move Float to Int	*				rd = *((int*) &rs1)
fmv.w.x	Move Int to Float	*				rd = *((float*) &rs1)
feq.s	Float Equality	*				rd = (rs1 == rs2) ? 1 : 0
flt.s	Float Less Than	*				rd = (rs1 < rs2) ? 1 : 0
fle.s	Float Less / Equal	*				rd = (rs1 <= rs2) ? 1 : 0
fclass.s	Float Classify	*				rd = 0..9

RV32C Compressed Extension

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td></td></tr>
<tr><td colspan="4">funct4</td><td colspan="5">rd/rs1</td><td colspan="5">rs2</td><td colspan="2">op</td><td>CR-type</td></tr>
<tr><td colspan="3">funct3</td><td>imm</td><td colspan="5">rd/rs1</td><td colspan="5">imm</td><td colspan="2">op</td><td>CI-type</td></tr>
<tr><td colspan="3">funct3</td><td colspan="6">imm</td><td colspan="5">rs2</td><td colspan="2">op</td><td>CSS-type</td></tr>
<tr><td colspan="3">funct3</td><td colspan="8">imm</td><td colspan="3">rd’</td><td colspan="2">op</td><td>CIW-type</td></tr>
<tr><td colspan="3">funct3</td><td colspan="3">imm</td><td colspan="3">rs1’</td><td colspan="2">imm</td><td colspan="3">rd’</td><td colspan="2">op</td><td>CL-type</td></tr>
<tr><td colspan="3">funct3</td><td colspan="3">imm</td><td colspan="3">rd’/rs1’</td><td colspan="2">imm</td><td colspan="3">rs2’</td><td colspan="2">op</td><td>CS-type</td></tr>
<tr><td colspan="3">funct3</td><td colspan="3">imm</td><td colspan="3">rs1’</td><td colspan="5">imm</td><td colspan="2">op</td><td>CB-type</td></tr>
<tr><td colspan="3">funct3</td><td colspan="11">offset</td><td colspan="2">op</td><td>CJ-type</td></tr>
</table>

Inst	Name	FMT	OP	Funct	Description
`c.lwsp`	Load Word from SP	CI	`10`	`010`	`lw rd, (4*imm)(sp)`
`c.swsp`	Store Word to SP	CSS	`10`	`110`	`sw rs2, (4*imm)(sp)`
`c.lw`	Load Word	CL	`00`	`010`	`lw rd’, (4*imm)(rs1’)`
`c.sw`	Store Word	CS	`00`	`110`	`sw rs1’, (4*imm)(rs2’)`
`c.j`	Jump	CJ	`01`	`101`	`jal x0, 2*offset`
`c.jal`	Jump And Link	CJ	`01`	`001`	`jal ra, 2*offset`
`c.jr`	Jump Reg	CR	`10`	`1000`	`jalr x0, rs1, 0`
`c.jalr`	Jump And Link Reg	CR	`10`	`1001`	`jalr ra, rs1, 0`
`c.beqz`	Branch == 0	CB	`01`	`110`	`beq rs’, x0, 2*imm`
`c.bnez`	Branch != 0	CB	`01`	`111`	`bne rs’, x0, 2*imm`
`c.li`	Load Immediate	CI	`01`	`010`	`addi rd, x0, imm`
`c.lui`	Load Upper Imm	CI	`01`	`011`	`lui rd, imm`
`c.addi`	ADD Immediate	CI	`01`	`000`	`addi rd, rd, imm`
`c.addi16sp`	ADD Imm * 16 to SP	CI	`01`	`011`	`addi sp, sp, 16*imm`
`c.addi4spn`	ADD Imm * 4 + SP	CIW	`00`	`000`	`addi rd’, sp, 4*imm`
`c.slli`	Shift Left Logical Imm	CI	`10`	`000`	`slli rd, rd, imm`
`c.srli`	Shift Right Logical Imm	CB	`01`	`100x00`	`srli rd’, rd’, imm`
`c.srai`	Shift Right Arith Imm	CB	`01`	`100x01`	`srai rd’, rd’, imm`
`c.andi`	AND Imm	CB	`01`	`100x10`	`andi rd’, rd’, imm`
`c.mv`	MoVe	CR	`10`	`1000`	`add rd, x0, rs2`
`c.add`	ADD	CR	`10`	`1001`	`add rd, rd, rs2`
`c.and`	AND	CS	`01`	`10001111`	`and rd’, rd’, rs2’`
`c.or`	OR	CS	`01`	`10001110`	`or rd’, rd’, rs2’`
`c.xor`	XOR	CS	`01`	`10001101`	`xor rd’, rd’, rs2’`
`c.sub`	SUB	CS	`01`	`10001100`	`sub rd’, rd’, rs2’`
`c.nop`	No OPeration	CI	`01`	`000`	`addi x0, x0, 0`
`c.ebreak`	Environment BREAK	CR	`10`	`1001`	`ebreak`

Pseudo Instructions

Pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if $=$ zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O

Registers

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Callee
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP args/return values	Caller
f12-17	fa2-7	FP args	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base		+RV{64,128}	
Loads	Load Byte	I	LB	rd,rs1,imm		
	Load Halfword	I	LH	rd,rs1,imm		
	Load Word	I	LW	rd,rs1,imm	L{D|Q}	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm		
	Load Half Unsigned	I	LHU	rd,rs1,imm	L{W|D}U	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm		
	Store Halfword	S	SH	rs1,rs2,imm		
	Store Word	S	SW	rs1,rs2,imm	S{D|Q}	rs1,rs2,imm
Shifts	Shift Left	R	SLL	rd,rs1,rs2	SLL{W|D}	rd,rs1,rs2
	Shift Left Immediate	I	SLLI	rd,rs1,shamt	SLLI{W|D}	rd,rs1,shamt
	Shift Right	R	SRL	rd,rs1,rs2	SRL{W|D}	rd,rs1,rs2
	Shift Right Immediate	I	SRLI	rd,rs1,shamt	SRLI{W|D}	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2	SRA{W|D}	rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI	rd,rs1,shamt	SRAI{W|D}	rd,rs1,shamt
Arithmetic	ADD	R	ADD	rd,rs1,rs2	ADD{W|D}	rd,rs1,rs2
	ADD Immediate	I	ADDI	rd,rs1,imm	ADDI{W|D}	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2	SUB{W|D}	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm		
	Add Upper Imm to PC	U	AUIPC	rd,imm		
Logical	XOR	R	XOR	rd,rs1,rs2		
	XOR Immediate	I	XORI	rd,rs1,imm		
	OR	R	OR	rd,rs1,rs2		
	OR Immediate	I	ORI	rd,rs1,imm		
	AND	R	AND	rd,rs1,rs2		
	AND Immediate	I	ANDI	rd,rs1,imm		
Compare	Set <	R	SLT	rd,rs1,rs2		
	Set < Immediate	I	SLTI	rd,rs1,imm		
	Set < Unsigned	R	SLTU	rd,rs1,rs2		
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm		
Branches	Branch =	SB	BEQ	rs1,rs2,imm		
	Branch ≠	SB	BNE	rs1,rs2,imm		
	Branch <	SB	BLT	rs1,rs2,imm		
	Branch ≥	SB	BGE	rs1,rs2,imm		
	Branch < Unsigned	SB	BLTU	rs1,rs2,imm		
	Branch ≥ Unsigned	SB	BGEU	rs1,rs2,imm		
Jump & Link	J&L	UJ	JAL	rd,imm		
	Jump & Link Register	UJ	JALR	rd,rs1,imm		
Synch	Synch thread	I	FENCE			
	Synch Instr & Data	I	FENCE.I			
System	System CALL	I	SCALL			
	System BREAK	I	SBREAK			
Counters	ReaD CYCLE	I	RDCYCLE	rd		
	ReaD CYCLE upper Half	I	RDCYCLEH	rd		
	ReaD TIME	I	RDTIME	rd		
	ReaD TIME upper Half	I	RDTIMEH	rd		
	ReaD INSTR RETired	I	RDINSTRET	rd		
	ReaD INSTR upper Half	I	RDINSTRETH	rd		

RV Privileged Instructions

Category	Name	RV mnemonic	
CSR Access	Atomic R/W	CSRRW	rd,csr,rs1
	Atomic Read & Set Bit	CSRRS	rd,csr,rs1
	Atomic Read & Clear Bit	CSRRC	rd,csr,rs1
	Atomic R/W Imm	CSRRWI	rd,csr,imm
	Atomic Read & Set Bit Imm	CSRRSI	rd,csr,imm
	Atomic Read & Clear Bit Imm	CSRRCI	rd,csr,imm
Change Level	Env. Call	ECALL	
	Environment Breakpoint	EBREAK	
	Environment Return	ERET	
Trap Redirect	to Supervisor	MRTS	
	Redirect Trap to Hypervisor	MRTH	
	Hypervisor Trap to Supervisor	HRTS	
Interrupt	Wait for Interrupt	WFI	
MMU	Supervisor FENCE	SFENCE.VM	rs1

Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC		RVI equivalent	
Loads	Load Word	CL	C.LW	rd′,rs1′,imm	LW	rd′,rs1′,imm*4
	Load Word SP	CI	C.LWSP	rd,imm	LW	rd,sp,imm*4
	Load Double	CL	C.LD	rd′,rs1′,imm	LD	rd′,rs1′,imm*8
	Load Double SP	CI	C.LDSP	rd,imm	LD	rd,sp,imm*8
	Load Quad	CL	C.LQ	rd′,rs1′,imm	LQ	rd′,rs1′,imm*16
	Load Quad SP	CI	C.LQSP	rd,imm	LQ	rd,sp,imm*16
Stores	Store Word	CS	C.SW	rs1′,rs2′,imm	SW	rs1′,rs2′,imm*4
	Store Word SP	CSS	C.SWSP	rs2,imm	SW	rs2,sp,imm*4
	Store Double	CS	C.SD	rs1′,rs2′,imm	SD	rs1′,rs2′,imm*8
	Store Double SP	CSS	C.SDSP	rs2,imm	SD	rs2,sp,imm*8
	Store Quad	CS	C.SQ	rs1′,rs2′,imm	SQ	rs1′,rs2′,imm*16
	Store Quad SP	CSS	C.SQSP	rs2,imm	SQ	rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD	rd,rs1	ADD	rd,rd,rs1
	ADD Word	CR	C.ADDW	rd,rs1	ADDW	rd,rd,imm
	ADD Immediate	CI	C.ADDI	rd,imm	ADDI	rd,rd,imm
	ADD Word Imm	CI	C.ADDIW	rd,imm	ADDIW	rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP	x0,imm	ADDI	sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN	rd',imm	ADDI	rd',sp,imm*4
	Load Immediate	CI	C.LI	rd,imm	ADDI	rd,x0,imm
	Load Upper Imm	CI	C.LUI	rd,imm	LUI	rd,imm
	MoVe	CR	C.MV	rd,rs1	ADD	rd,rs1,x0
	SUB	CR	C.SUB	rd,rs1	SUB	rd,rd,rs1
Shifts	Shift Left Imm	CI	C.SLLI	rd,imm	SLLI	rd,rd,imm
Branches	Branch=0	CB	C.BEQZ	rs1′,imm	BEQ	rs1',x0,imm
	Branch≠0	CB	C.BNEZ	rs1′,imm	BNE	rs1',x0,imm
Jump	Jump	CJ	C.J	imm	JAL	x0,imm
	Jump Register	CR	C.JR	rd,rs1	JALR	x0,rs1,0
Jump & Link	J&L	CJ	C.JAL	imm	JAL	ra,imm
	Jump & Link Register	CR	C.JALR	rs1	JALR	ra,rs1,0
System	Env. BREAK	CI	C.EBREAK		EBREAK	

32-bit Instruction Formats

	31	30–25	24–21	20	19–15	14–12	11–8	7	6–0
R	funct7		rs2		rs1	funct3	rd		opcode
I	imm[11:0]				rs1	funct3	rd		opcode
S	imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode
SB	imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode
U	imm[31:12]						rd		opcode
UJ	imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode

16-bit (RVC) Instruction Formats

	15 14 13 12	11 10	9 8 7	6 5	4 3 2	1 0
CR	funct4	rd/rs1		rs2		op
CI	funct3 / imm	rd/rs1		imm		op
CSS	funct3	imm		rs2		op
CIW	funct3	imm			rd′	op
CL	funct3	imm	rs1′	imm	rd′	op
CS	funct3	imm	rs1′	imm	rs2′	op
CB	funct3	offset	rs1′	offset		op
CJ	funct3	jump target				op

RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

Free & Open RISC-V Reference Card (riscv.org) ②

Optional Multiply-Divide Instruction Extension: RVM				
Category / Name	Fmt	RV32M (Multiply-Divide)		+RV{64,128}
Multiply MULtiply	R	MUL rd,rs1,rs2		MUL{W|D} rd,rs1,rs2
MULtiply upper Half	R	MULH rd,rs1,rs2		
MULtiply Half Sign/Uns	R	MULHSU rd,rs1,rs2		
MULtiply upper Half Uns	R	MULHU rd,rs1,rs2		
Divide DIVide	R	DIV rd,rs1,rs2		DIV{W|D} rd,rs1,rs2
DIVide Unsigned	R	DIVU rd,rs1,rs2		
Remainder REMainder	R	REM rd,rs1,rs2		REM{W|D} rd,rs1,rs2
REMainder Unsigned	R	REMU rd,rs1,rs2		REMU{W|D} rd,rs1,rs2

Optional Atomic Instruction Extension: RVA			
Category / Name	Fmt	RV32A (Atomic)	+RV{64,128}
Load Load Reserved	R	LR.W rd,rs1	LR.{D|Q} rd,rs1
Store Store Conditional	R	SC.W rd,rs1,rs2	SC.{D|Q} rd,rs1,rs2
Swap SWAP	R	AMOSWAP.W rd,rs1,rs2	AMOSWAP.{D|Q} rd,rs1,rs2
Add ADD	R	AMOADD.W rd,rs1,rs2	AMOADD.{D|Q} rd,rs1,rs2
Logical XOR	R	AMOXOR.W rd,rs1,rs2	AMOXOR.{D|Q} rd,rs1,rs2
AND	R	AMOAND.W rd,rs1,rs2	AMOAND.{D|Q} rd,rs1,rs2
OR	R	AMOOR.W rd,rs1,rs2	AMOOR.{D|Q} rd,rs1,rs2
Min/Max MINimum	R	AMOMIN.W rd,rs1,rs2	AMOMIN.{D|Q} rd,rs1,rs2
MAXimum	R	AMOMAX.W rd,rs1,rs2	AMOMAX.{D|Q} rd,rs1,rs2
MINimum Unsigned	R	AMOMINU.W rd,rs1,rs2	AMOMINU.{D|Q} rd,rs1,rs2
MAXimum Unsigned	R	AMOMAXU.W rd,rs1,rs2	AMOMAXU.{D|Q} rd,rs1,rs2

Three Optional Floating-Point Instruction Extensions: RVF, RVD, & RVQ			
Category / Name	Fmt	RV32{F|D|Q} (HP/SP,DP,QP Fl Pt)	+RV{64,128}
Move Move from Integer	R	FMV.{H|S}.X rd,rs1	FMV.{D|Q}.X rd,rs1
Move to Integer	R	FMV.X.{H|S} rd,rs1	FMV.X.{D|Q} rd,rs1
Convert Convert from Int	R	FCVT.{H|S|D|Q}.W rd,rs1	FCVT.{H|S|D|Q}.{L|T} rd,rs1
Convert from Int Unsigned	R	FCVT.{H|S|D|Q}.WU rd,rs1	FCVT.{H|S|D|Q}.{L|T}U rd,rs1
Convert to Int	R	FCVT.W.{H|S|D|Q} rd,rs1	FCVT.{L|T}.{H|S|D|Q} rd,rs1
Convert to Int Unsigned	R	FCVT.WU.{H|S|D|Q} rd,rs1	FCVT.{L|T}U.{H|S|D|Q} rd,rs1
Load Load	I	FL{W,D,Q} rd,rs1,imm	
Store Store	S	FS{W,D,Q} rs1,rs2,imm	
Arithmetic ADD	R	FADD.{S|D|Q} rd,rs1,rs2	
SUBtract	R	FSUB.{S|D|Q} rd,rs1,rs2	
MULtiply	R	FMUL.{S|D|Q} rd,rs1,rs2	
DIVide	R	FDIV.{S|D|Q} rd,rs1,rs2	
SQuare RooT	R	FSQRT.{S|D|Q} rd,rs1	
Mul-Add Multiply-ADD	R	FMADD.{S|D|Q} rd,rs1,rs2,rs3	
Multiply-SUBtract	R	FMSUB.{S|D|Q} rd,rs1,rs2,rs3	
Negative Multiply-SUBtract	R	FNMSUB.{S|D|Q} rd,rs1,rs2,rs3	
Negative Multiply-ADD	R	FNMADD.{S|D|Q} rd,rs1,rs2,rs3	
Sign Inject SiGN source	R	FSGNJ.{S|D|Q} rd,rs1,rs2	
Negative SiGN source	R	FSGNJN.{S|D|Q} rd,rs1,rs2	
Xor SiGN source	R	FSGNJX.{S|D|Q} rd,rs1,rs2	
Min/Max MINimum	R	FMIN.{S|D|Q} rd,rs1,rs2	
MAXimum	R	FMAX.{S|D|Q} rd,rs1,rs2	
Compare Compare Float =	R	FEQ.{S|D|Q} rd,rs1,rs2	
Compare Float <	R	FLT.{S|D|Q} rd,rs1,rs2	
Compare Float ≤	R	FLE.{S|D|Q} rd,rs1,rs2	
Categorization Classify Type	R	FCLASS.{S|D|Q} rd,rs1	
Configuration Read Status	R	FRCSR rd	
Read Rounding Mode	R	FRRM rd	
Read Flags	R	FRFLAGS rd	
Swap Status Reg	R	FSCSR rd,rs1	
Swap Rounding Mode	R	FSRM rd,rs1	
Swap Flags	R	FSFLAGS rd,rs1	
Swap Rounding Mode Imm	I	FSRMI rd,imm	
Swap Flags Imm	I	FSFLAGSI rd,imm	

RISC-V Calling Convention

Register	ABI Name	Saver	Description
x0	zero	---	Hard-wired zero
x1	ra	Caller	Return address
x2	sp	Callee	Stack pointer
x3	gp	---	Global pointer
x4	tp	---	Thread pointer
x5-7	t0-2	Caller	Temporaries
x8	s0/fp	Callee	Saved register/frame pointer
x9	s1	Callee	Saved register
x10-11	a0-1	Caller	Function arguments/return values
x12-17	a2-7	Caller	Function arguments
x18-27	s2-11	Callee	Saved registers
x28-31	t3-t6	Caller	Temporaries
f0-7	ft0-7	Caller	FP temporaries
f8-9	fs0-1	Callee	FP saved registers
f10-11	fa0-1	Caller	FP arguments/return values
f12-17	fa2-7	Caller	FP arguments
f18-27	fs2-11	Callee	FP saved registers
f28-31	ft8-11	Caller	FP temporaries

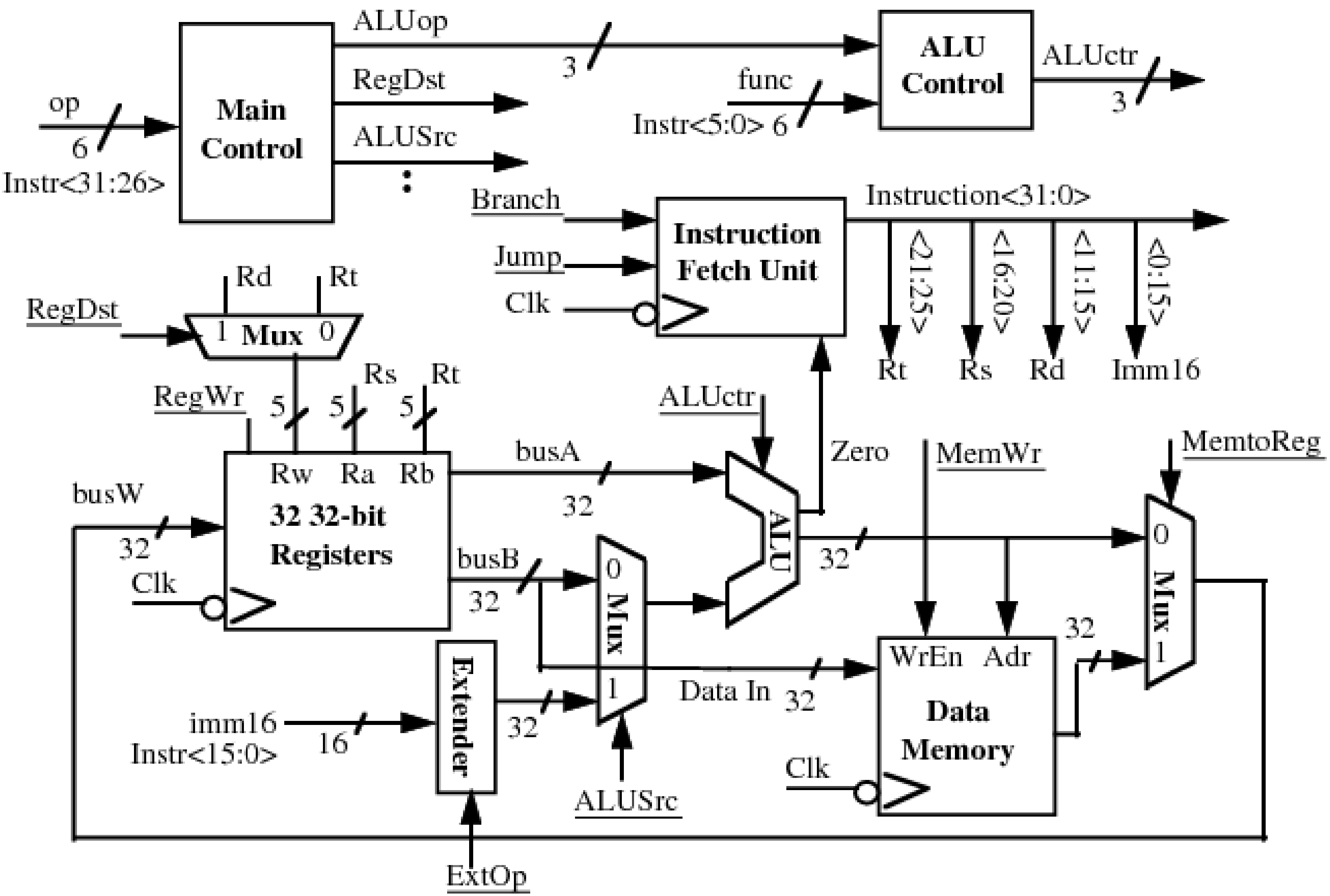
RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. Each larger address adds some instructions: 4 for RVM, 11 for RVA, and 6 each for RVF/D/Q. Using regex notation, `{}` *means set, so* `L{D|Q}` *is both* `LD` *and* `LQ`. *See risc.org. (8/21/15 revision)*

Acknowledgment: Almost all of these slides are based on Dave Patterson's CS152 Lecture Slides at UC, Berkeyley.

COMPUTER SYSTEMS ORGANIZATION

Multi Cycle CPU Design -- Spring 2012 -- IIIT-H -- Suresh Purini

A Single Cycle Processor



Drawbacks of Single Cycle Processor

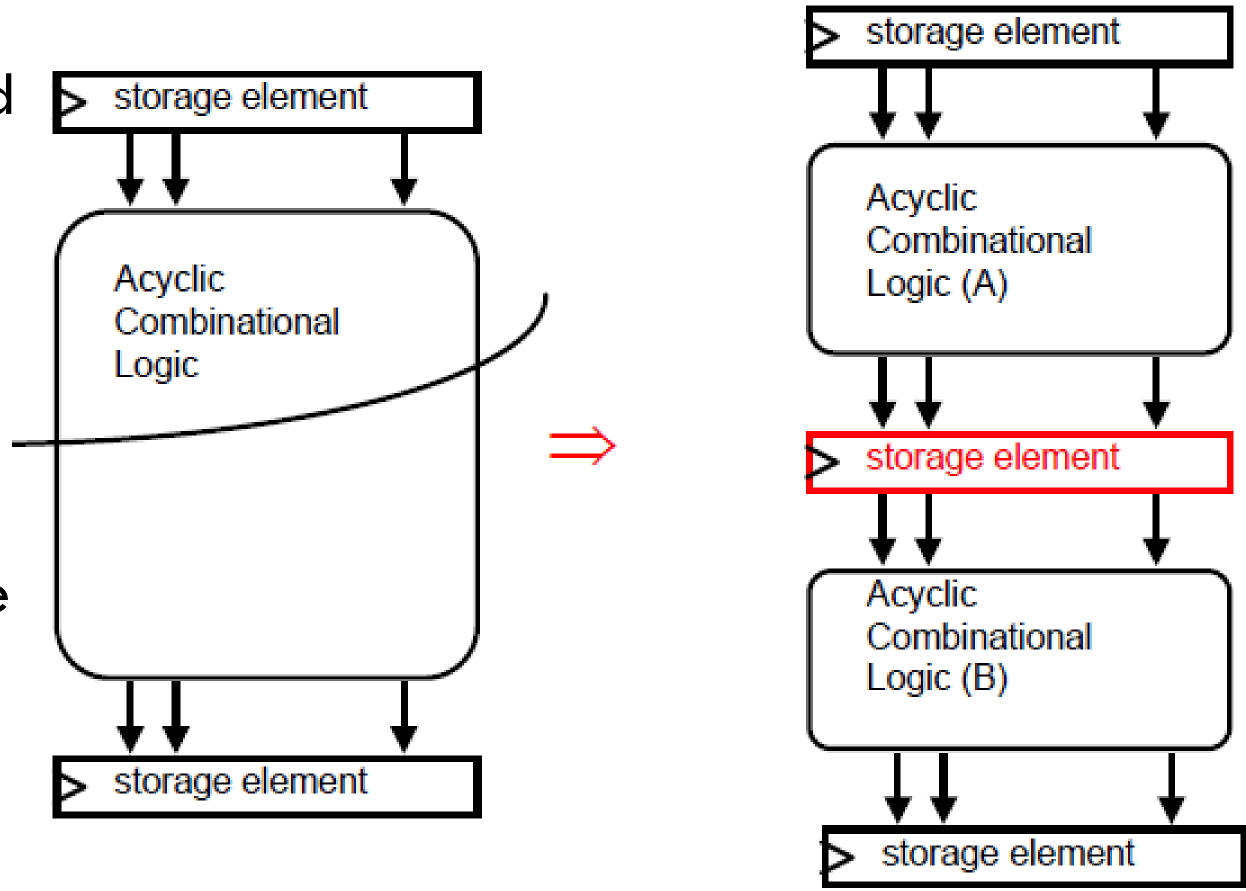
- Long cycle time: Cycle time must be long enough for the load instruction:
  - PC's Clock -to-Q +
  - Instruction Memory Access Time +
  - Register File Access Time +
  - ALU Delay (address calculation) +
  - Data Memory Access Time +
  - Register File Setup Time
- Cycle time is much longer than needed for all other instructions.
  - R-type instructions do not require data memory access
  - Jump does not require ALU operation nor data memory access

Overview of a Multiple Cycle Implementation

- The root of the single cycle processor's problems:
  - The cycle time has to be long enough for the slowest instruction
- Solution:
  - Break the instruction into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
    - Cycle time: time it takes to execute the longest step
    - Keep all the steps to have similar length
  - This is the essence of the multiple cycle processor

Reducing Cycle Time

- Cut combinational dependency graph and insert register / latch
- Do same work in two fast cycles, rather than one slow one
- May be able to short-circuit path and remove some components for some instructions!



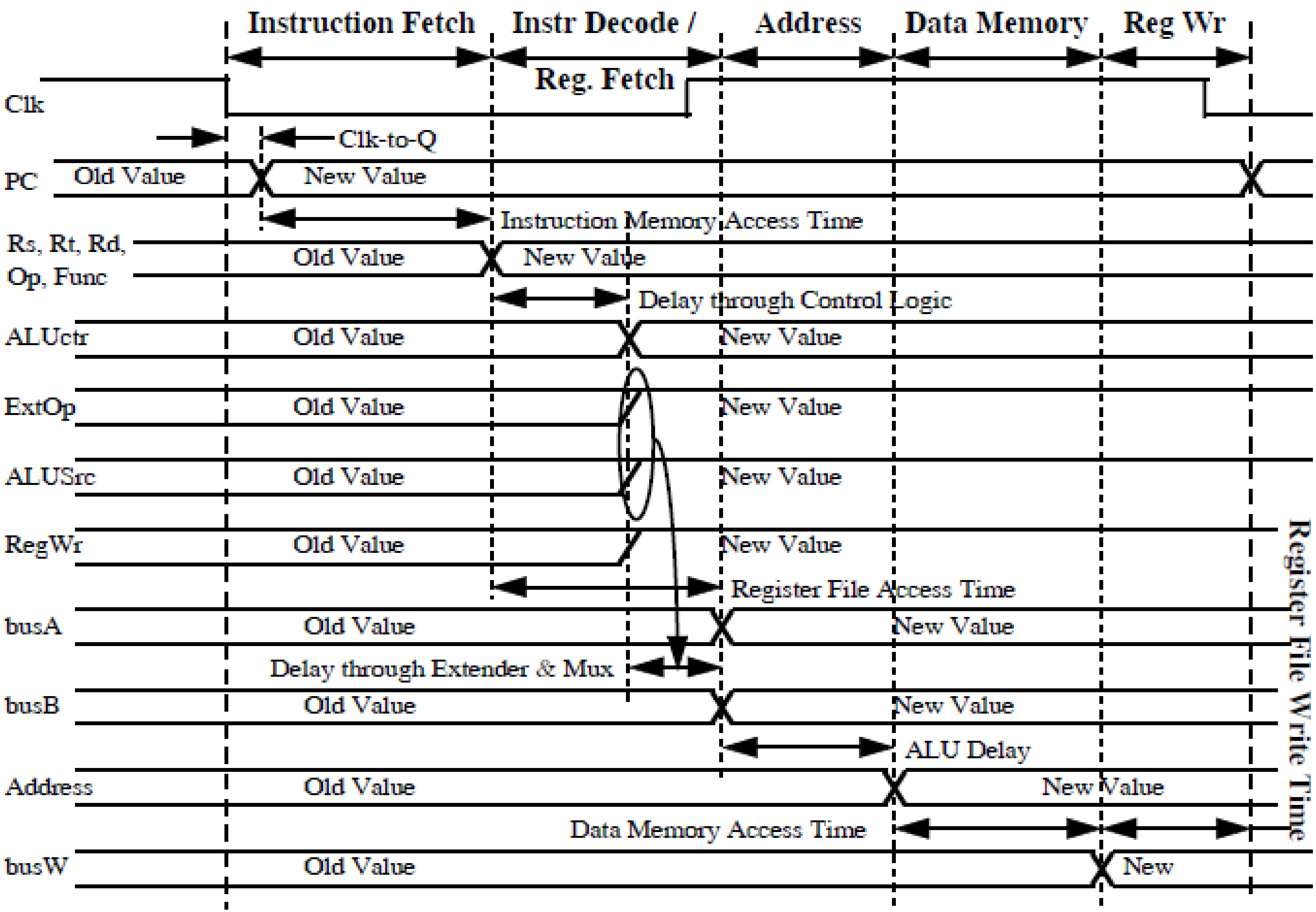
Advantages of Multiple Cycle Processor

- Cycle time is much shorter
- Different instructions take different number of cycles to complete
  - Load takes five cycles
  - Jump only takes three cycles
- Allows a functional unit to be used more than once per instruction

The Big Picture: Performance Perspective

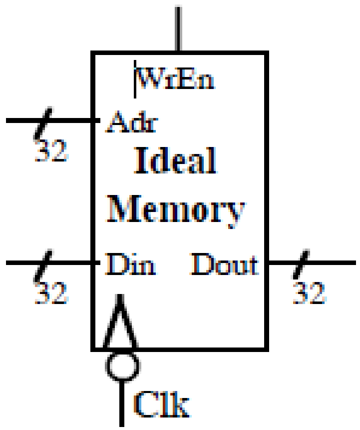
- Performance of a machine was determined by
  - Instruction Count
  - Clock cycle Time
  - Clock cycles per instruction
- Processor Design (data path and control) will determine
  - Clock cycle time
  - Clock cycles per instruction

The Five Steps of a Load Instruction

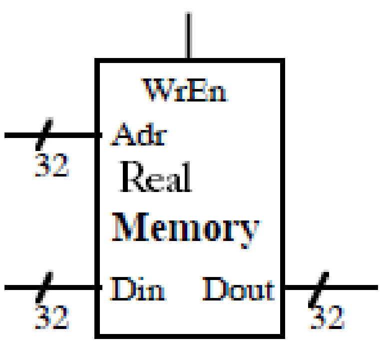


Memory Write Timing: Ideal Vs Reality

- In the Single Cycle Processor memory module is simplified
  - Write happens at the clock tick
  - Address, data, and write enable must be stable one "set-up" time before the clock tick



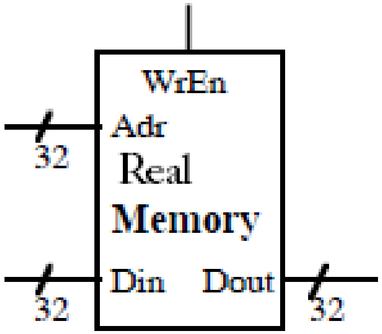
- In real life memory module has no clock input
  - The write path is a combinational logic delay path:
  - Write enable goes to 1 and Din settles down
  - Memory write access delay
  - Din is written into mem[address]



- Important: Address and Data must be stable BEFORE Write Enable goes to 1

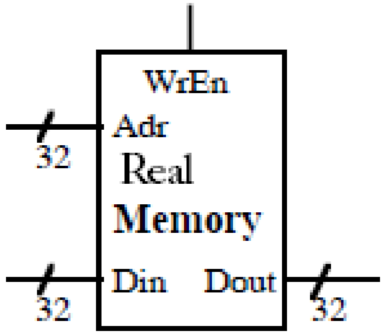
Race Condition Between Address and Write Enable

- The "real" (no clock input) memory may not work reliably in the single cycle processor because:
  - We cannot guarantee Address will be stable BEFORE WrEn = 1
  - There is a race between Adr and WrEn



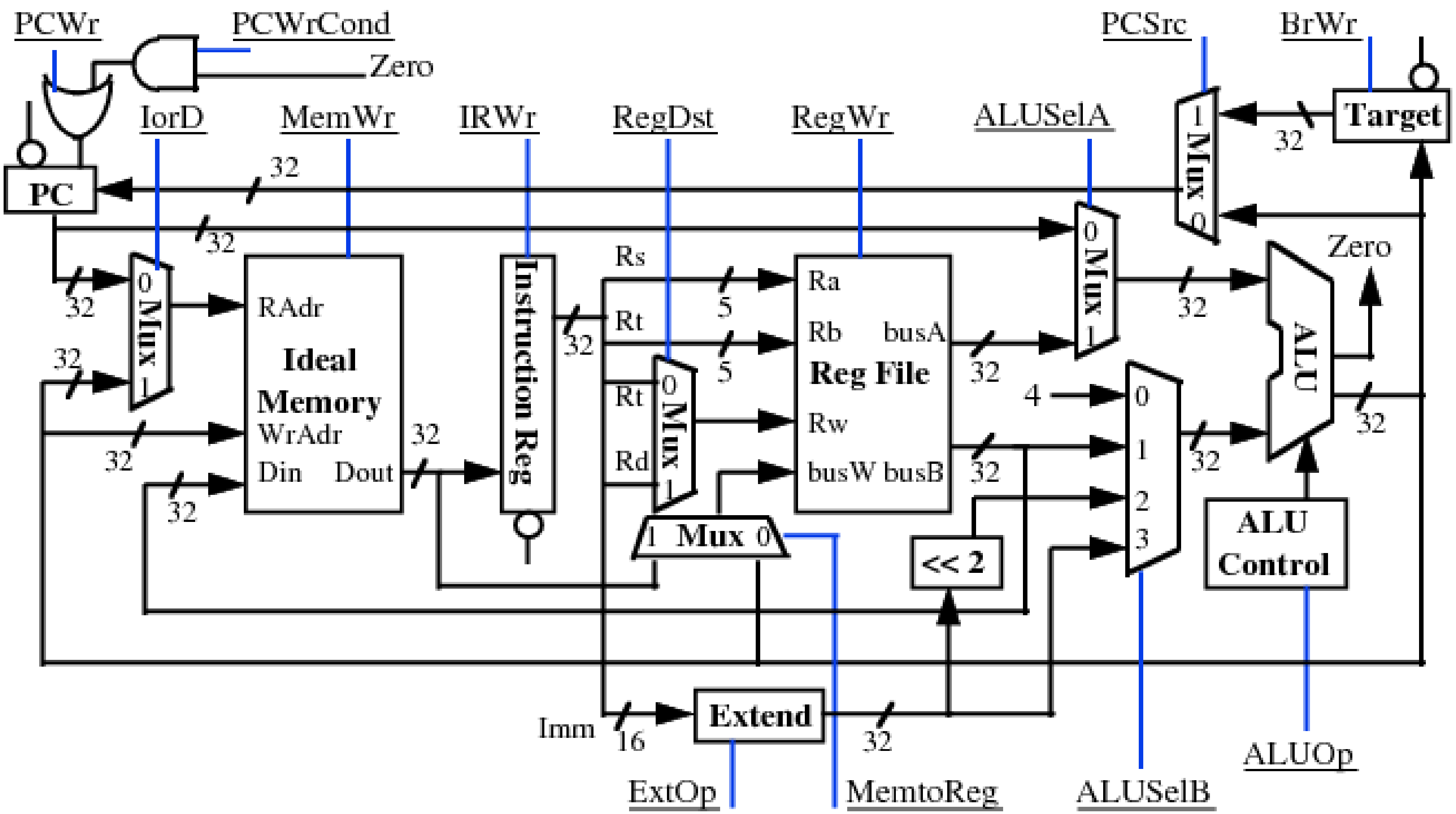
How to Avoid this Race Condition?

- Solution for the multiple cycle implementation:
  - Make sure Address is stable by the end of Cycle N
  - Assert Write Enable signal ONE cycle later at Cycle (N + 1)
  - Address cannot change until Write Enable is deasserted.

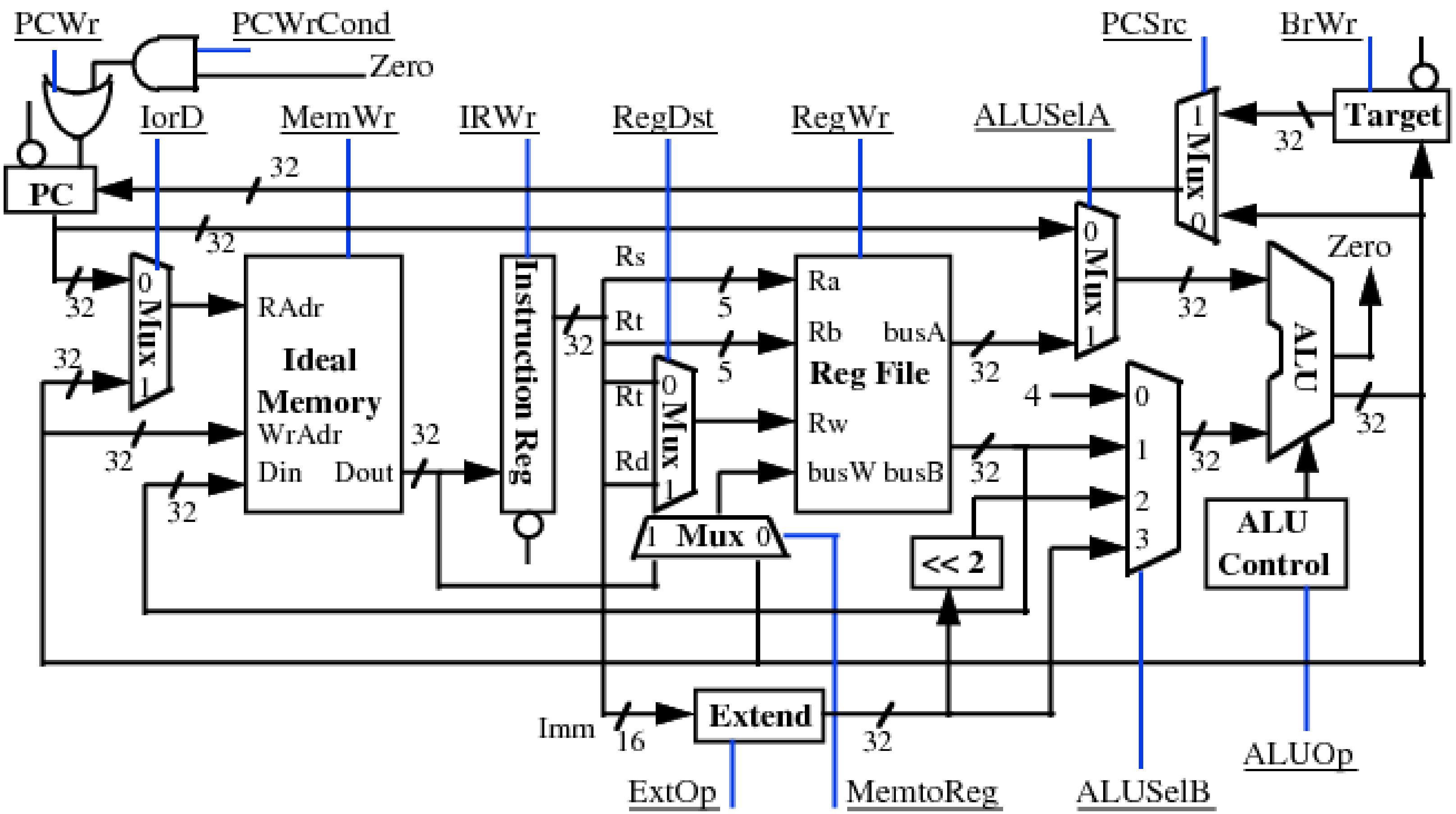


We can make use the same logic if we are using unclocked Register File to build our multiple cycle processor.

Multiple Cycle Data Path



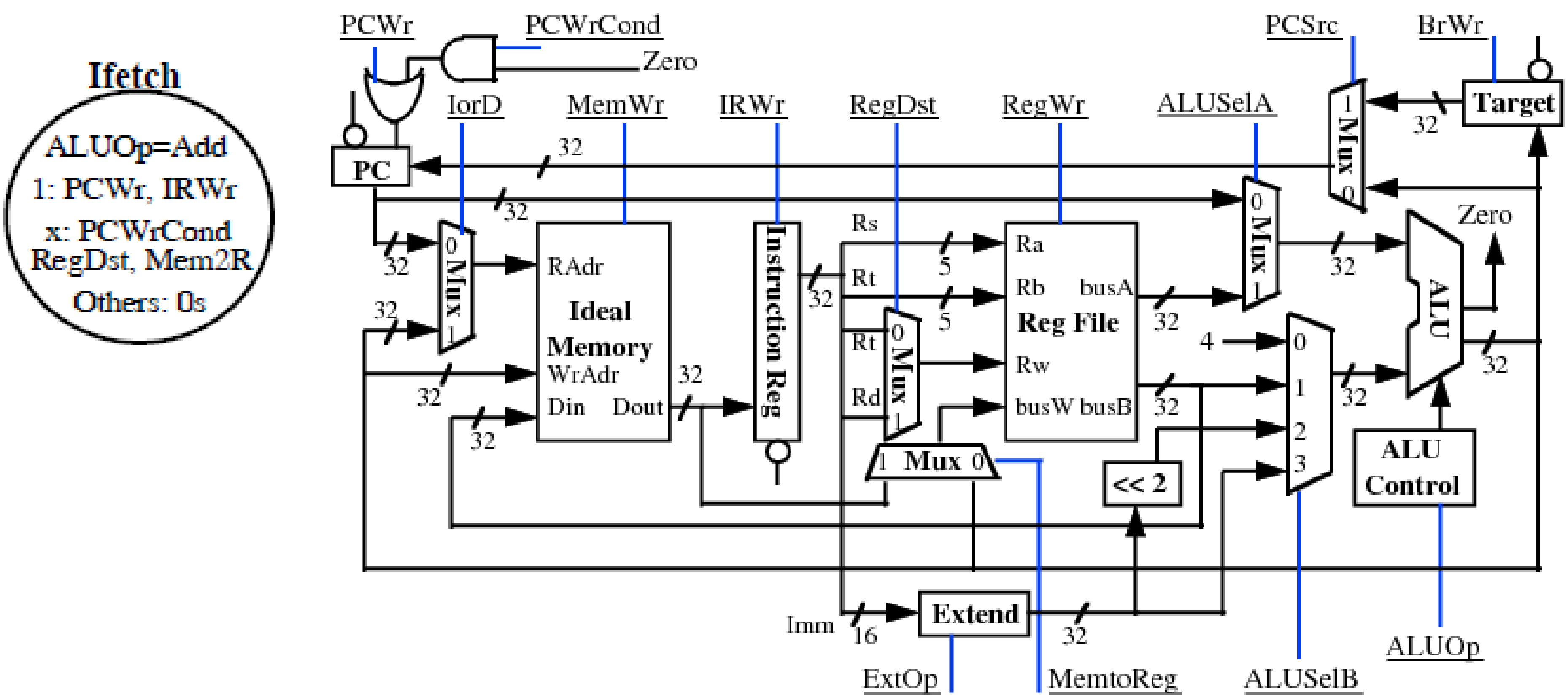
Multiple Cycle Data Path



Instruction Fetch Phase

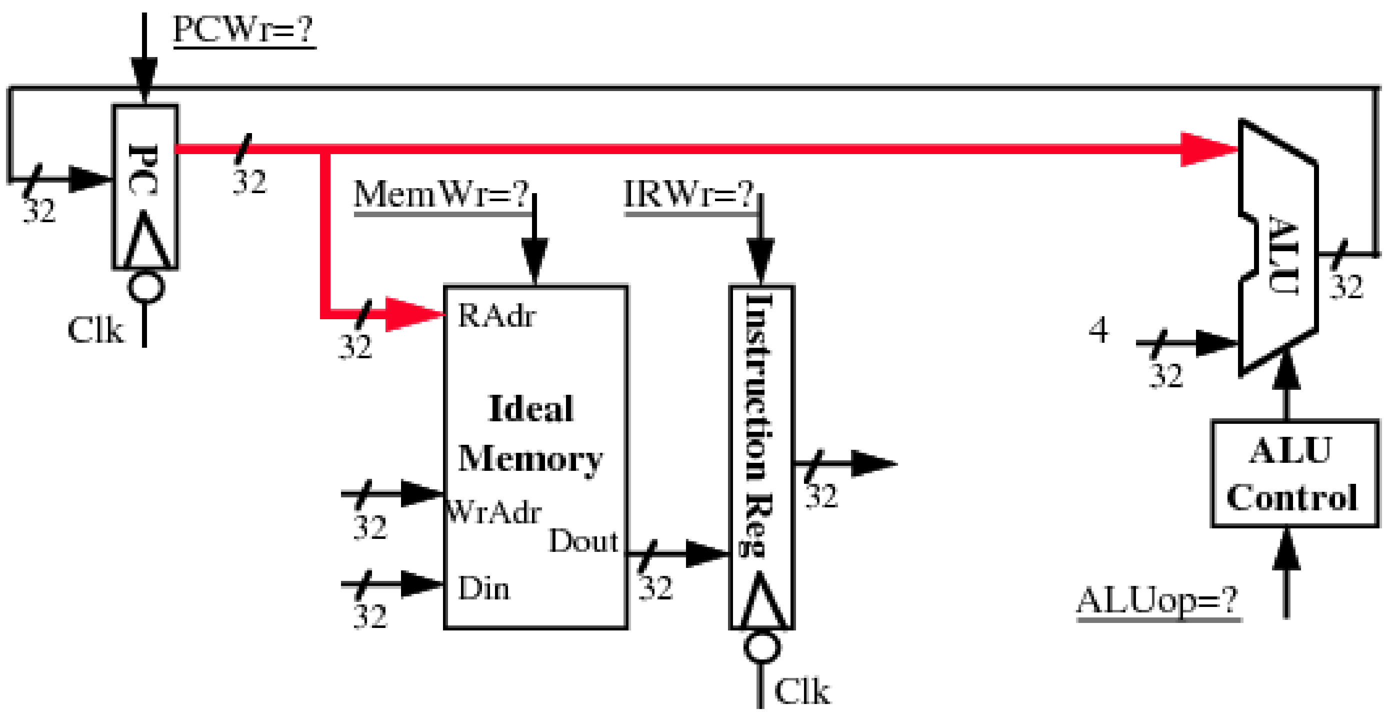
- IR = mem[PC]
- PC = PC + 4

PCWr	IorD	MemWr	IrWr	RegDst	RegWr	ALUSelA
1	0	0	1	x	0	0
PCWrCond	**PCSrc**	**BrWr**	**ExtOp**	**MemtoReg**	**ALUOp**	**ALUSelB**
x	0	0	x	x	ADD	0



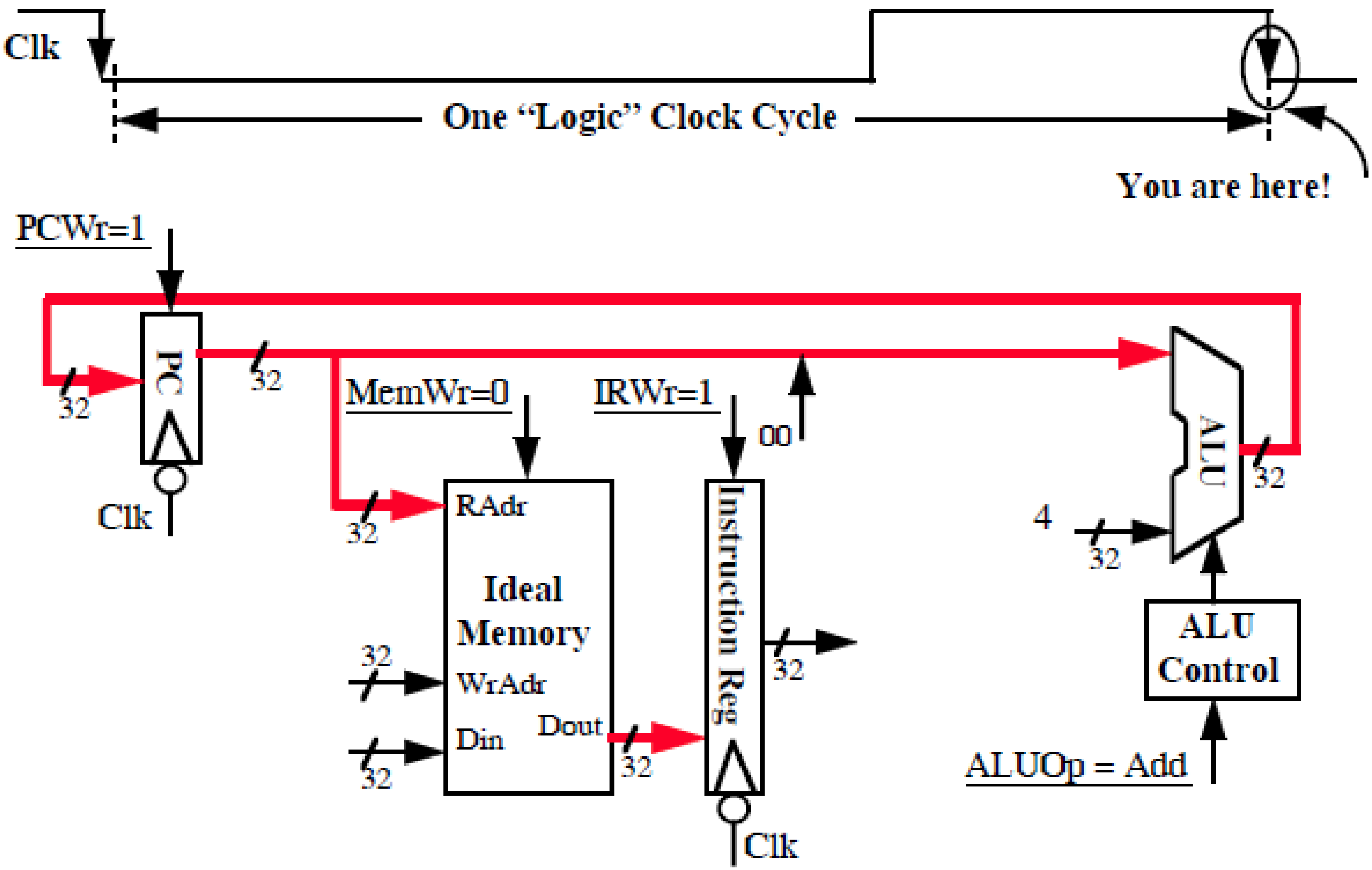
Instruction Fetch Cycle: In the Beginning

- Every cycle begins right AFTER the clock tick:
  - mem[PC]   PC<31:0> + 4

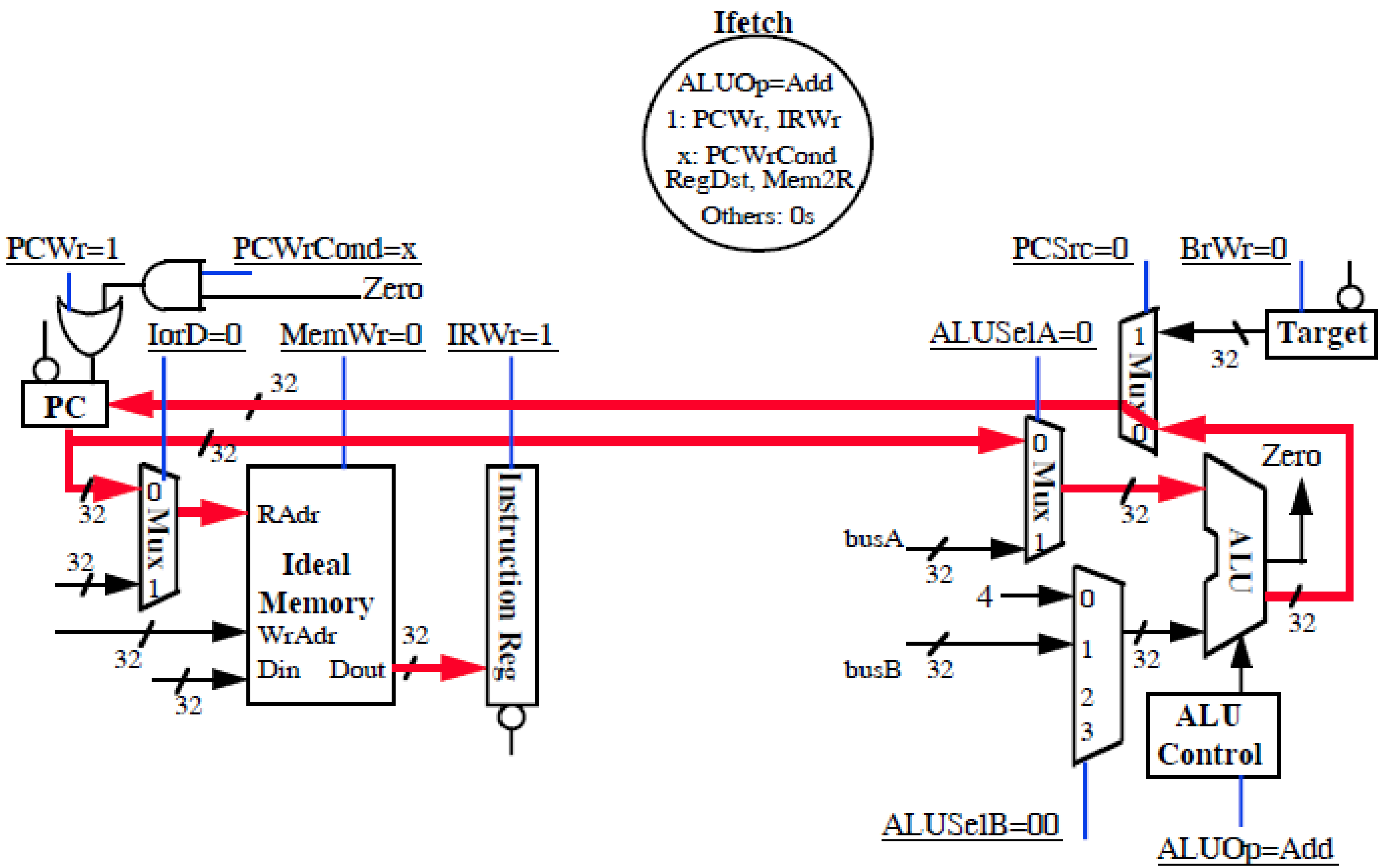


Instruction Fetch Cycle: The End

- Every cycle ends AT the next clock tick (storage element updates):
  - IR <-- mem[PC]   PC<31:0> <-- PC<31:0> + 4

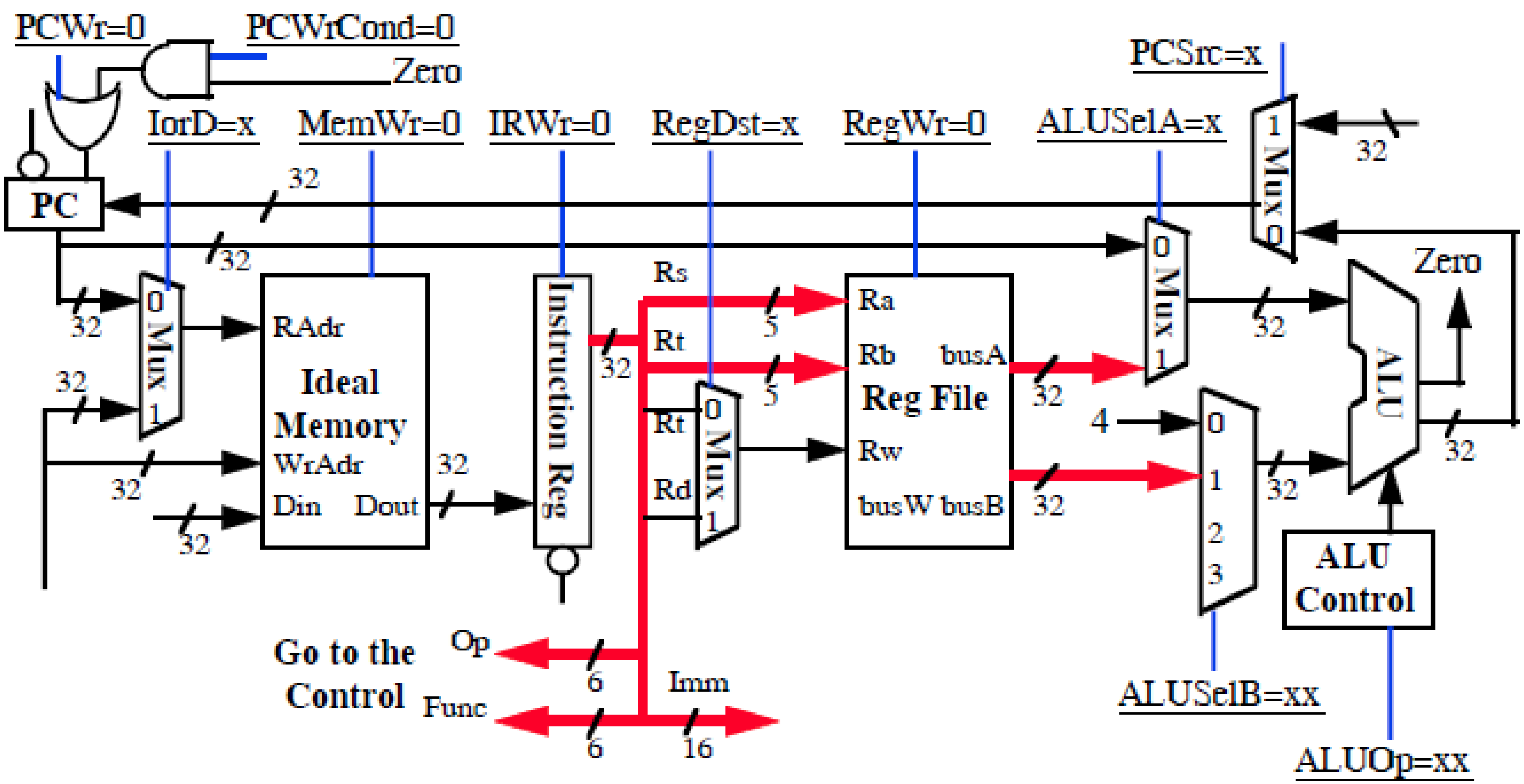


Instruction Fetch Cycle: Overall Picture



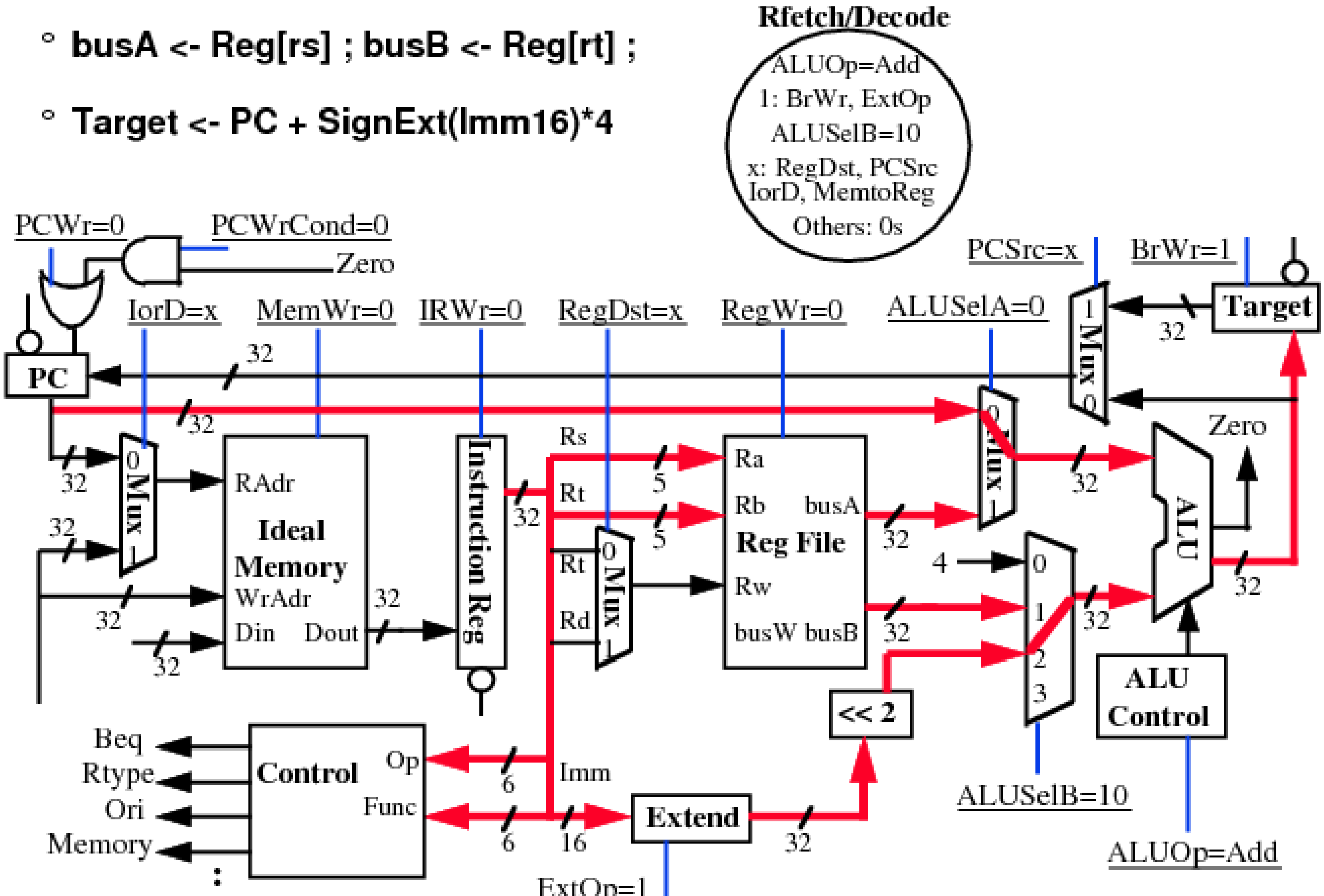
Register Fetch / Instruction Decode

- busA <- RegFile[rs] ; busB <- RegFile[rt] ;
- ALU is not being used: ALUctr = xx



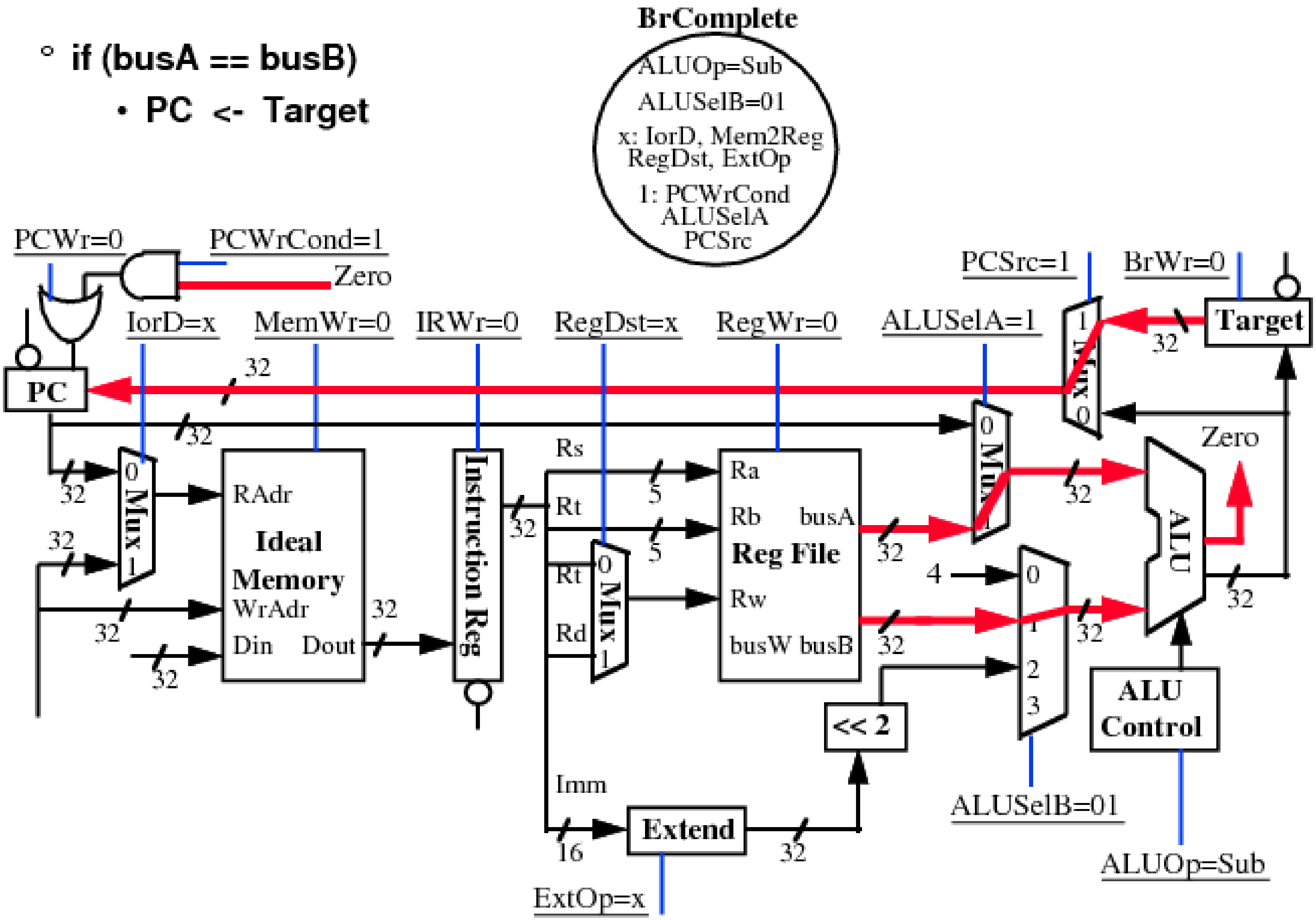
Register Fetch / Instruction Decode (Continue)

- busA <- Reg[rs] ; busB <- Reg[rt] ;
- Target <- PC + SignExt(Imm16)*4



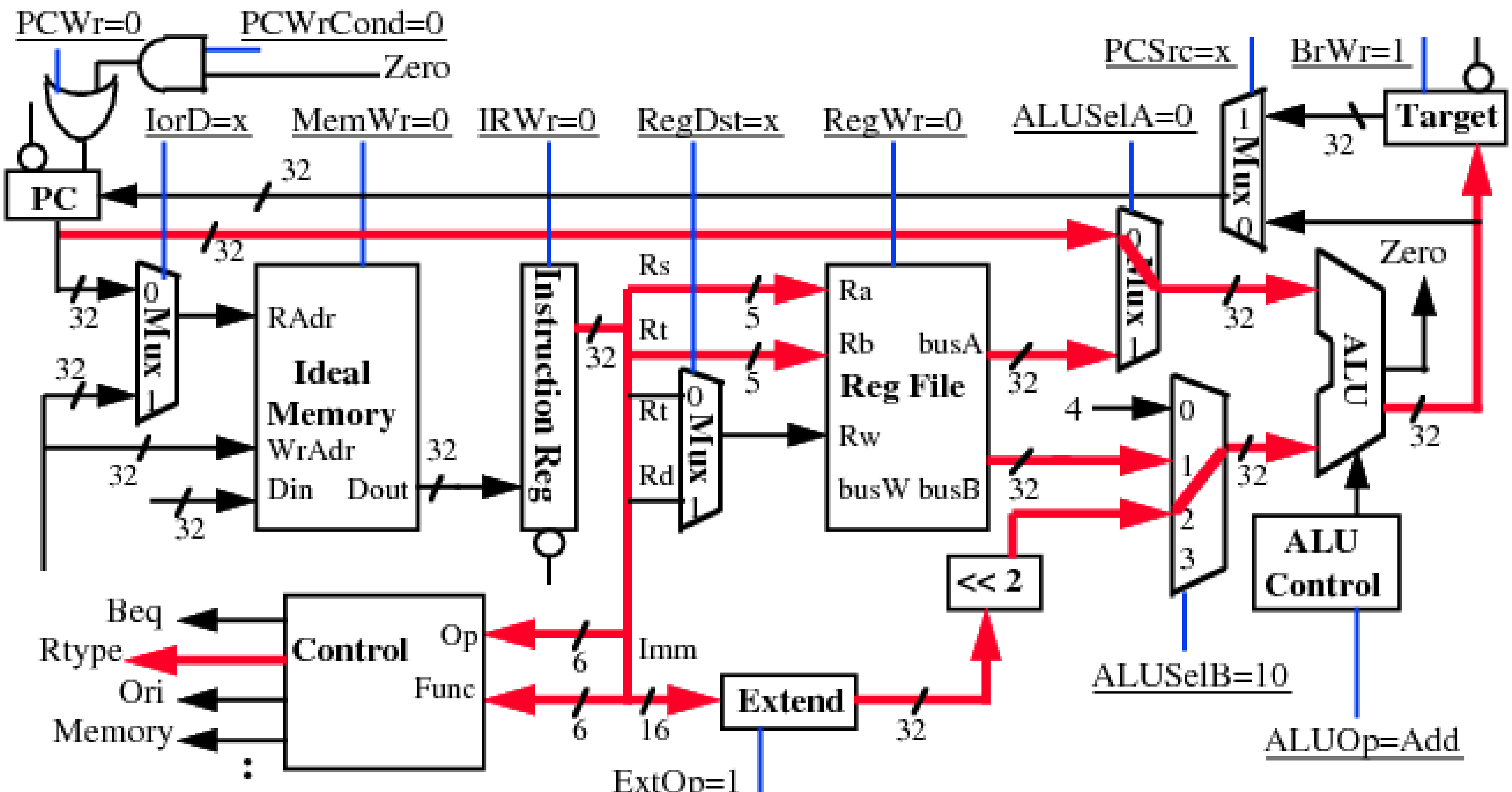
Branch Completion

- if (busA == busB)
  - PC <- Target



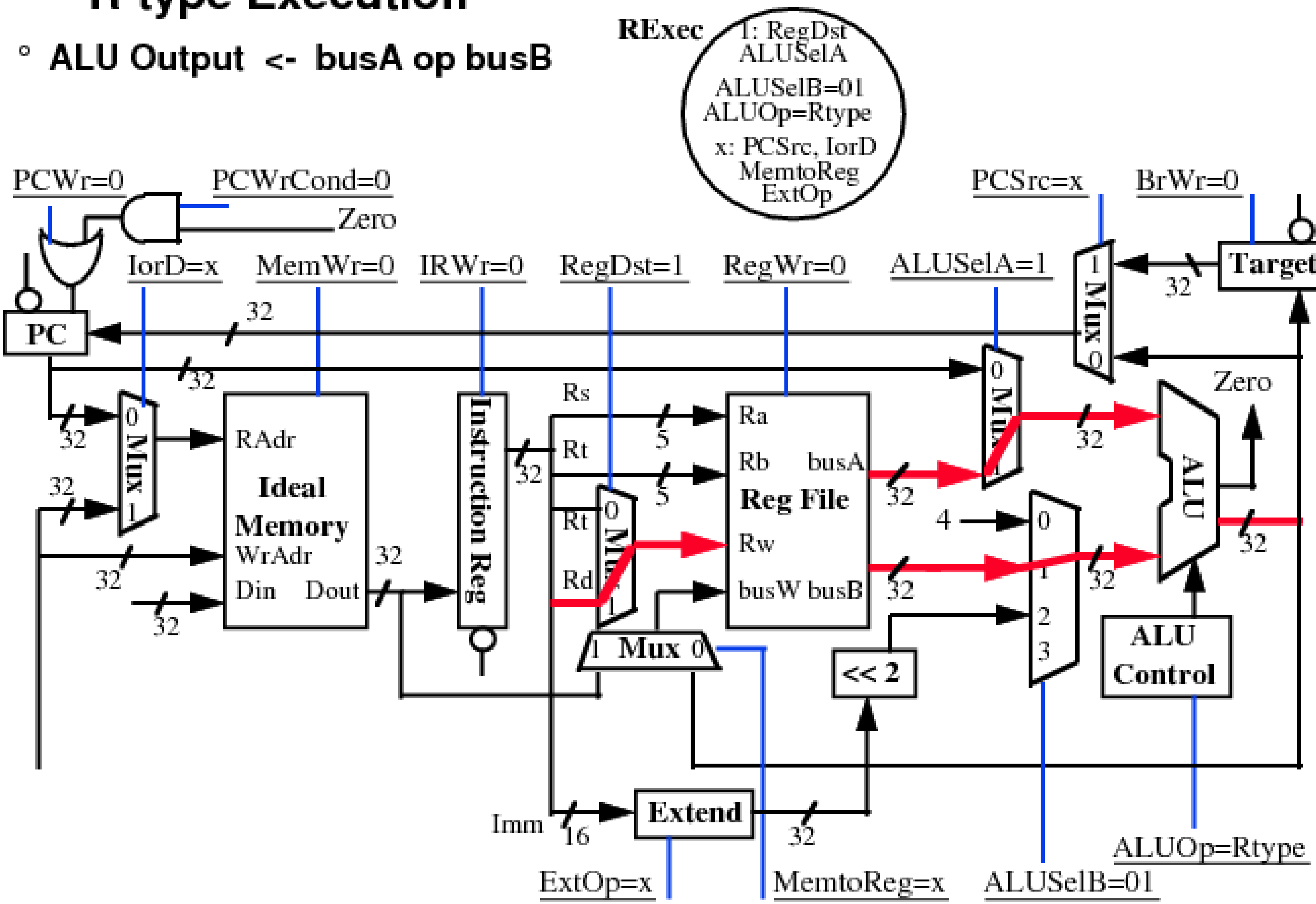
Instruction Decode: We have a R-type!

- Next Cycle: R-type Execution



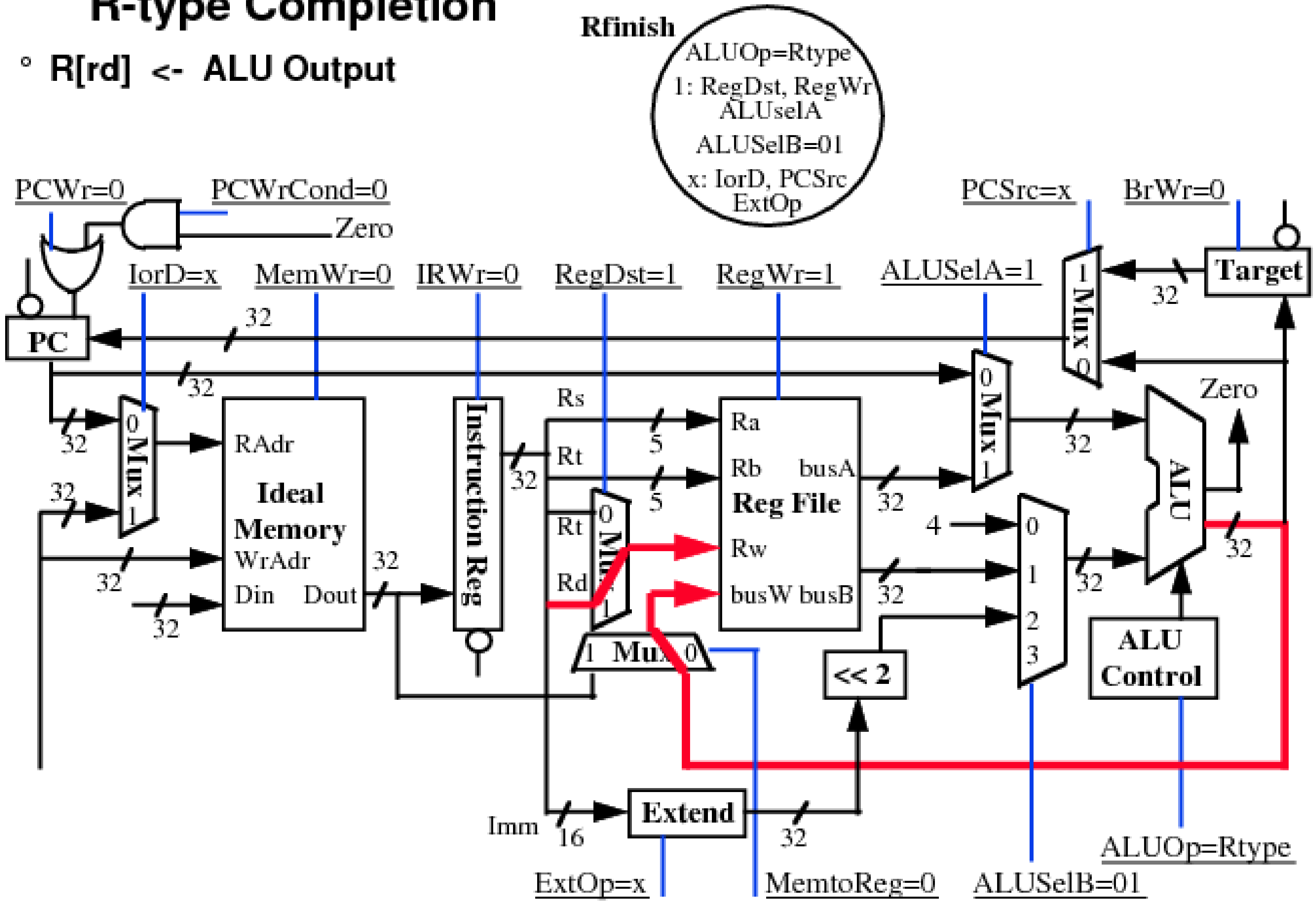
R-type Execution

- ALU Output <- busA op busB



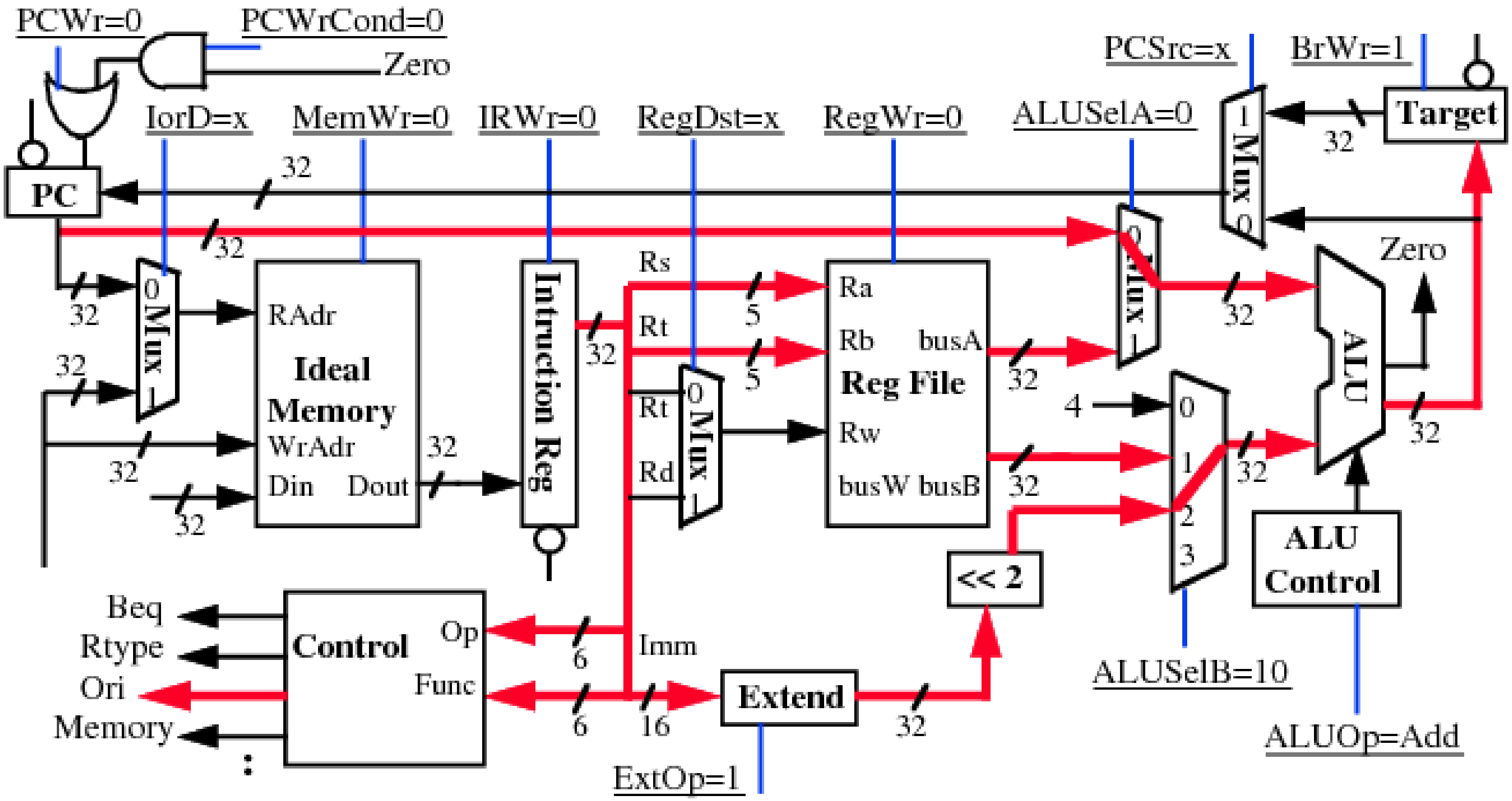
R-type Completion

° R[rd] <- ALU Output



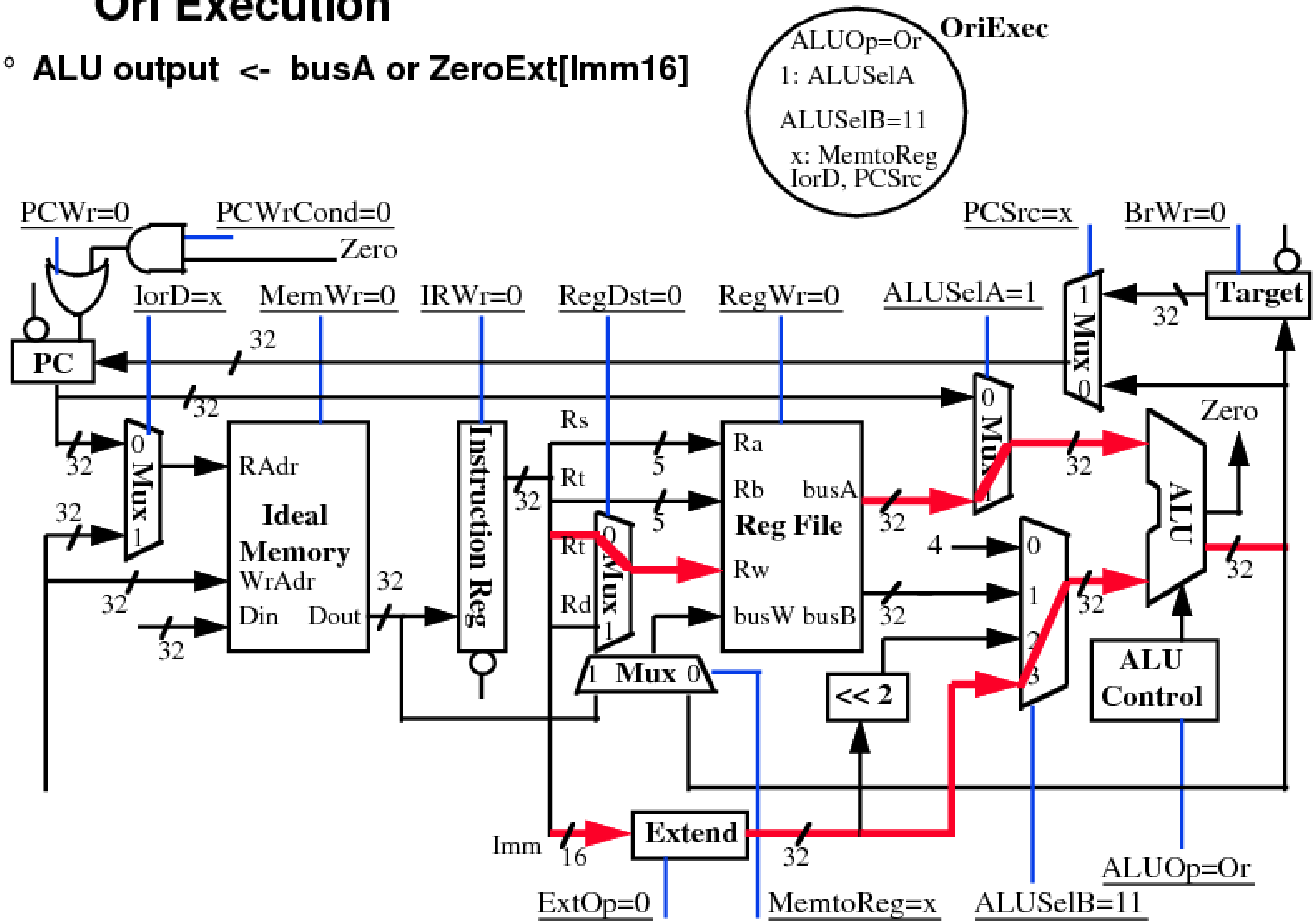
Instruction Decode: We have an Ori!

- Next Cycle: Ori Execution



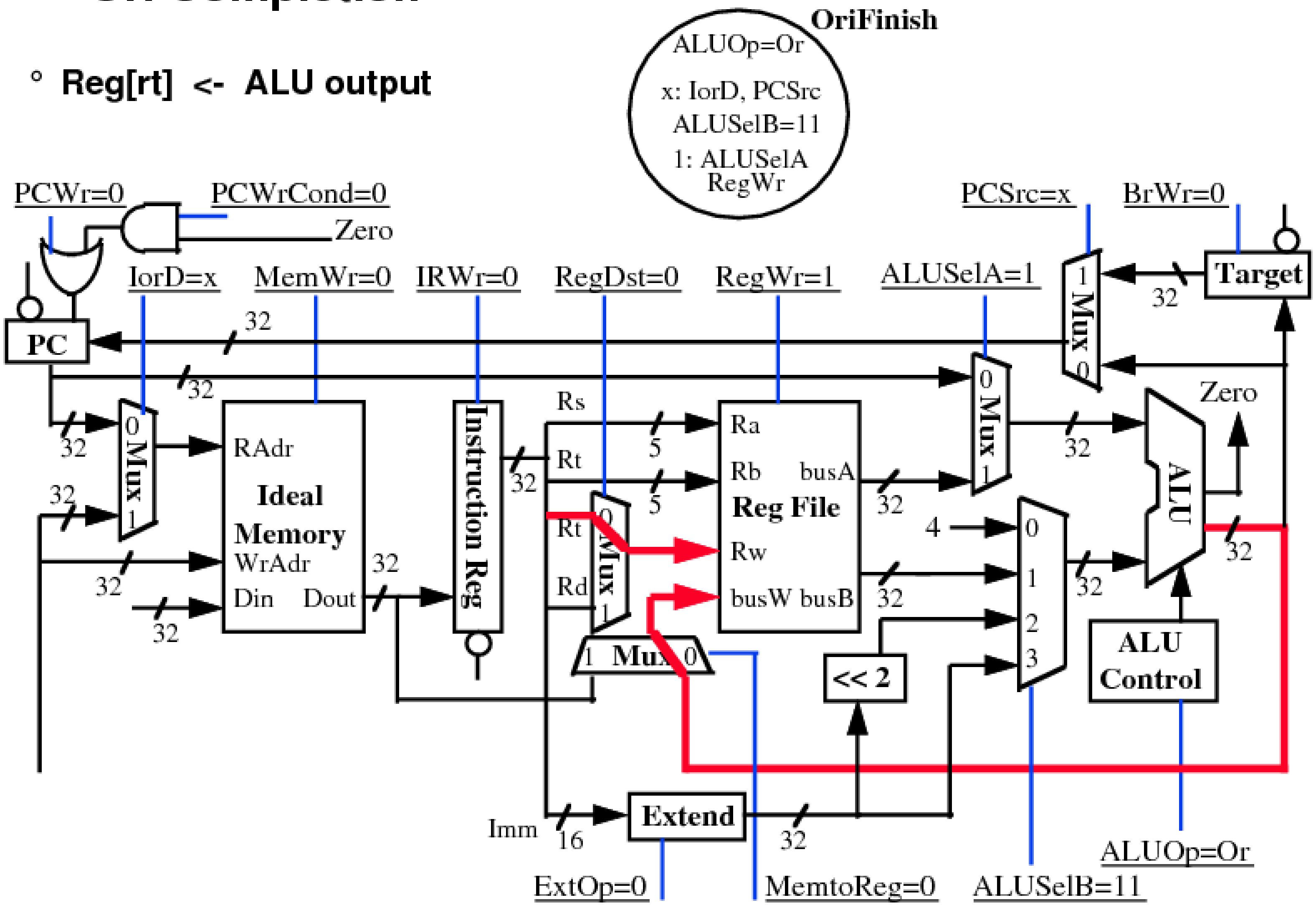
Ori Execution

- ALU output <- busA or ZeroExt[Imm16]



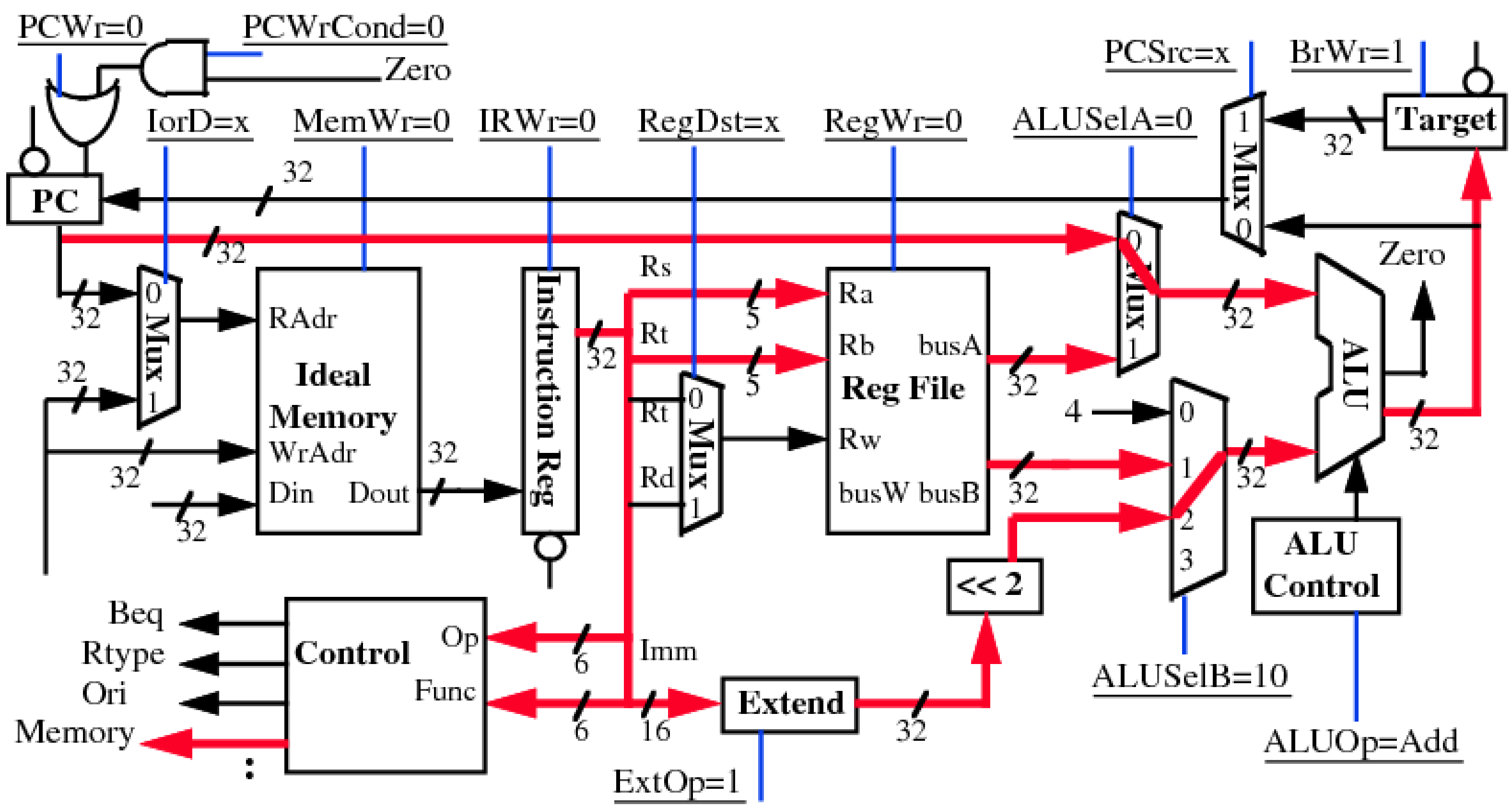
Ori Completion

° Reg[rt] <- ALU output



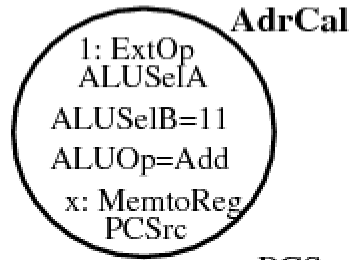
Instruction Decode: We have a Memory Access!

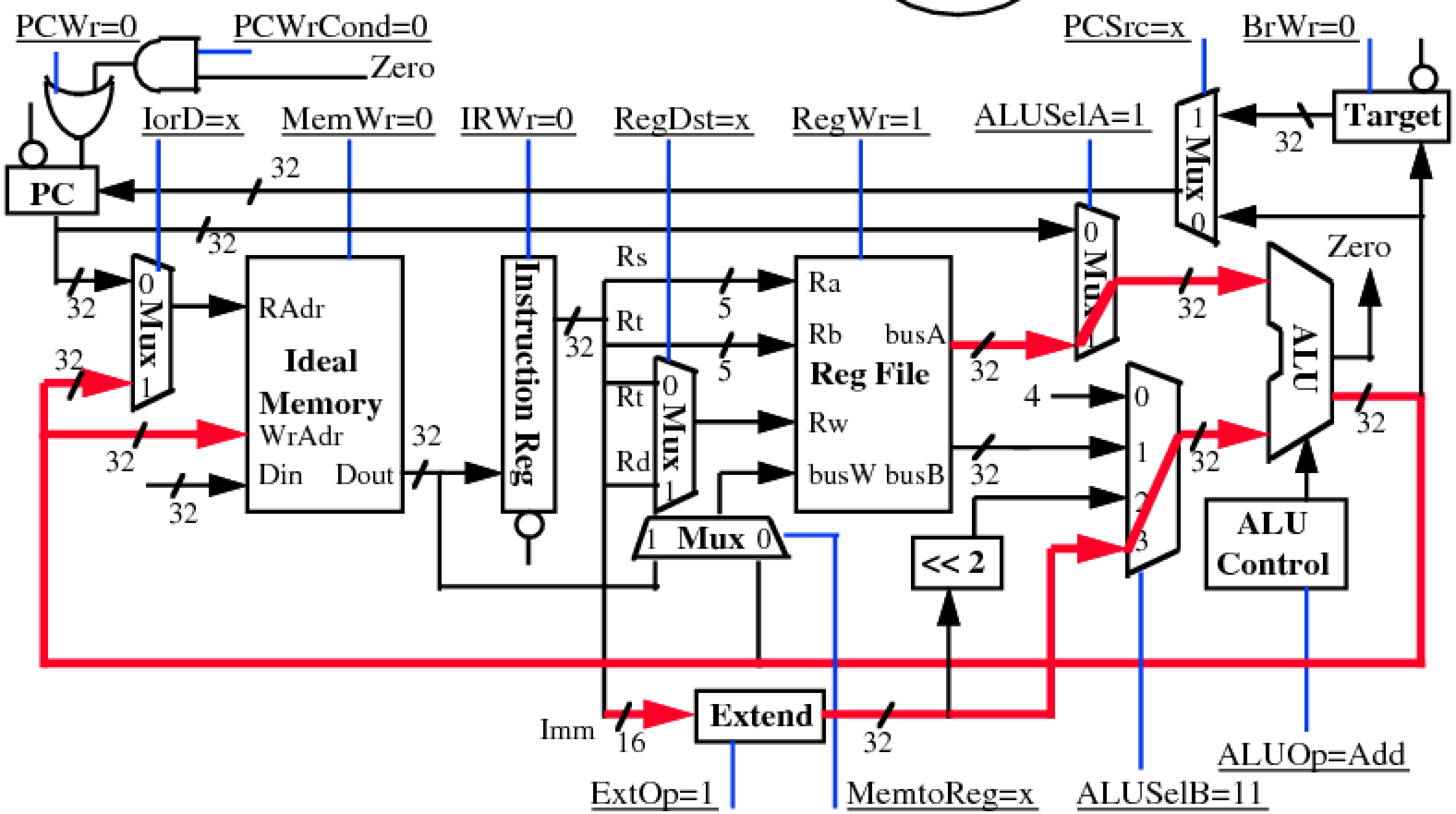
° **Next Cycle: Memory Address Calculation**



Memory Address Calculation

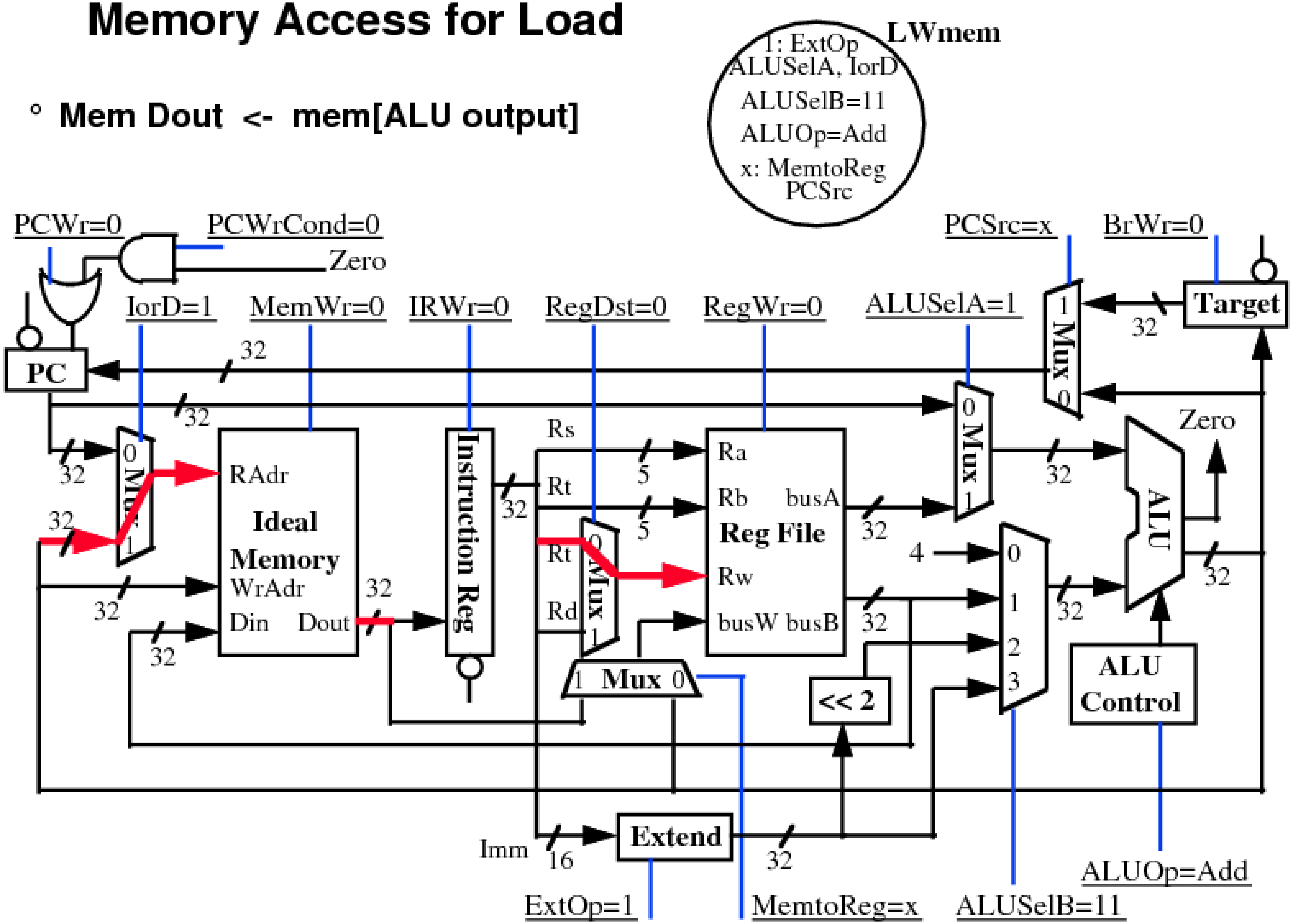
- ALU output <- busA + SignExt[Imm16]





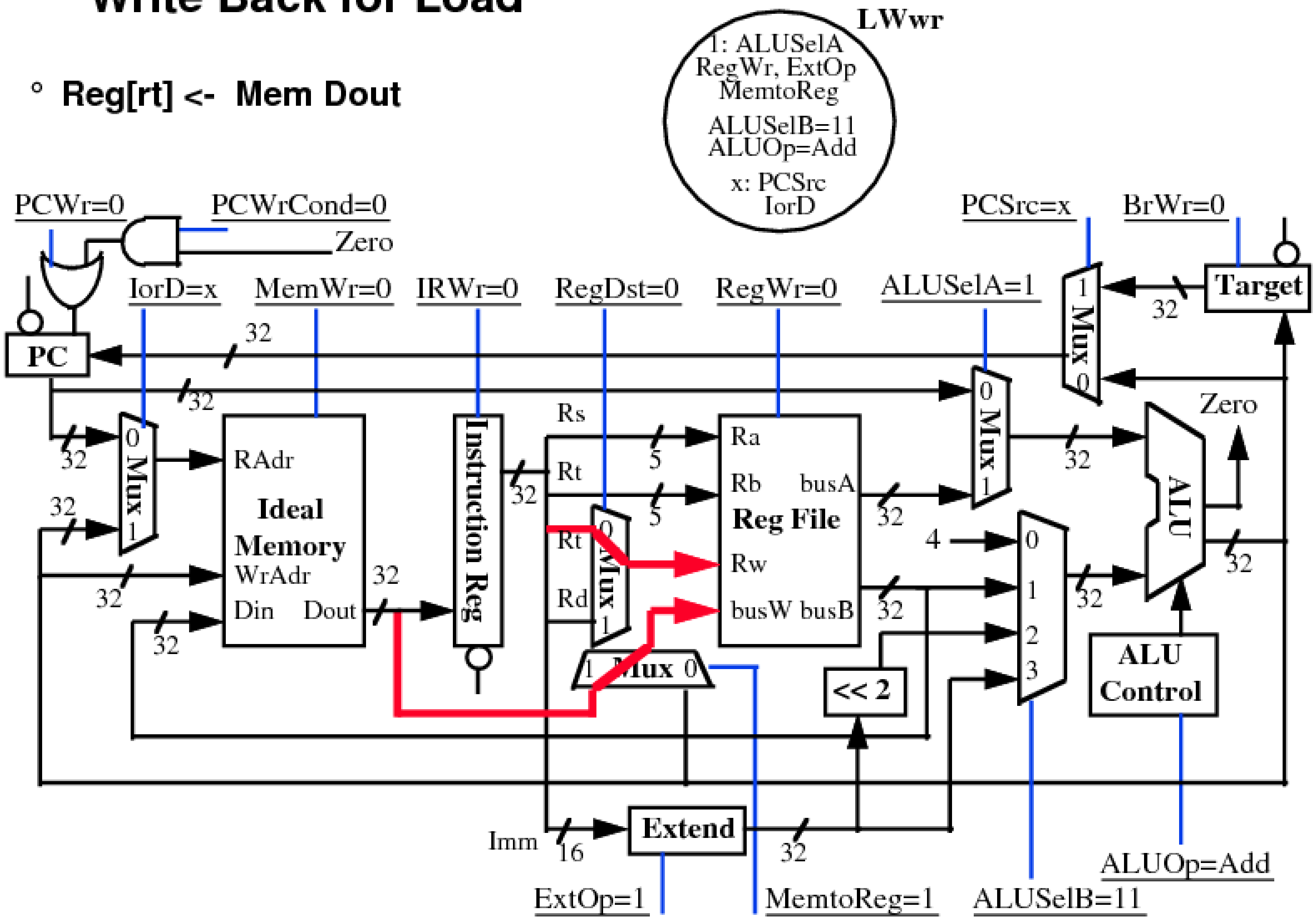
Memory Access for Load

° Mem Dout <- mem[ALU output]



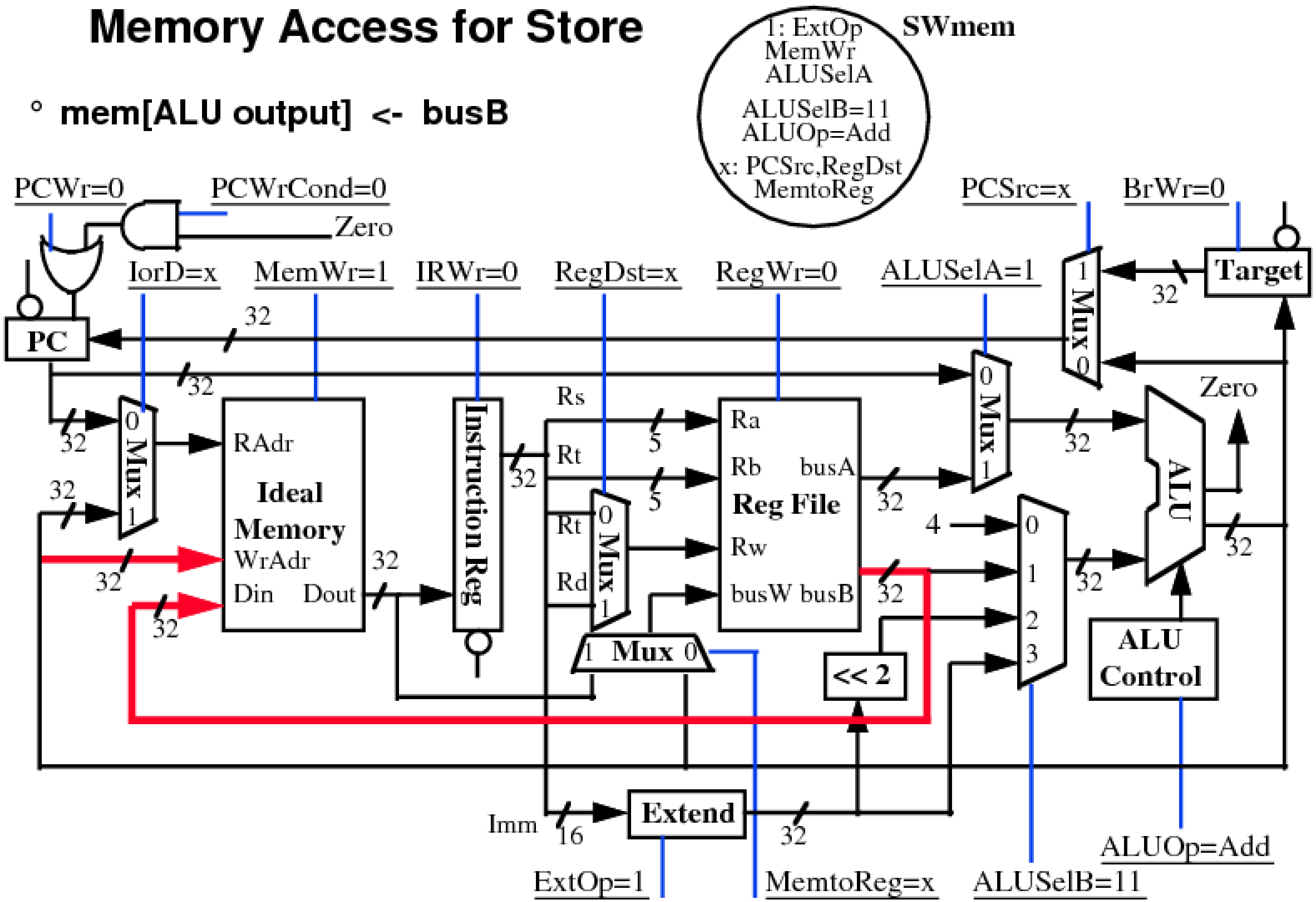
Write Back for Load

- Reg[rt] <- Mem Dout

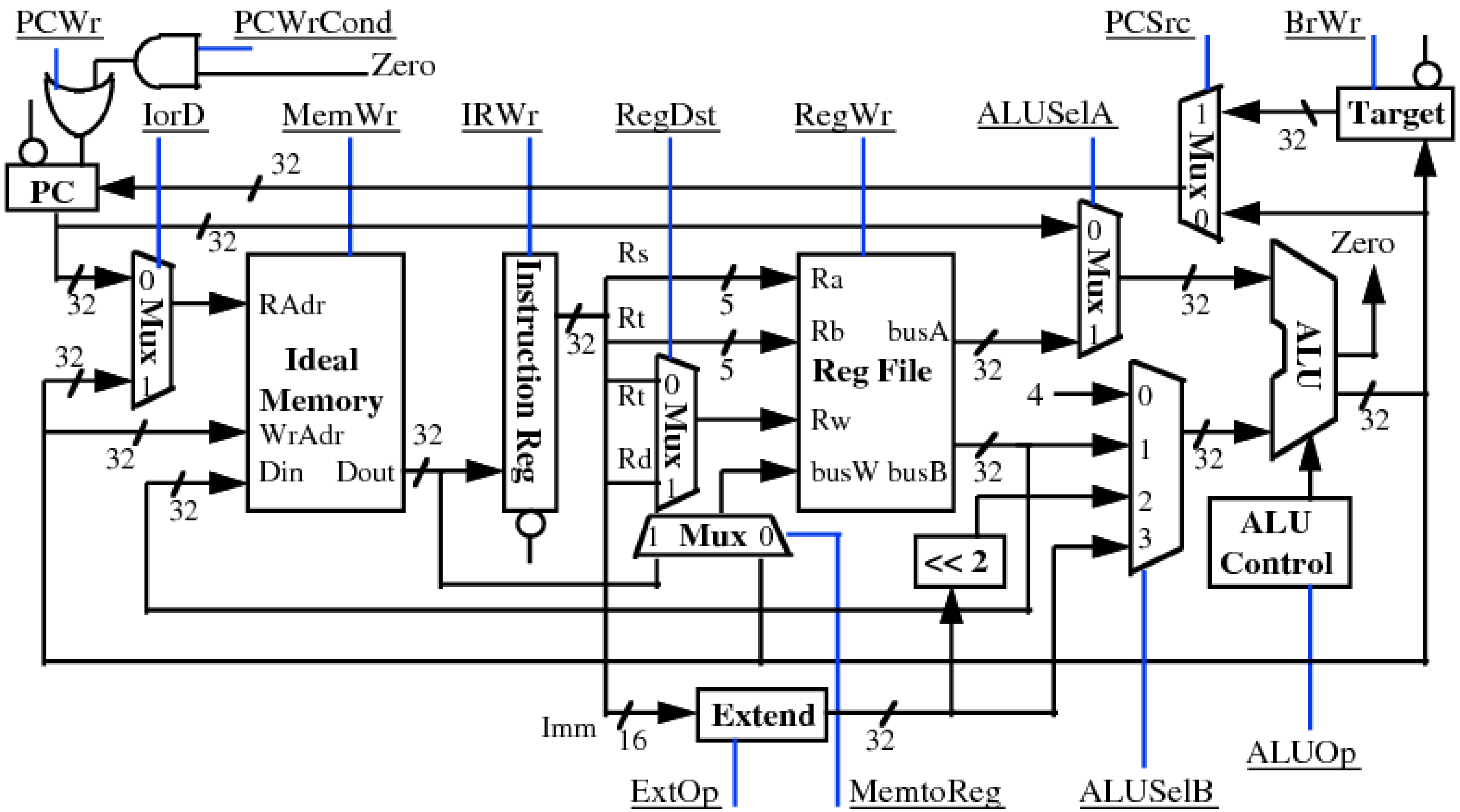


Memory Access for Store

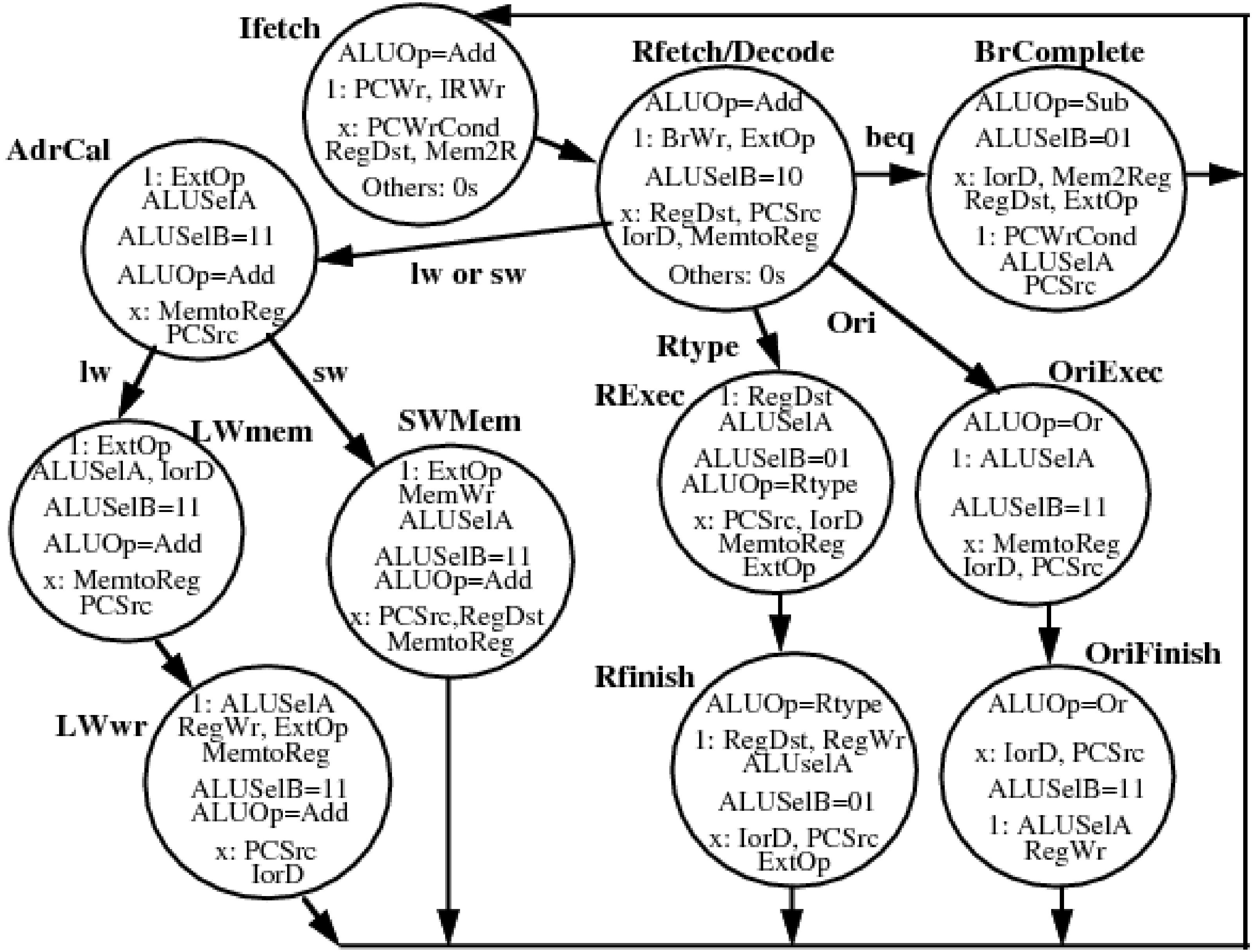
° mem[ALU output] <- busB



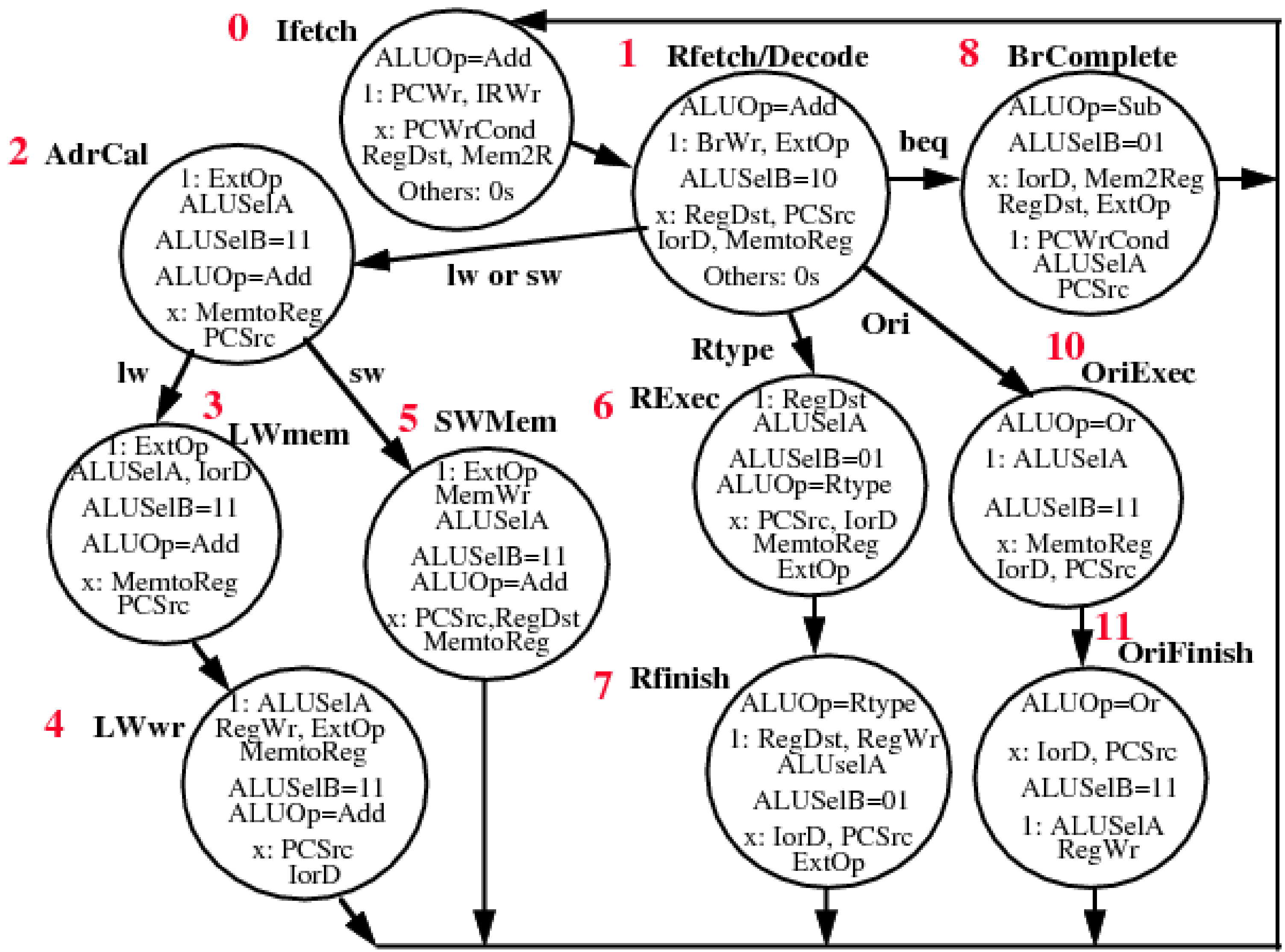
Putting it all together: Multiple Cycle Datapath



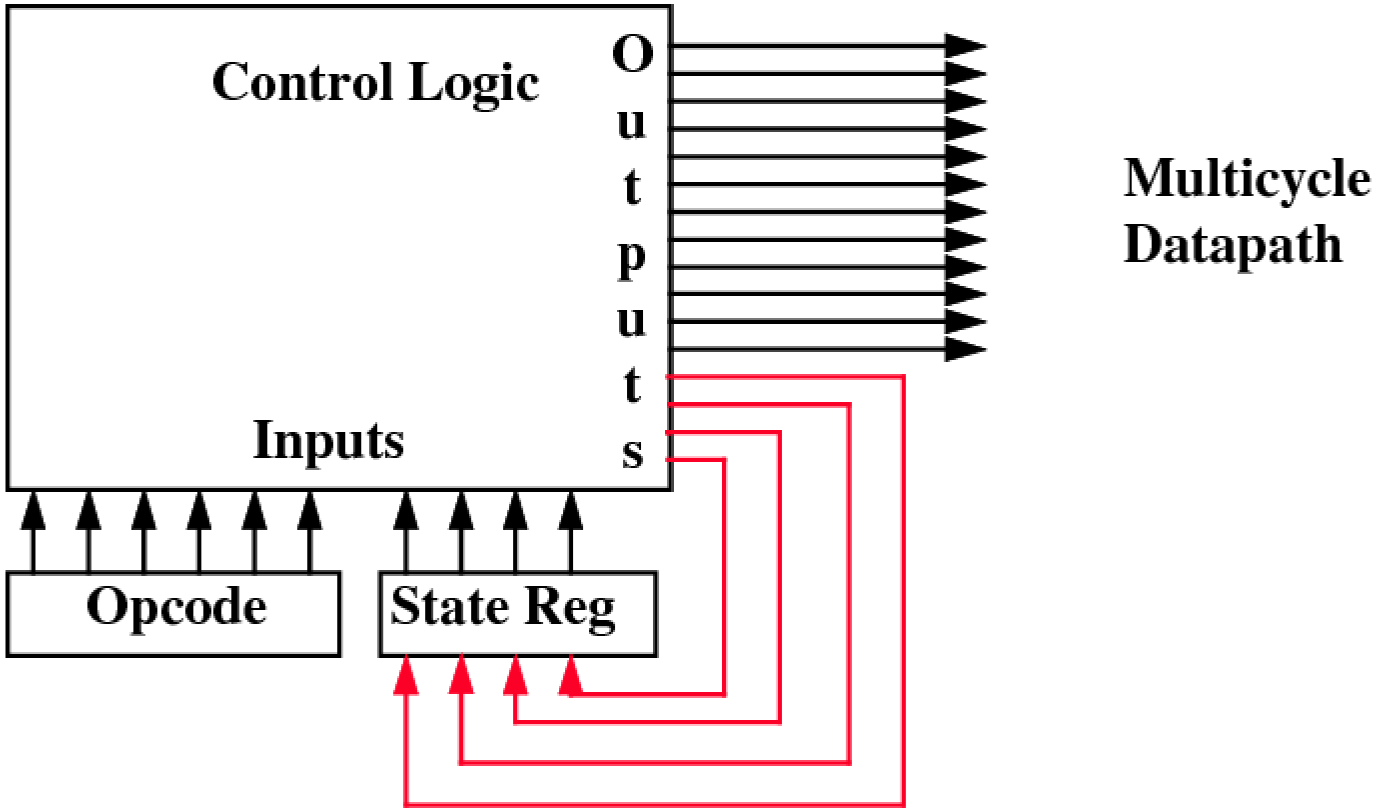
Putting it all together: Control State Diagram



Control Logic in the Form of Finite State Diagram



Sequencing Control: Explicit Next State Function

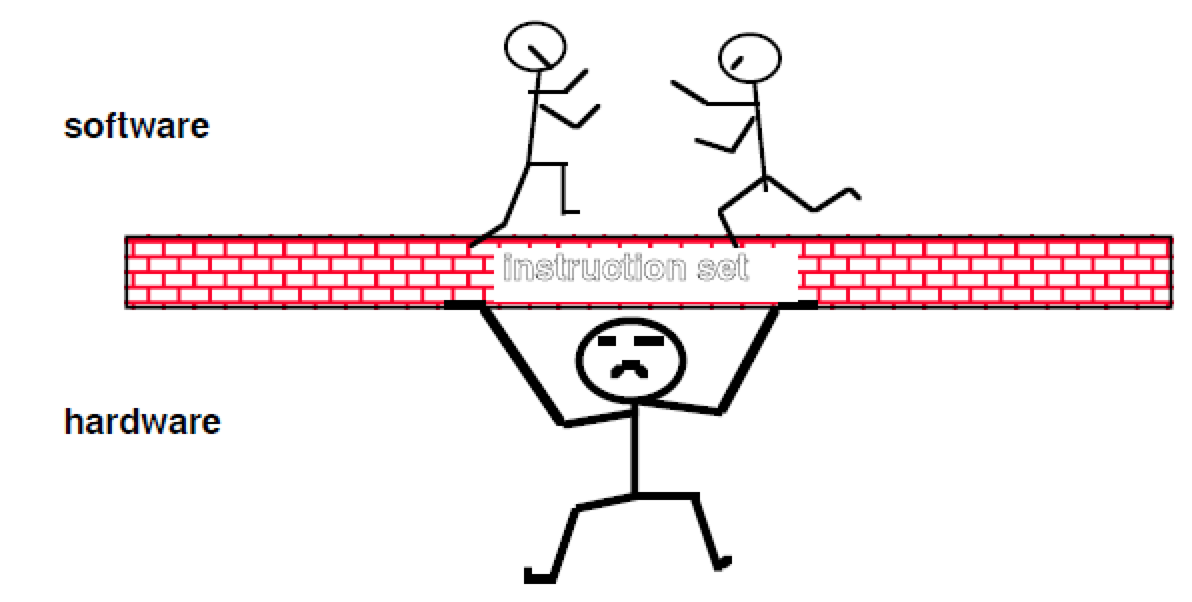


The Big Picture: Performance Perspective

- Performance of a machine was determined by
  - Instruction Count
  - Clock cycle Time
  - Clock cycles per instruction
- Processor Design (data path and control) will determine
  - Clock cycle time
  - Clock cycles per instruction
- We shall first design a Single Cycle Processor
  - Advantage: One clock cycle per instruction
  - Disadvantage: Long cycle time

Summary

- Disadvantages of the Single Cycle Processor
  - Long cycle time
  - Cycle time is too long for all instructions except load
- Multiple cycle processor
  - Divide the instruction into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle



COMPUTER SYSTEMS ORGANIZATION

Acknowledgment: Almost all of these slides are based on Dave Patterson's CS152 Lecture Slides at UC, Berkeyley

MIPS CPU Instructions

Three Types of CPU Instructions

- R-type
- I-Type
- J-Type

R-Type Instructions

- R-type Instruction Format

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	sa	function
6	5	5	5	5	6

R-Type Instructions: ADD

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000
6	5	5	5	5	6

Format: `ADD rd, rs, rt` **MIPS32**

- Format: ADD rd, rs, rt
- R[rd] = R[rs] + R[rt]
- 32-bit 2's Complement Addition
- Destination register will not be modified if integer overflow exceptions occurs.

R-Type Instructions: SUB

31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
SPECIAL 000000	rs	rt	rd	0 00000	SUB 100010
6	5	5	5	5	6

- Format: SUB rd, rs, rt
- R[rd] = R[rs] - R[rt]
- 32-bit signed subtraction
- Destination register will not be modified if integer overflow exceptions occurs.

R-Type Instructions: AND

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	AND 100100
6	5	5	5	5	6

Format: `AND rd, rs, rt` **MIPS32**

- Format: AND rd, rs, rt
- R[rd] = R[rs] & R[rt]

R-Type Instructions: OR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	OR 100101
6	5	5	5	5	6

Format: `OR rd, rs, rt` **MIPS32**

- Format: OR rd, rs, rt
- R[rd] = R[rs] | R[rt]

R-Type Instructions: SLT

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SLT 101010
6	5	5	5	5	6

Format: `SLT rd, rs, rt` **MIPS32**

- Format: SLT rd, rs, rt
- R[rd] = R[rs] < R[rt] ? 1:0
- Signed comparision

There are many other R-type instructions like ADDU, NOR, XOR etc.

R-Type Instructions: SLL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SLL 000000
6	5	5	5	5	6

Format: `SLL rd, rt, sa` **MIPS32**

- Format: SLL rd, rt, sa
- R[rd] = R[rt] << sa

Note: In our processor design we do not implement shift instructions.

I-type Instructions

31 … 26	25 … 21	20 … 16	15 … 0
opcode	rs	rt	immediate
6	5	5	16

I-type Instructions

31–26	25–21	20–16	15–0
ADDI 001000	rs	rt	immediate
6	5	5	16

Format: `ADDI rt, rs, immediate` **MIPS32**

- Format: ADDI rt, rs, immediate
- R[rt] = R[rs] + sign_extend(immediate)
- immediate is 16-bit signed immediate
- 32-bit 2'complement addition
- Destination register will not be updated if integer overflow exception occurs

I-type Instructions

31 26	25 21	20 16	15 0
ANDI 001100	rs	rt	immediate
6	5	5	16

Format: `ANDI rt, rs, immediate` **MIPS32**

- Format: ANDI rt, rs, immediate
- R[rt] = R[rs] & zero_extend(immediate)

I-type Instructions: ORI

31 … 26	25 … 21	20 … 16	15 … 0
ORI 001101	rs	rt	immediate
6	5	5	16

Format: `ORI rt, rs, immediate` **MIPS32**

- Format: ORI rt, rs, immediate
- R[rt] = R[rs] | zero_extend(immediate)

I-type Instructions: SLTI

31 26	25 21	20 16	15 0
SLTI 001010	rs	rt	immediate
6	5	5	16

Format: `SLTI rt, rs, immediate` **MIPS32**

- Format: SLTI rt, rs, immediate
- R[rt] = R[rs] < sign_extend(immediate) ? 1:0

I-type Instructions: LW

31 … 26	25 … 21	20 … 16	15 … 0
LW 100011	base	rt	offset
6	5	5	16

Format: `LW rt, offset(base)` **MIPS32**

- Format: LW rt, offset(base)
- vdddr = sign_extend(offset) + R[base]
- R[rt] = Mem[vaddr]
- If vaddr is now word-aligned, an exception will be raised.

I-type Instructions: SW

31 26	25 21	20 16	15 0
SW 101011	base	rt	offset
6	5	5	16

Format: `SW rt, offset(base)` **MIPS32**

- Format: SW rt, offset(base)
- vdddr = sign_extend(offset) + R[base]
- Mem[vaddr] = R[rt]
- If vaddr is now word-aligned, an exception will be raised.

I-type Instructions: SLTI

31 – 26	25 – 21	20 – 16	15 – 0
SLTI 001010	rs	rt	immediate
6	5	5	16

Format: `SLTI rt, rs, immediate` **MIPS32**

- Format: SLTI rt, rs, immediate
- R[rt] = R[rs] < sign_extend(immediate) ? 1:0

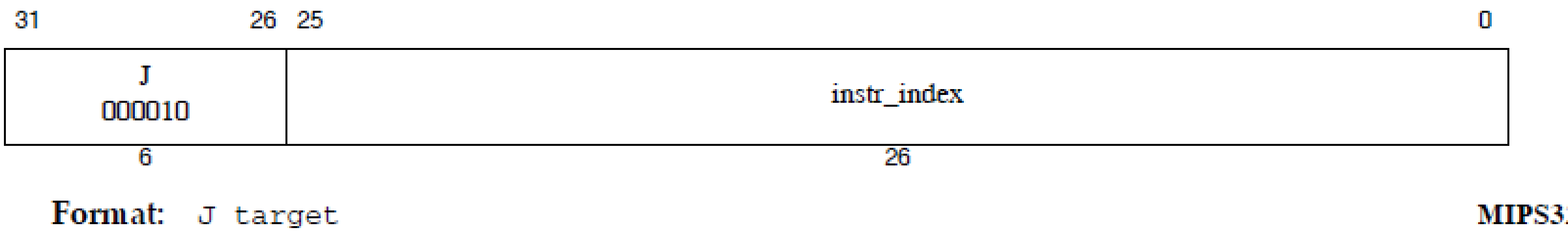
I-Type Instructions: BEQ

31 26	25 21	20 16	15 0
BEQ 000100	rs	rt	offset
6	5	5	16

Format: `BEQ rs, rt, offset` **MIPS32**

- Format: BEQ rs, rt, offset
- If R[rs] == R[rt] then
  - PC = addr_of_branch + 4 + sign_extend(offset << 2)

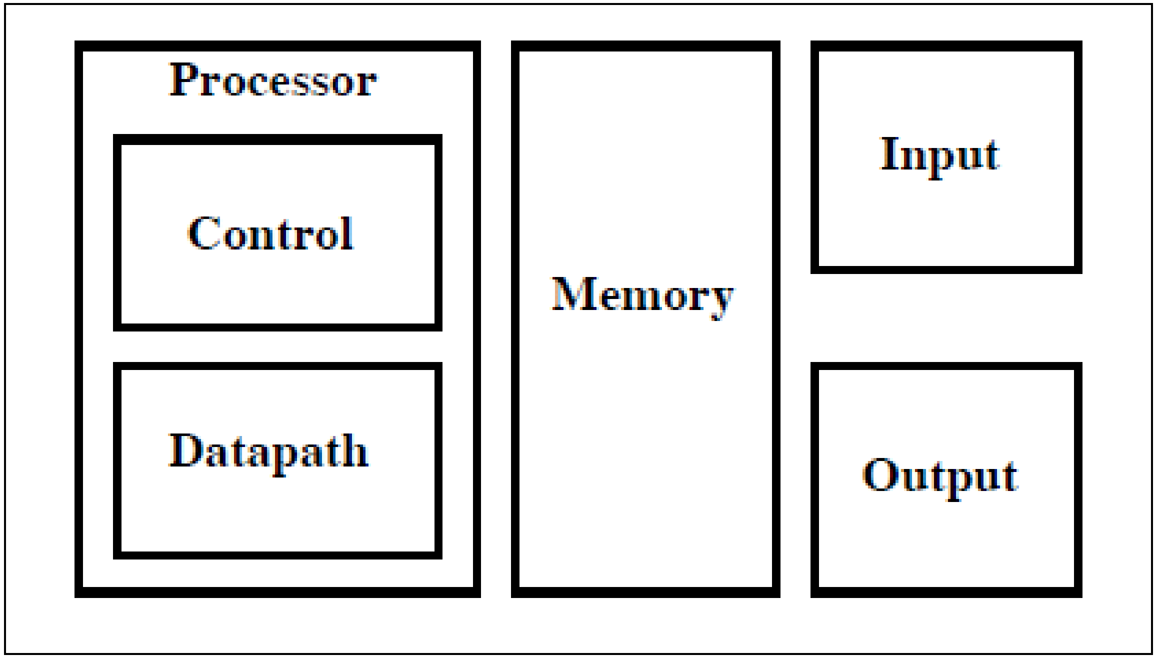
MIPS J-Type Instructions



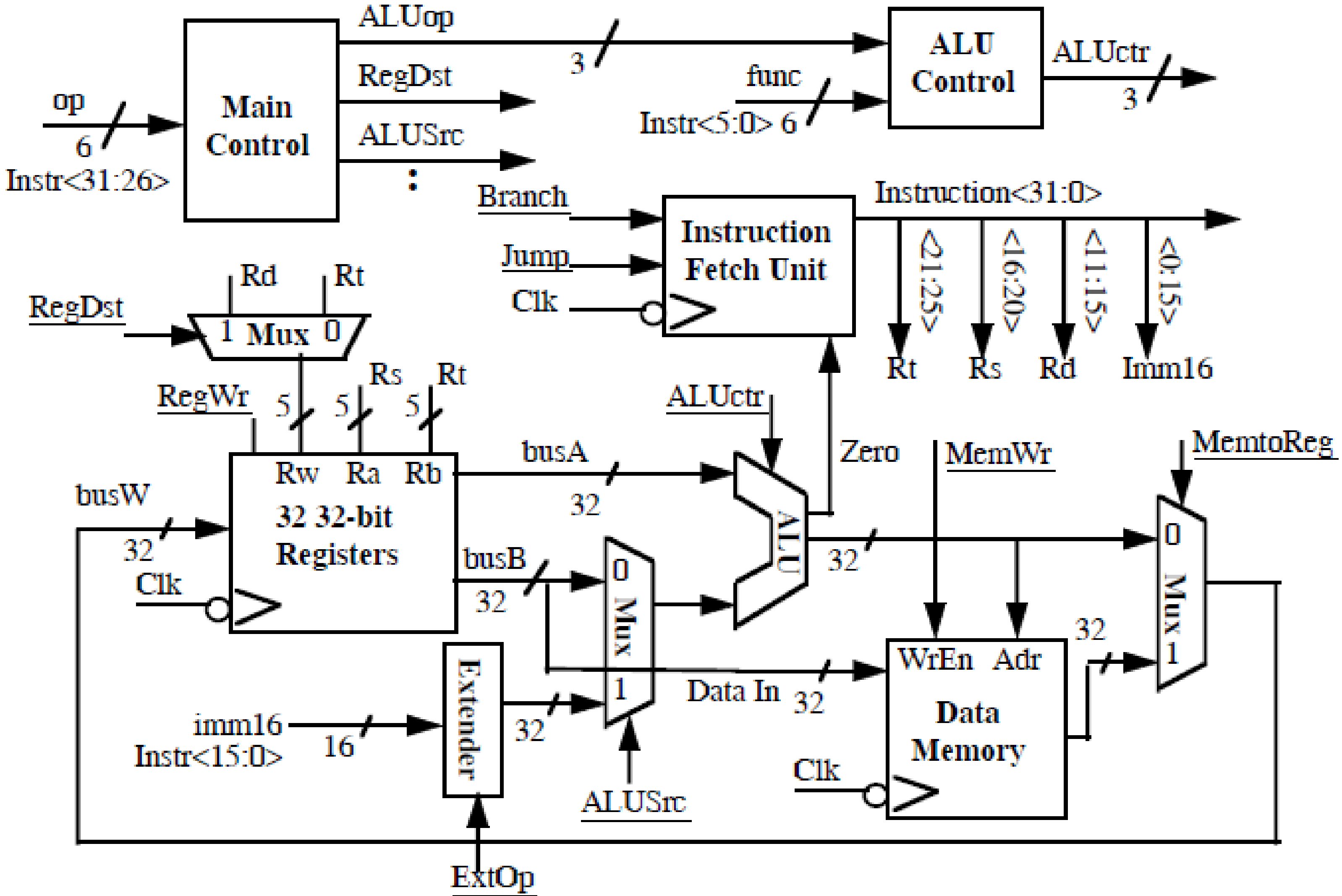
- Format: J target
- Target Address
  - Lower 28 bits: instr_index||00
  - Upper Four Bits: Bits 31, 30, 29, 28 of the address of the Jump Instruction.

The Big Picture: Where are We Now?

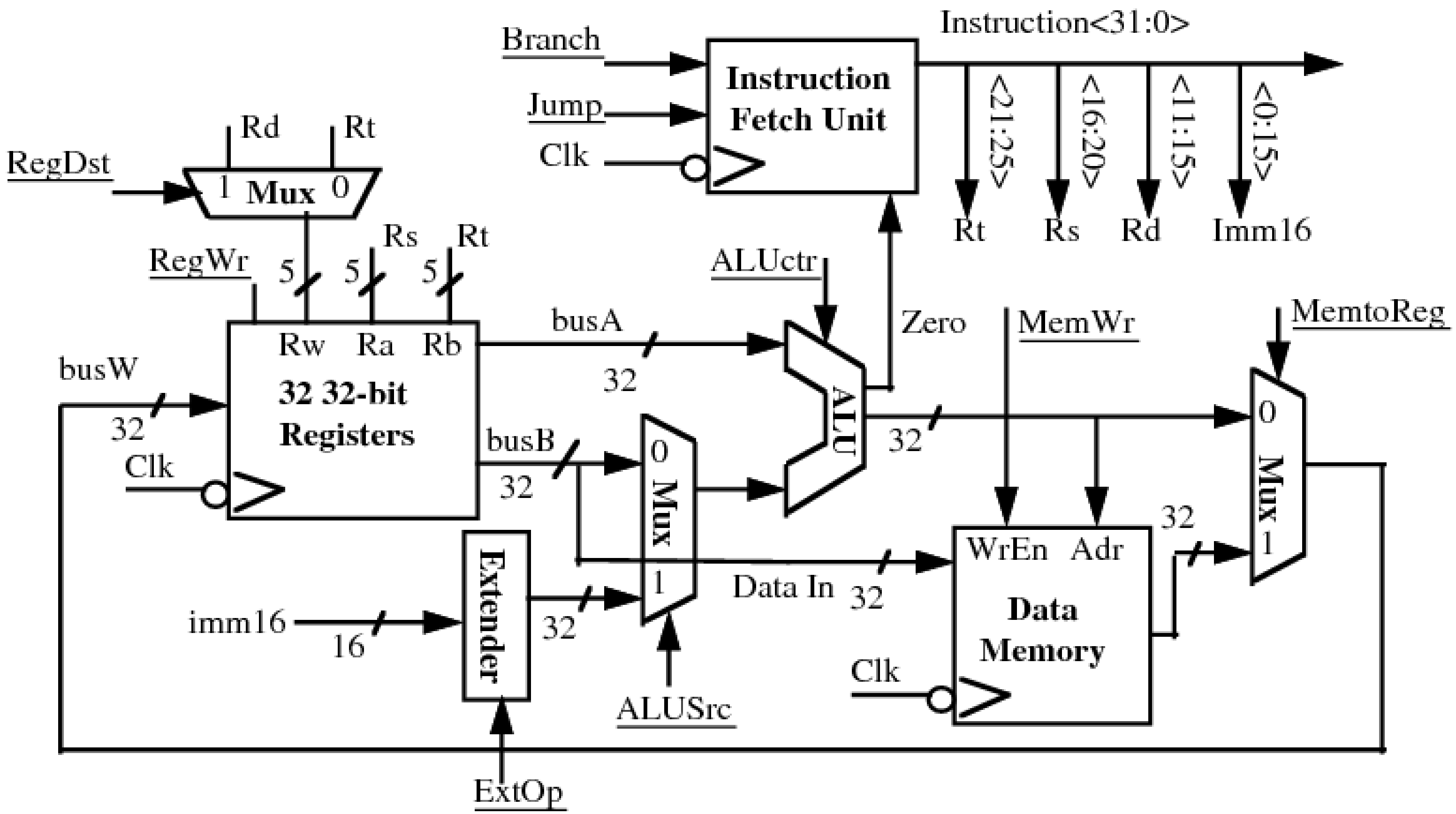
- Five Classic Components of a Computer



MIPS Processor: Control Path + Data Path



Data Path

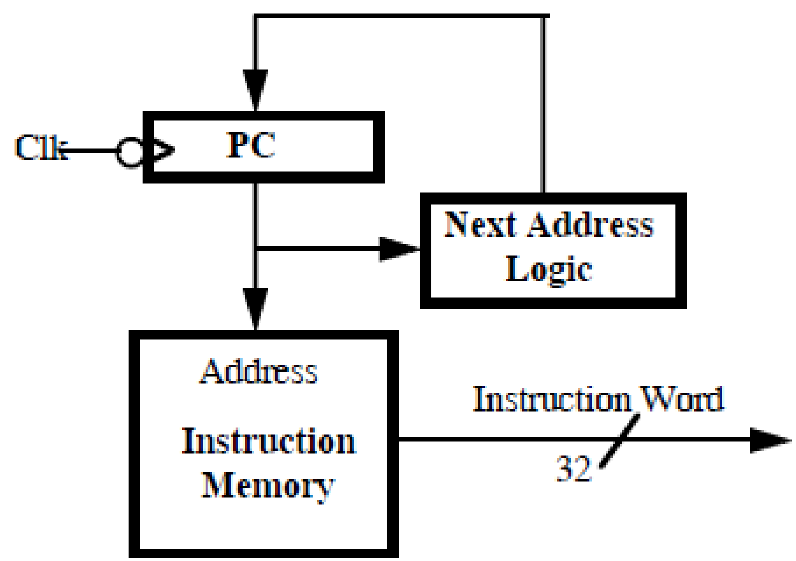


Instruction<31:0>
Branch
Jump
Clk
Instruction Fetch Unit
<21:25>
<16:20>
<11:15>
<0:15>
Rt
Rs
Rd
Imm16
RegDst
Rd
Rt
1 Mux 0
RegWr
5
5
5
Rs
Rt
ALUctr
Zero
MemWr
MemtoReg
busA
32
Rw Ra Rb
32 32-bit Registers
busW
32
Clk
busB
32
0 Mux 1
ALU
32
0 Mux 1
WrEn Adr
Data In
32
Data Memory
Clk
32
imm16
16
Extender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

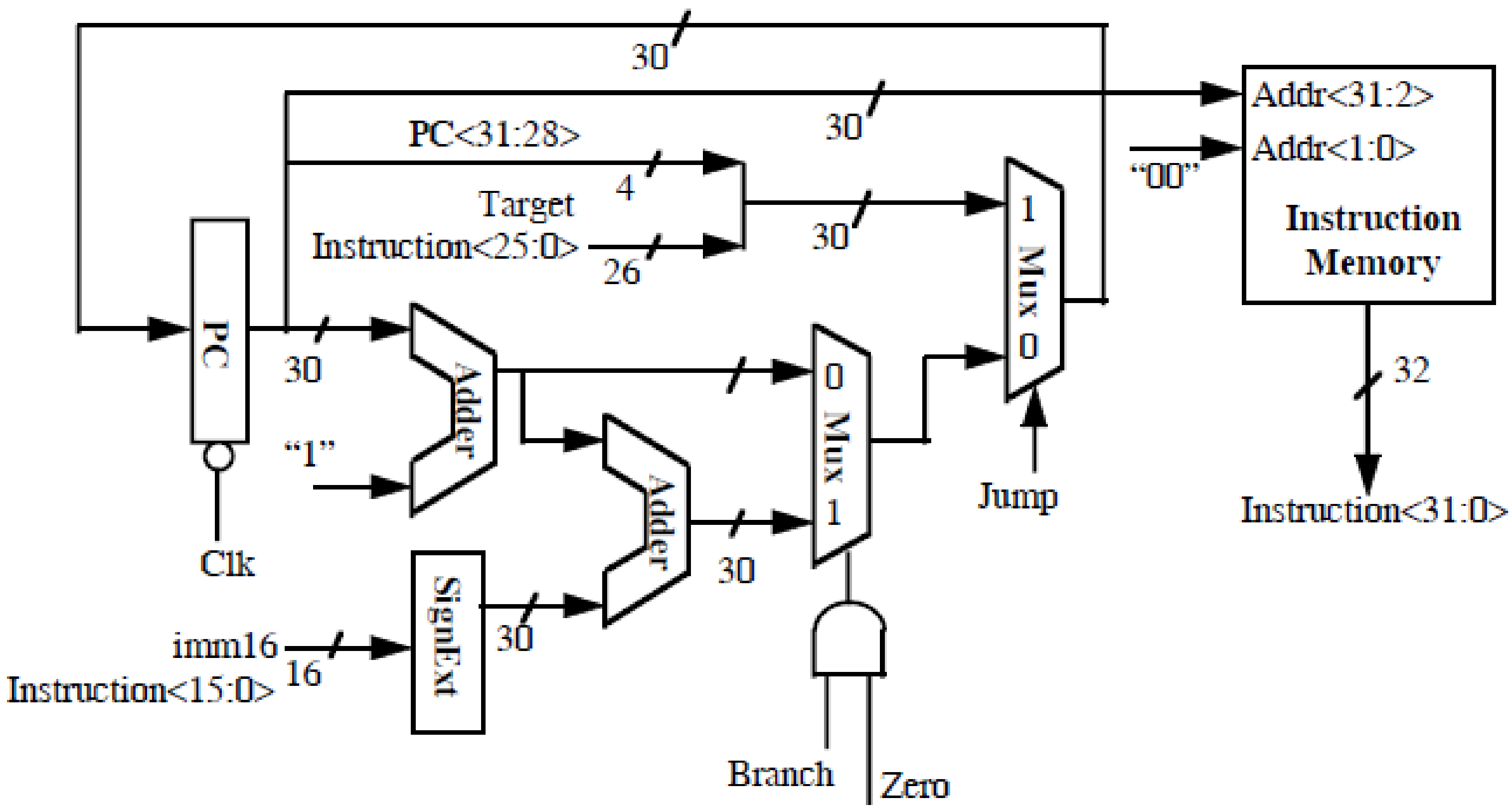
Overview of Instruction Fetch Unit

At a falling clock edge what happens:

- PC gets updated at the falling clock edge
- Fetch the Instruction from the address pointed to by PC
- Pass the PC through the next address logic
- Next value of the PC
  - Sequential Code
    - nextPC = PC + 4
  - Branch and Jump
    - nextPC = "something else"



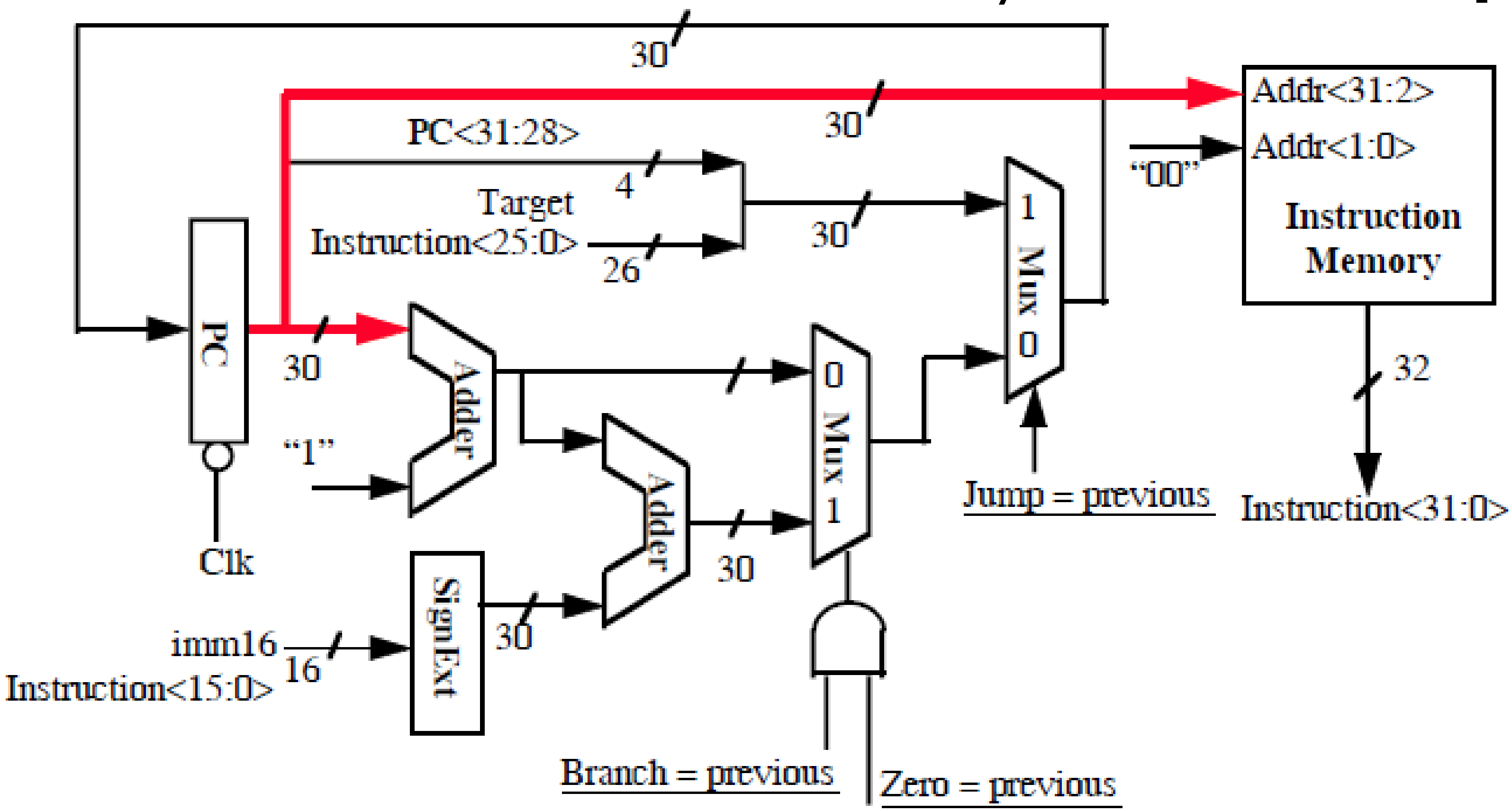
Instruction Fetch Unit



Instruction Fetch Unit at the Beginning of Add / Subtract

The followin two steps are the same for all the instructions.

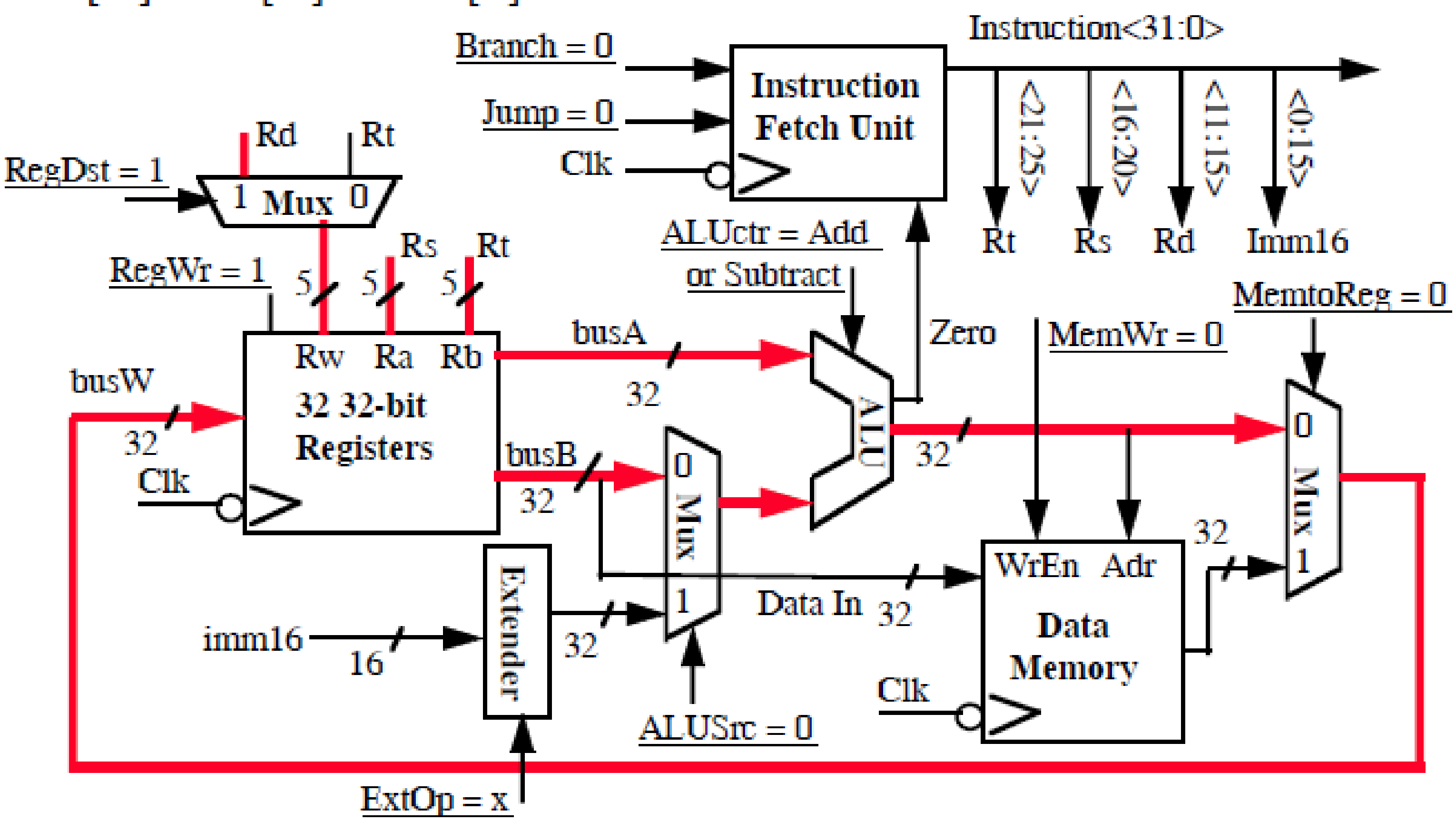
1. PC = nextPC
2. Fetch the instruction from Instruction memory: Instruction = mem[PC]



The Single Cycle Datapath during Add and Subtract

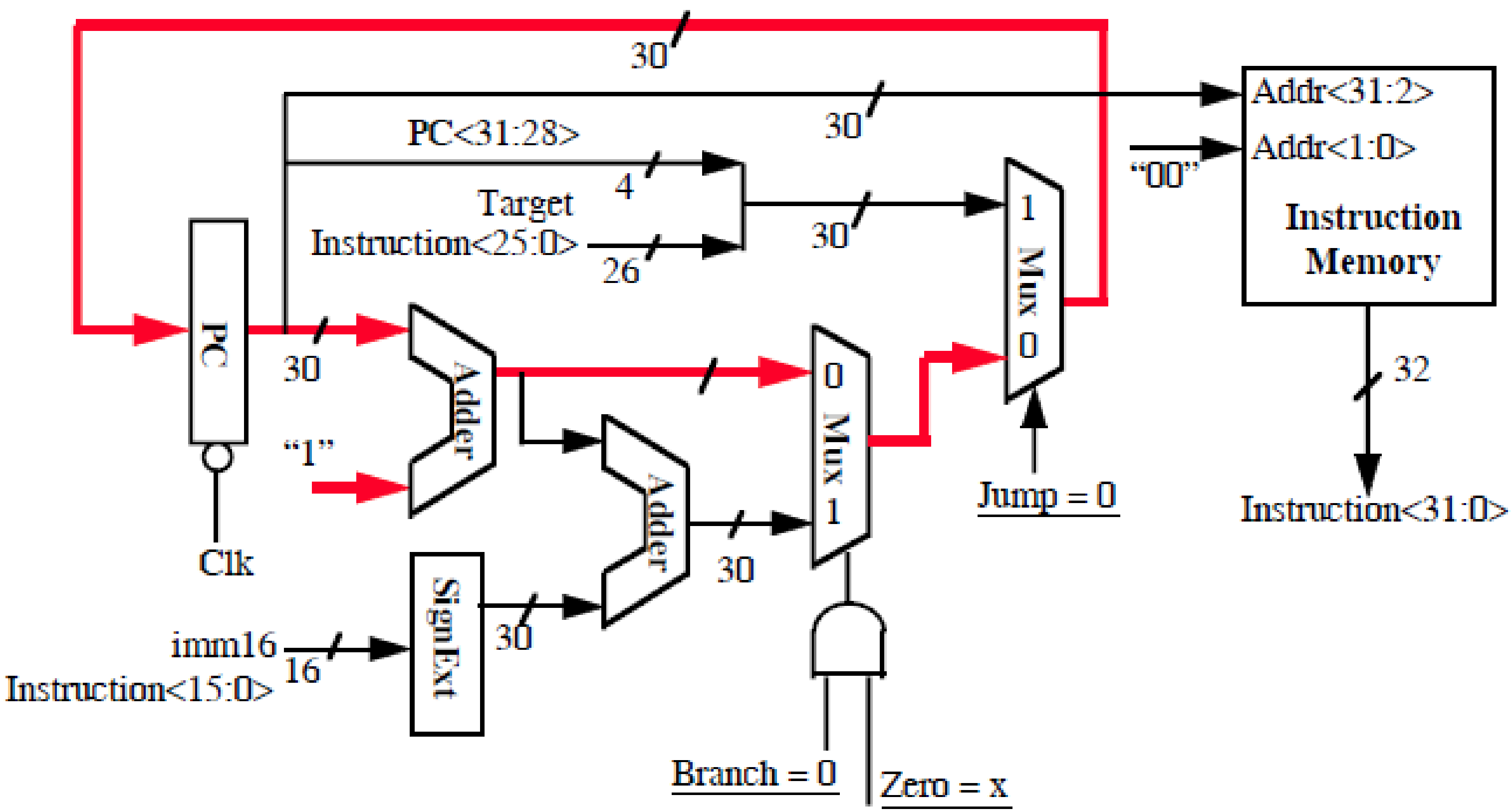
31	26	21	16	11	6	0
op	rs	rt	rd	shamt	funct	

- R[rd] <- R[rs] + / - R[rt]

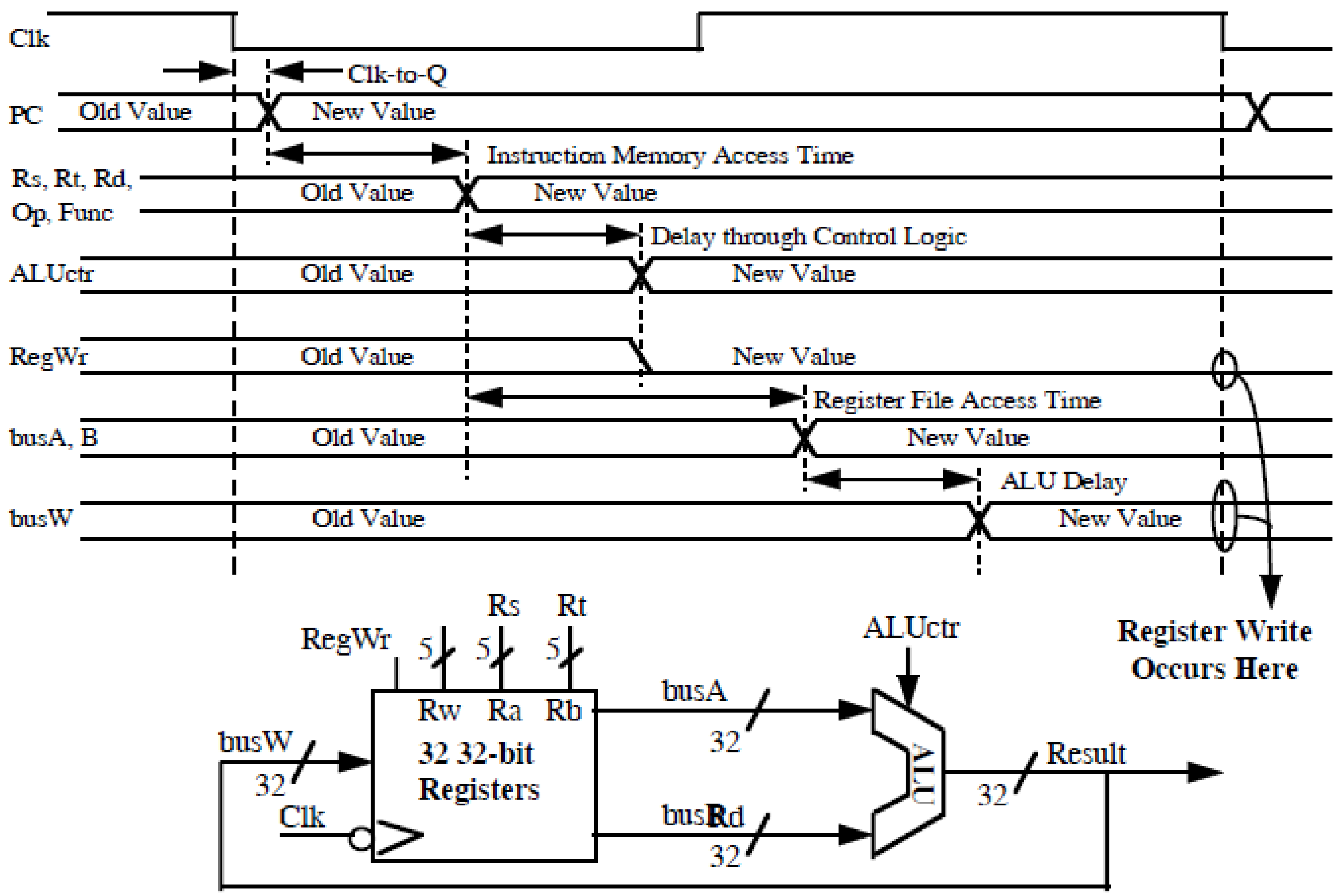


Instruction Fetch Unit at the End of Add and Subtract

- PC = PC + 4
  - This is the same for all instructions except: Branch and Jump



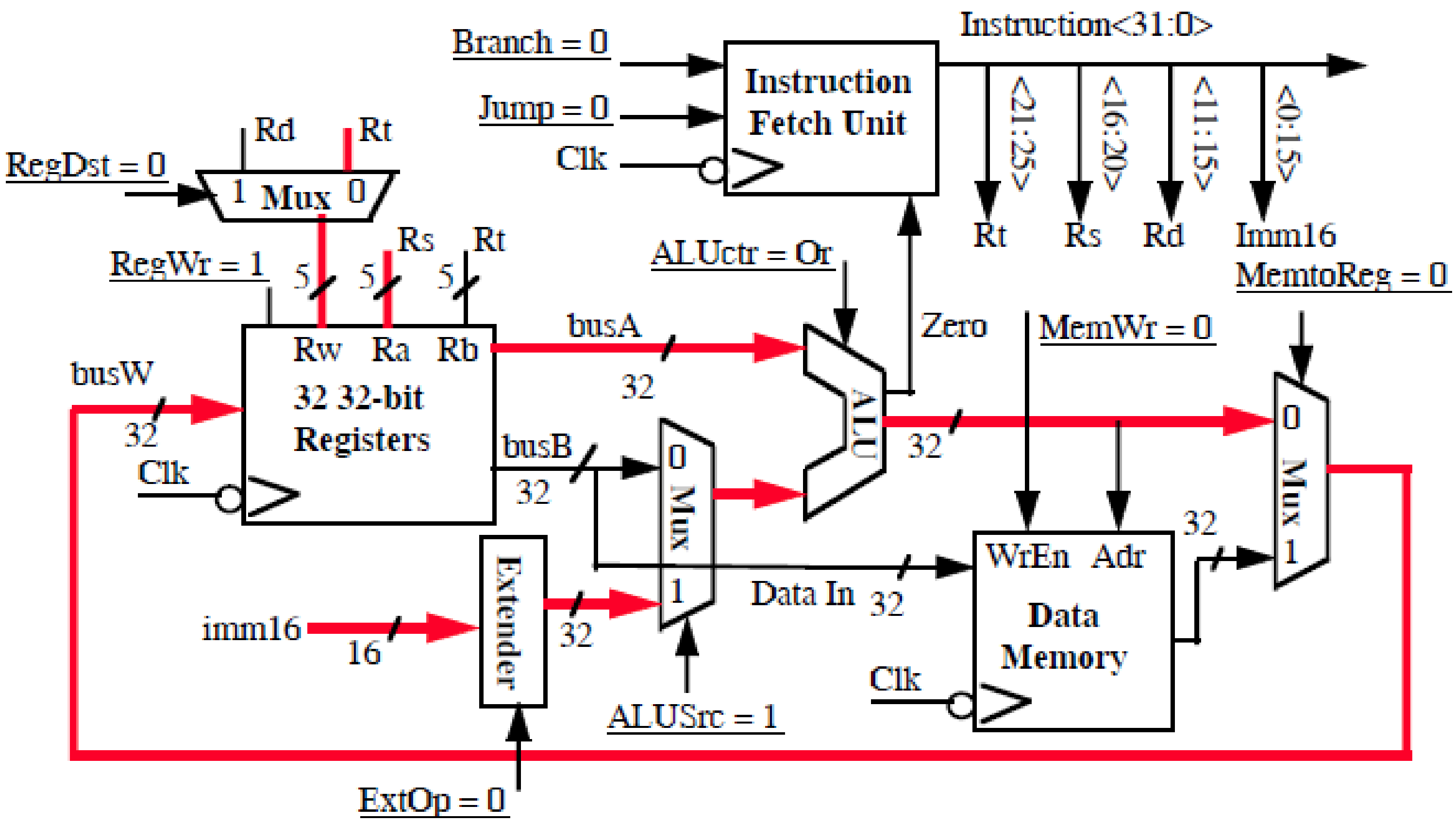
Register – Register Timing



The Single Cycle Datapath during Or Immediate

31 op 26	rs 21	rt 16	immediate 0

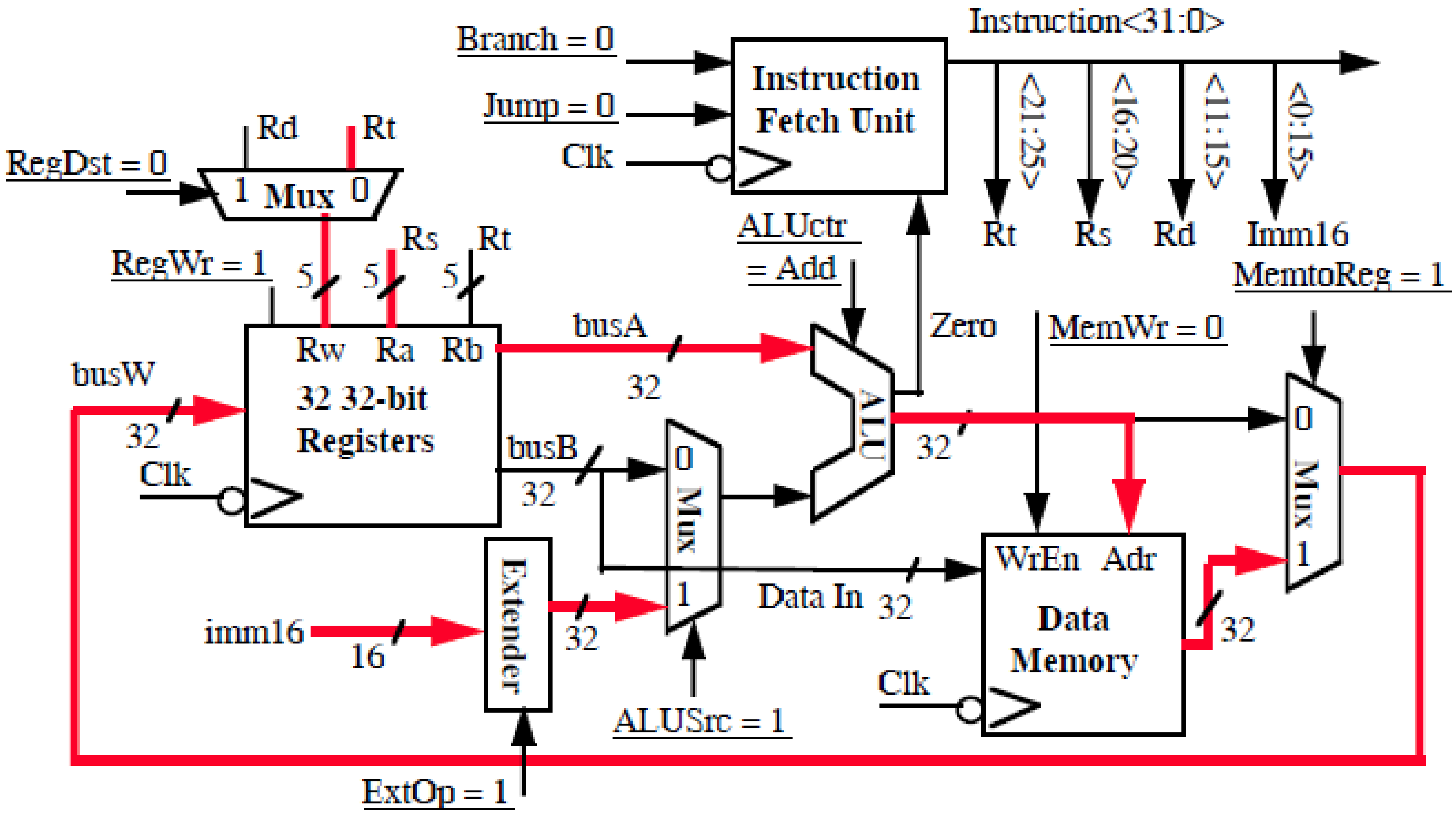
- R[rt] <- R[rs] or ZeroExt[Imm16]



The Single Cycle Datapath during Load

31	26	21	16	0
op	rs	rt	immediate	

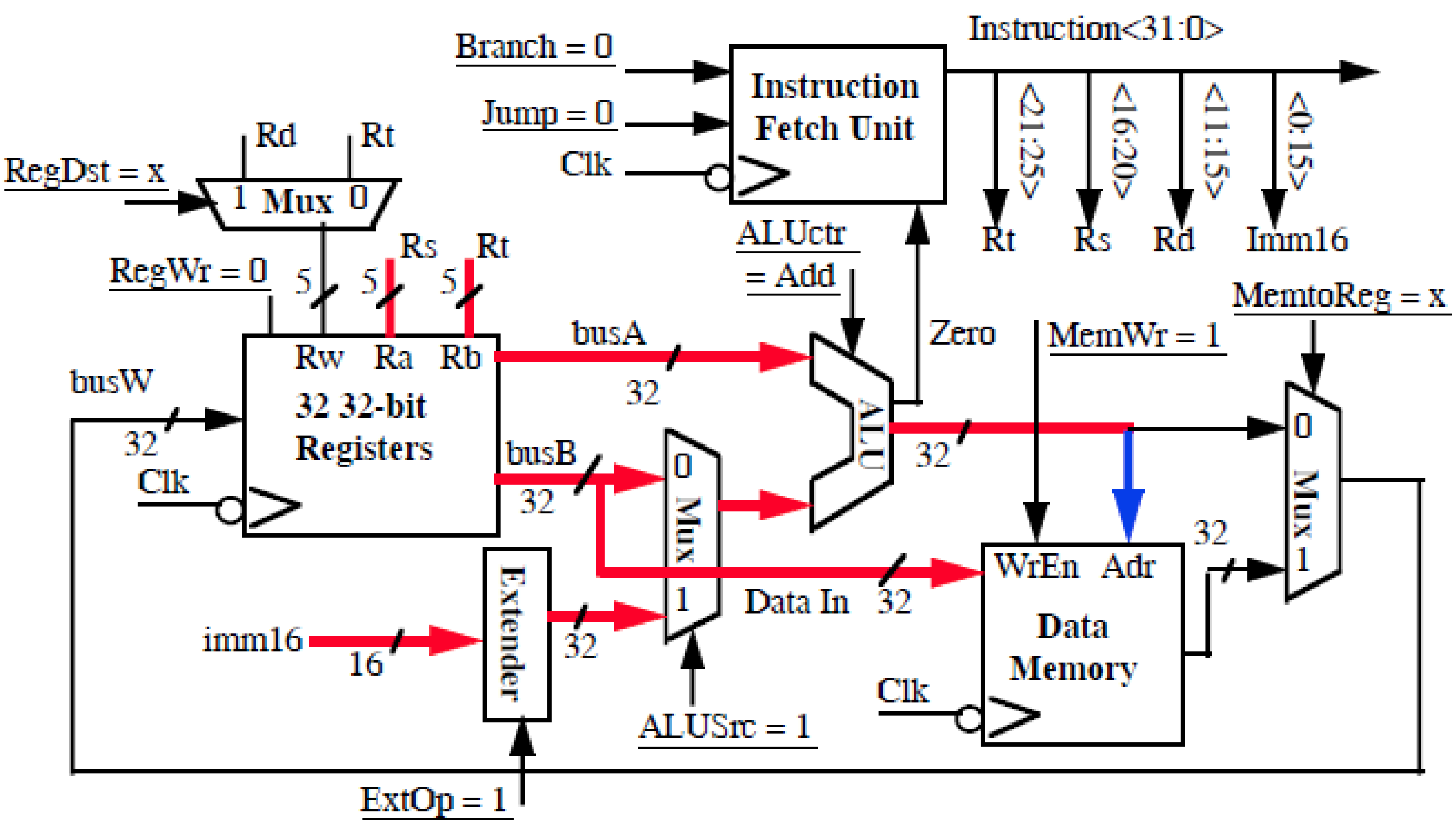
° R[rt] <- Data Memory {R[rs] + SignExt[imm16]}



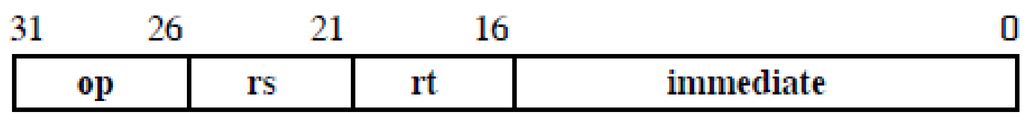
The Single Cycle Datapath during Store

31	26	21	16	0
op	rs	rt	immediate	

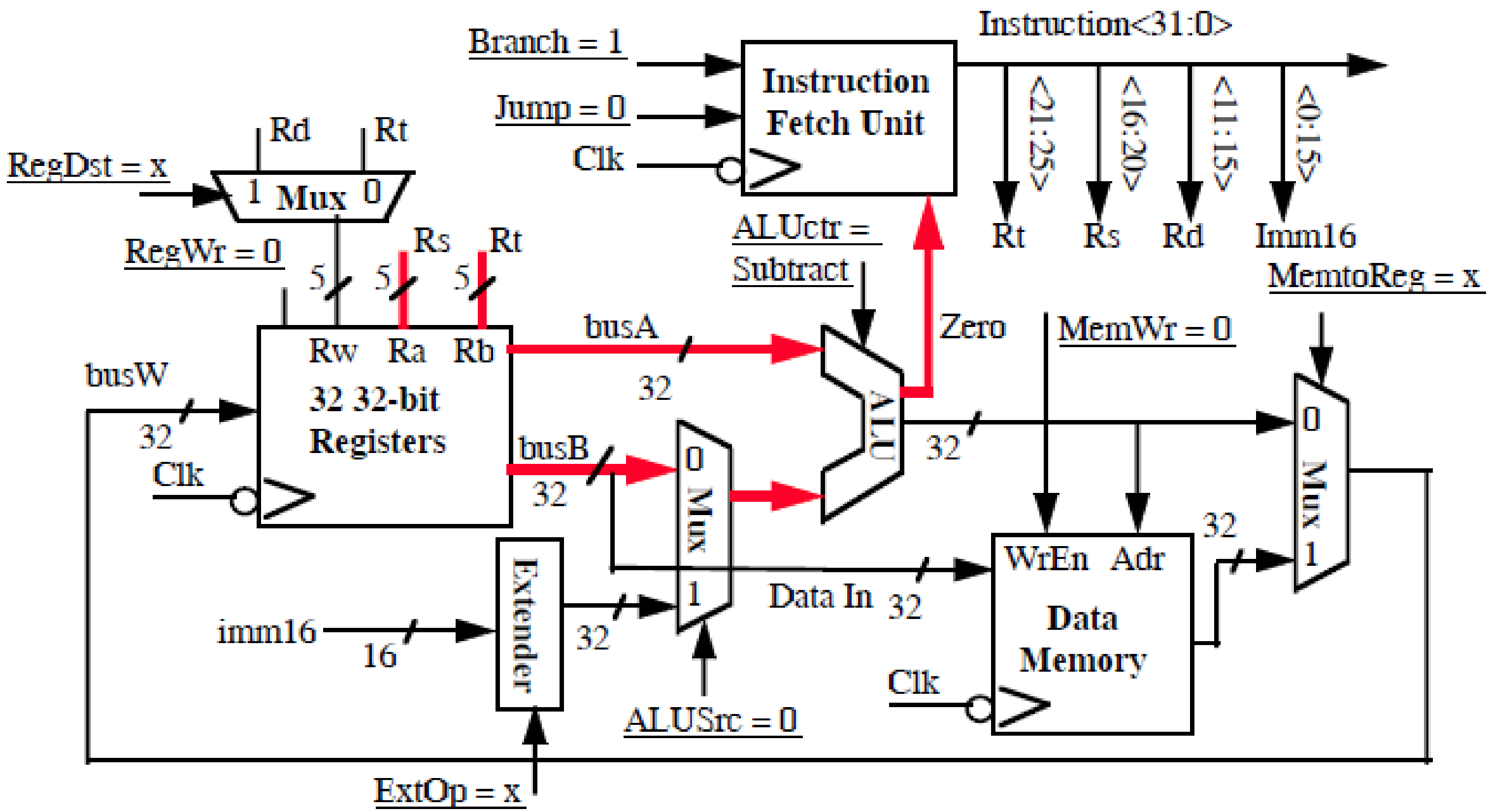
- Data Memory {R[rs] + SignExt[imm16]} <- R[rt]



The Single Cycle Datapath during Branch



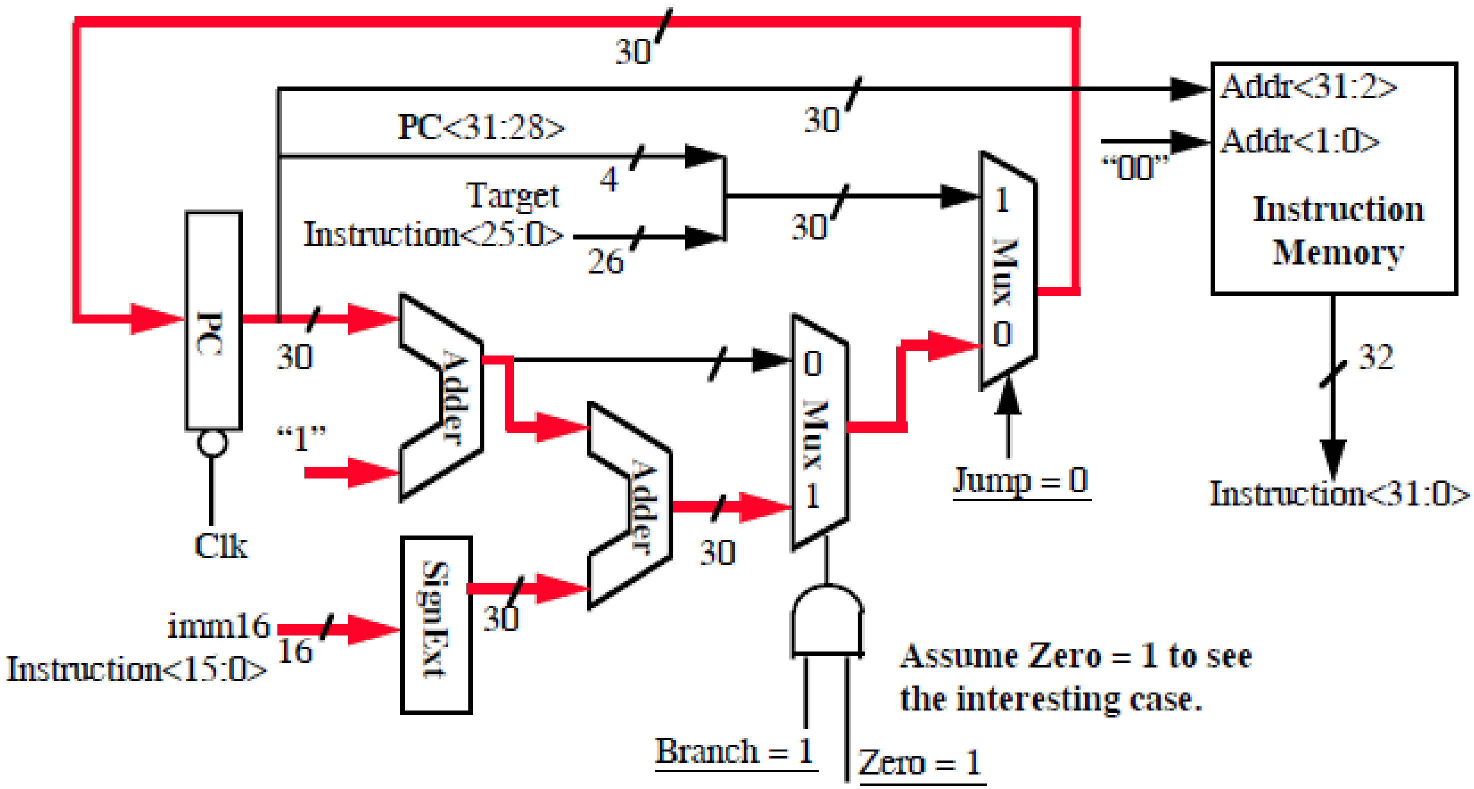
- if (R[rs] - R[rt] == 0) then Zero <- 1 ; else Zero <- 0



Instruction Fetch Unit at the End of Branch

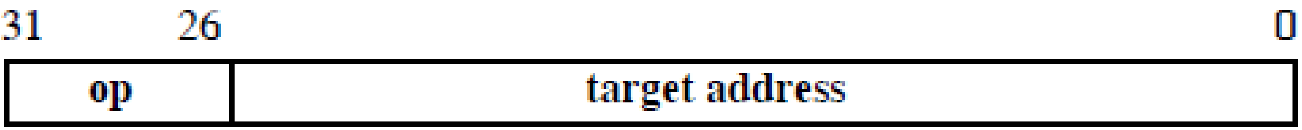
31 op 26	rs 21	rt 16	immediate 0

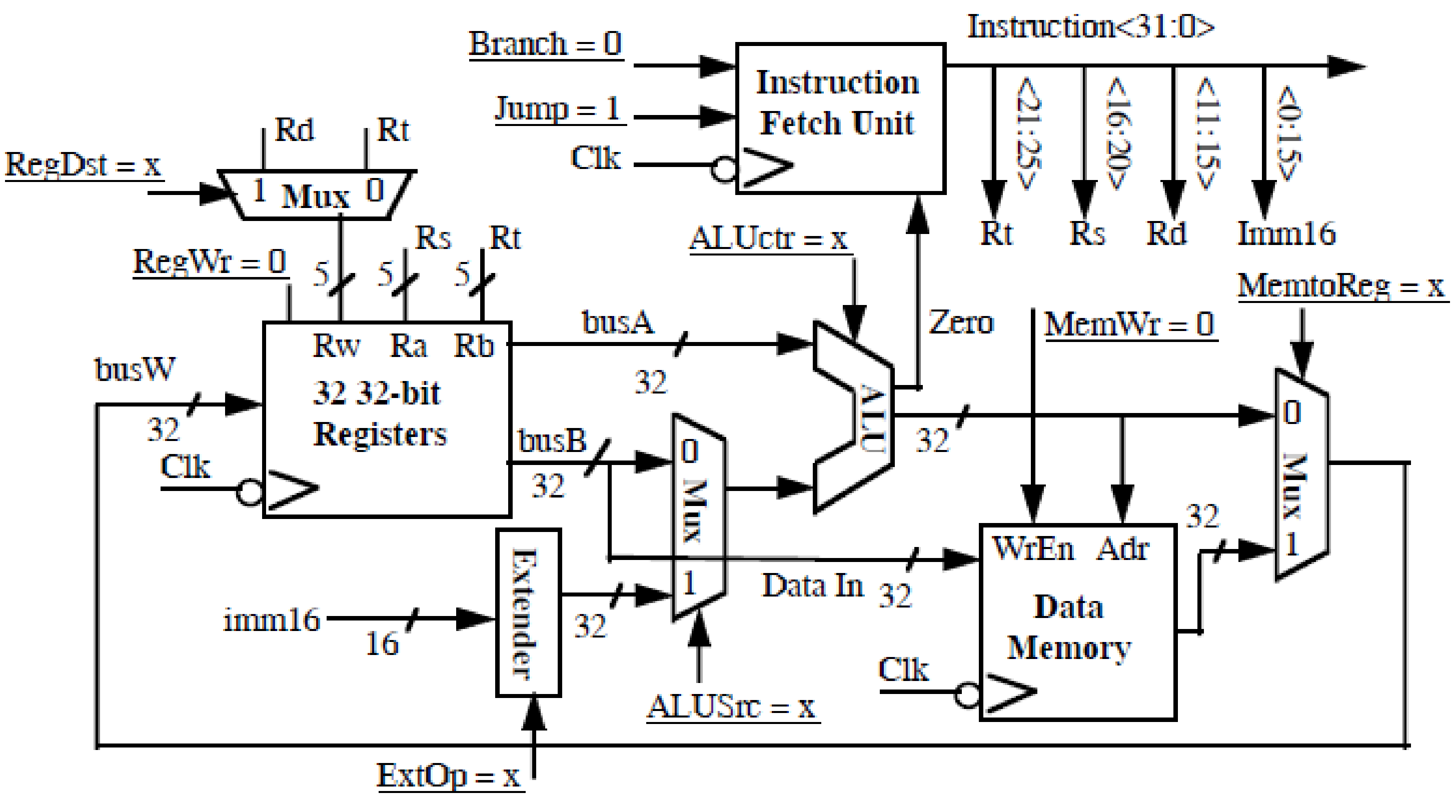
- if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4



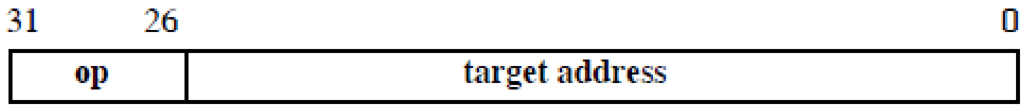
The Single Cycle Datapath during Jump

- Nothing to do! Make sure control signals are set correctly!

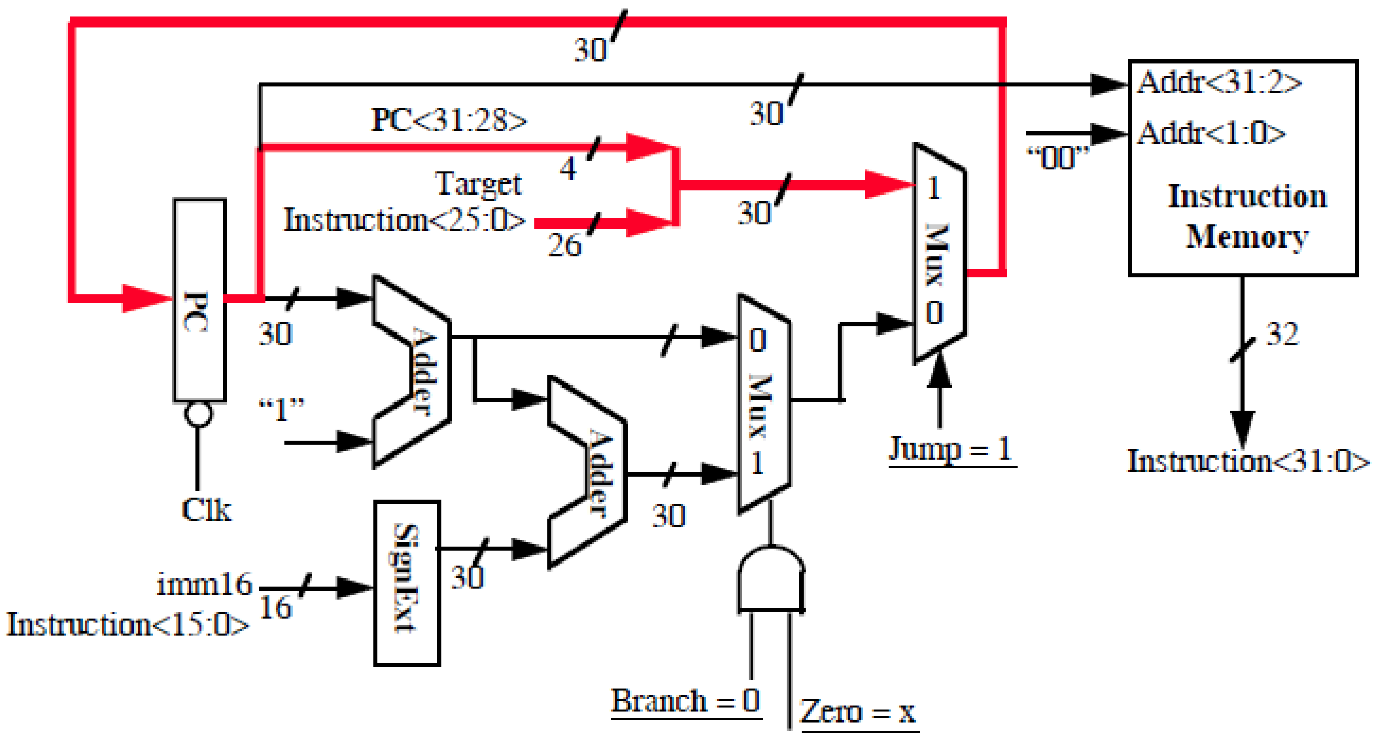




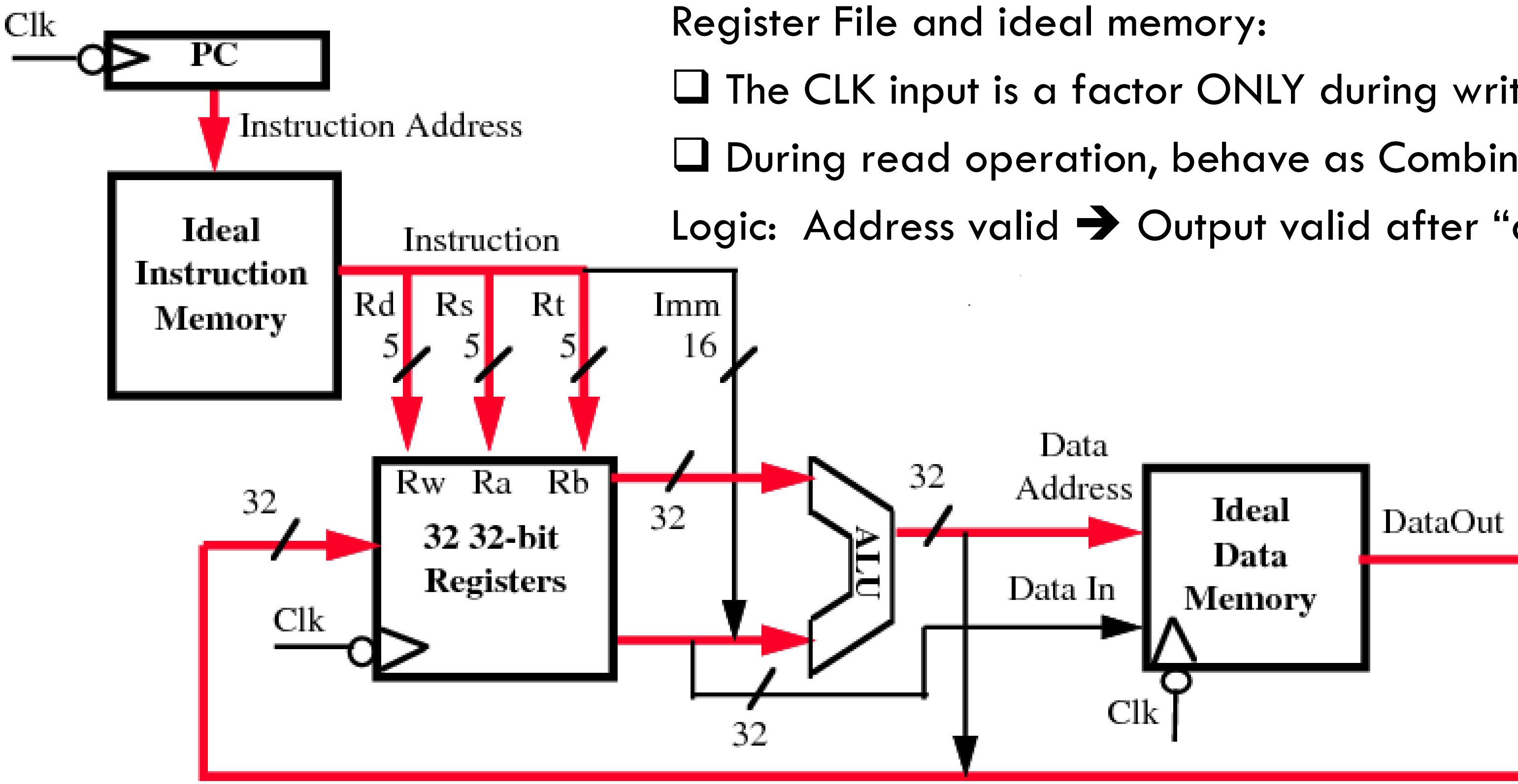
Instruction Fetch Unit at the End of Jump



- PC <- PC<31:29> concat target<25:0> concat "00"



An Abstract View of the Critical Path



Register File and ideal memory:

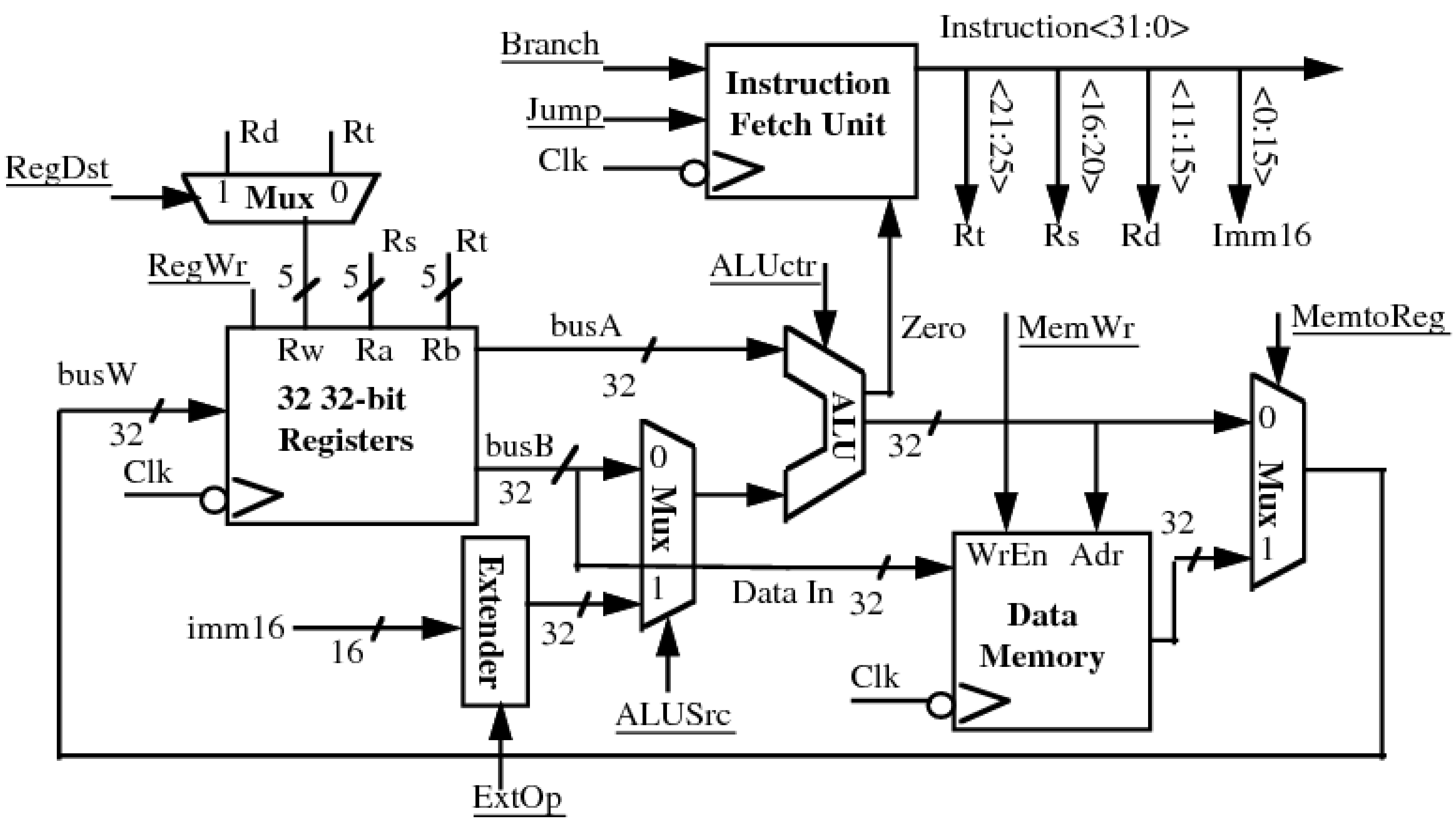
- The CLK input is a factor ONLY during write operation
- During read operation, behave as Combinational

Logic: Address valid ➔ Output valid after "access time"

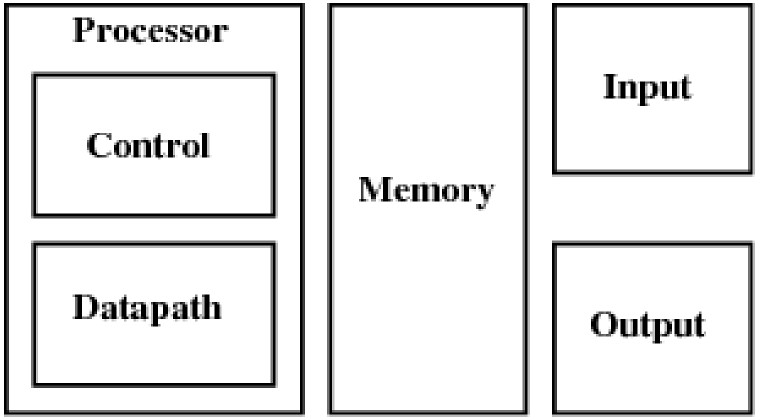
Critical Path (Load Operation) = PC's Clk-to-Q + Instruction Memory's Access Time + Register File's Access Time + ALU to Perform 32-bit Add + Data Memory Access Time + Setup Time for Register File Write

Putting it all together: A Single Cycle Datapath

We have everything except control signals (underline)



The Big Picture: Where are we Now?



- The Five Classic Components of a Computer
- Next Topic: Control Path Design

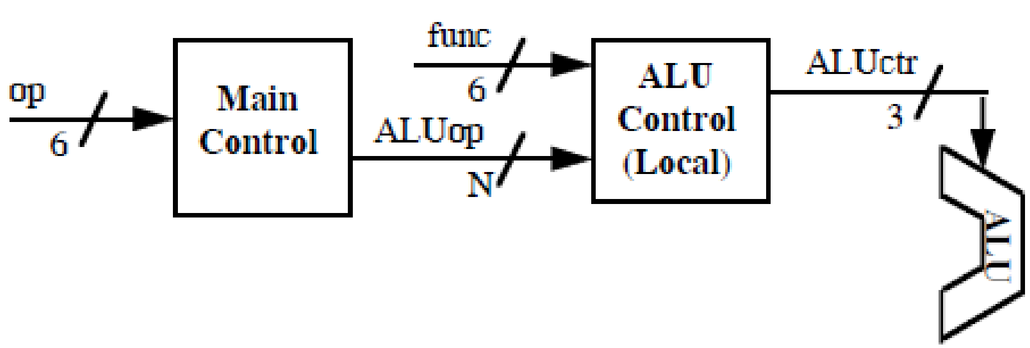
A Summary of Control Signals

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	0	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
Branch	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

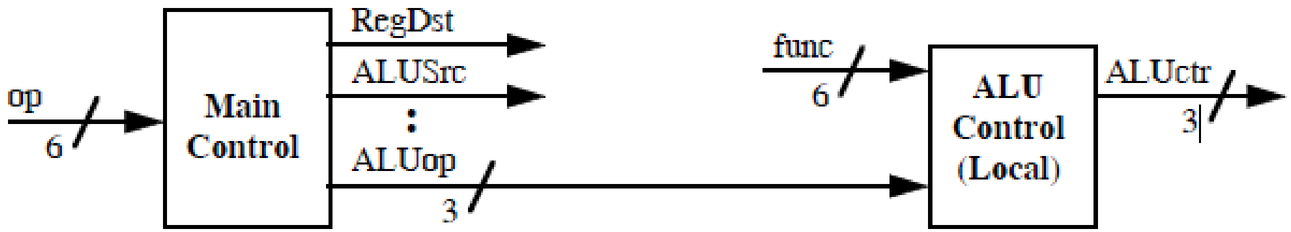
The Concept of Local Decoding

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 “Truth Table” for the Main Control

Key Idea: Two levels of Control logic.

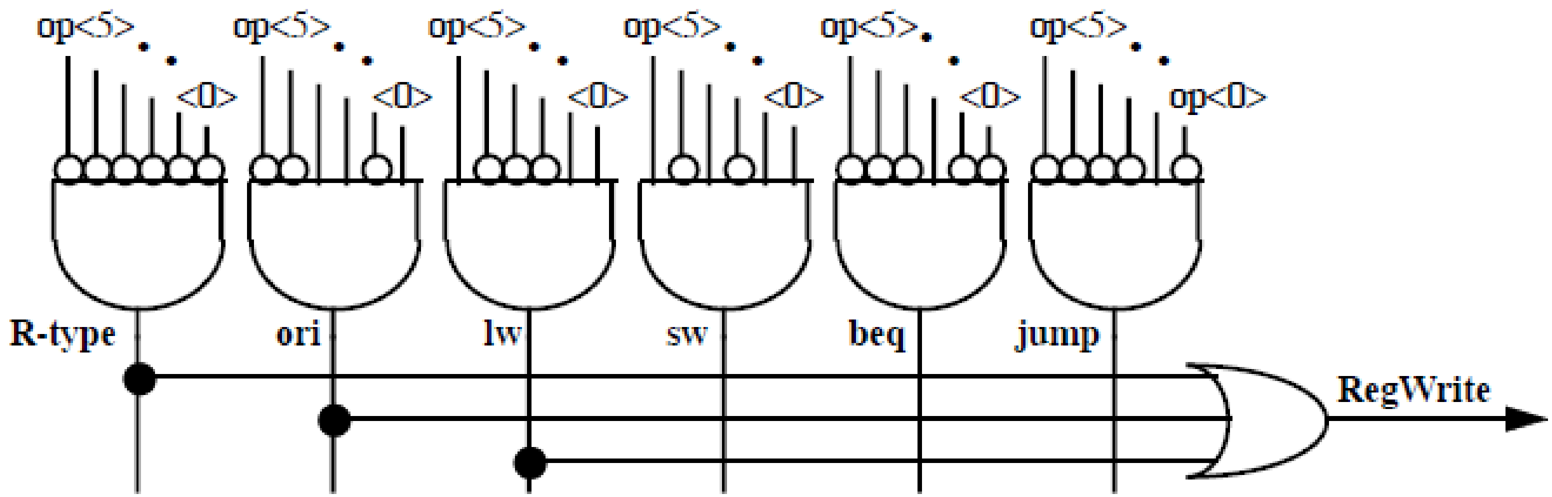


op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0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

Question: Can you write the truth table for the ALU control keeping in mind the ALU we designed in the class?

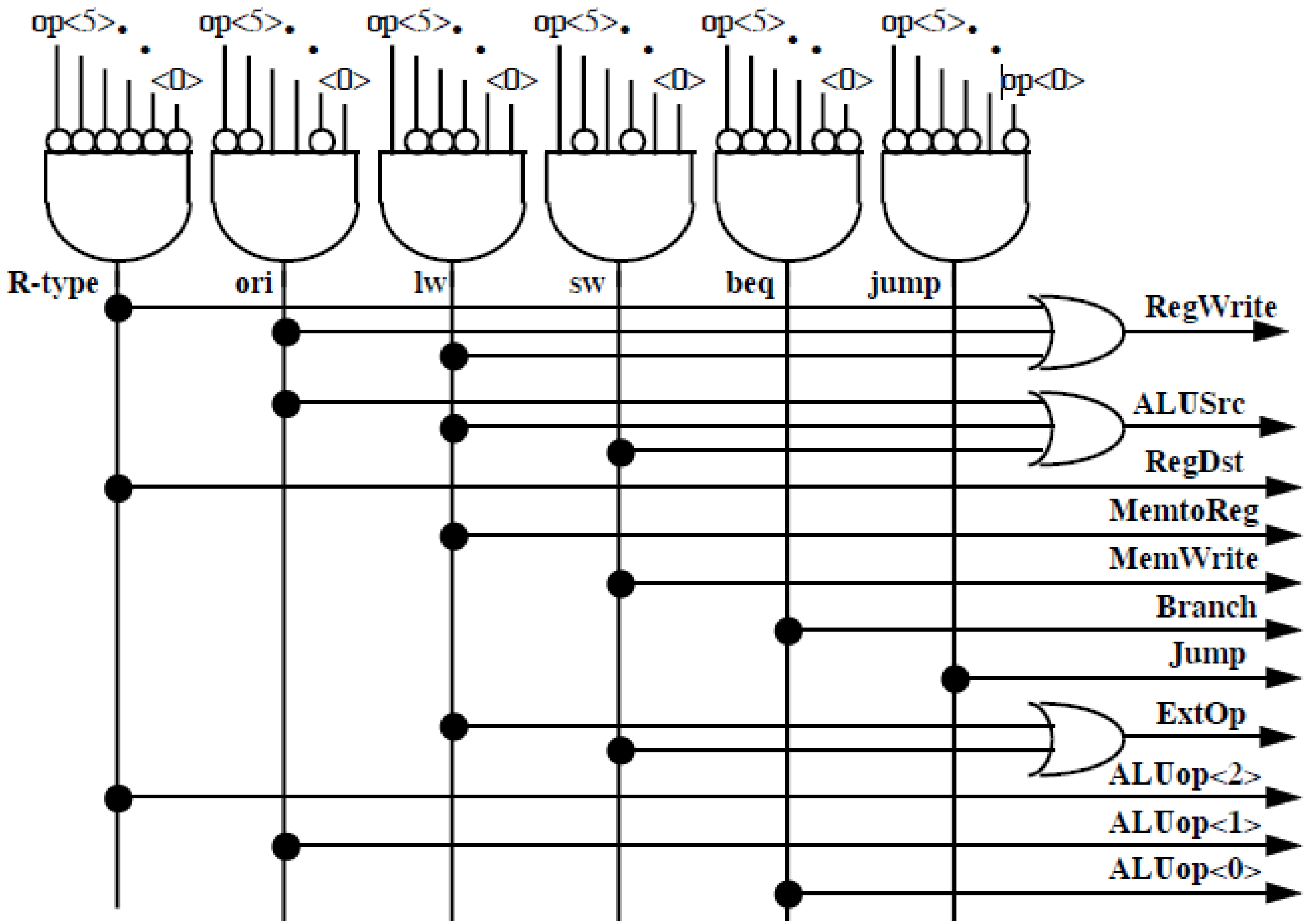
The “Truth Table” for RegWrite

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

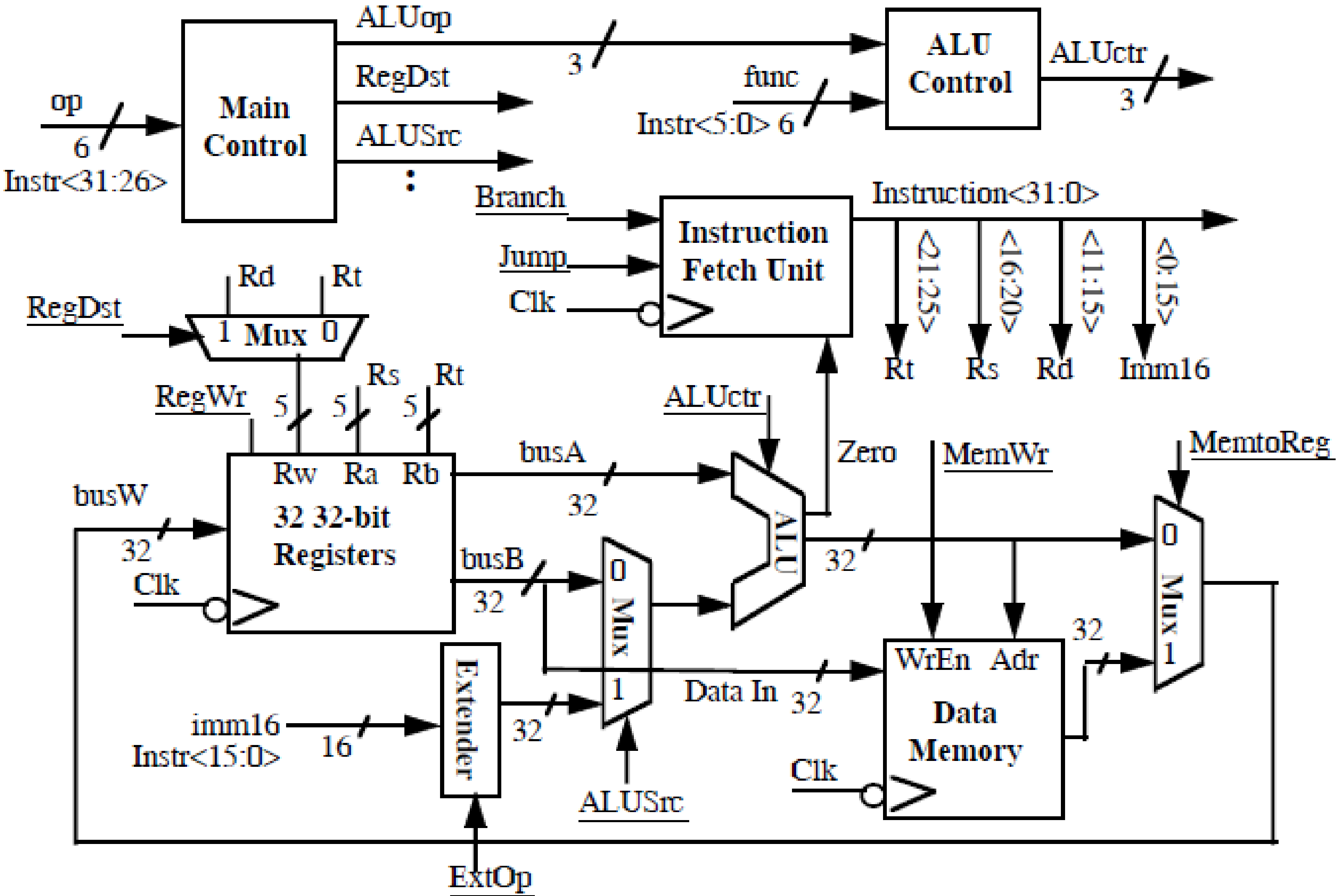


PLA Implementation of Main Control

Hmm! What is PLA?



Putting it All Together: A Single Cycle Processor



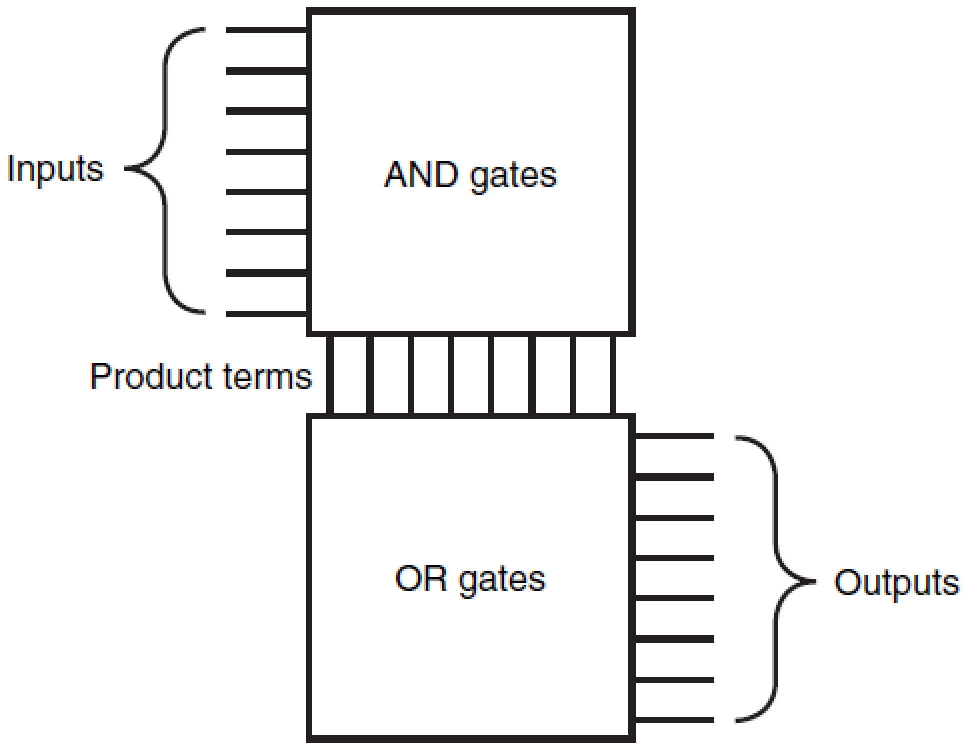
Drawback of this Single Cycle Processor

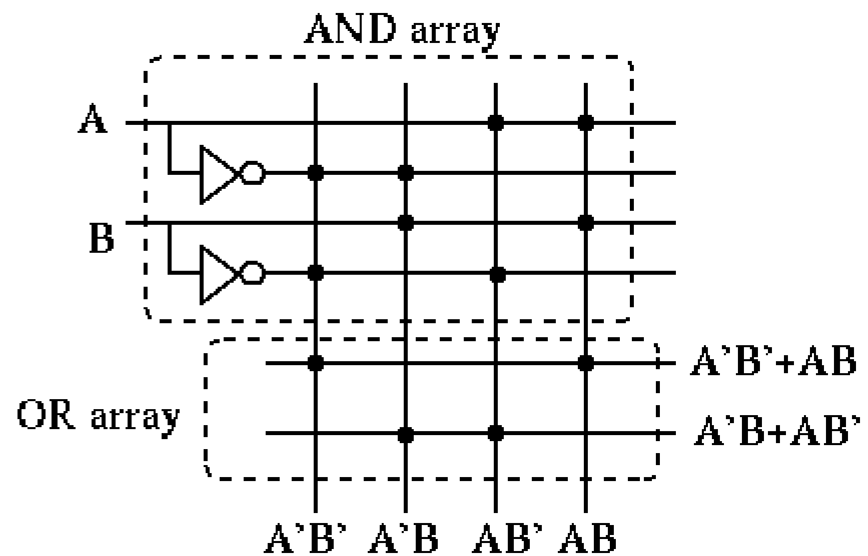
- Long cycle time:
  - Cycle time must be long enough for the load instruction:
    - PC's Clock -to-Q +
    - Instruction Memory Access Time +
    - Register File Access Time +
    - ALU Delay (address calculation) +
    - Data Memory Access Time +
    - Register File Setup Time
- Cycle time is much longer than needed for all other instructions

We are assuming Clock Skew is zero

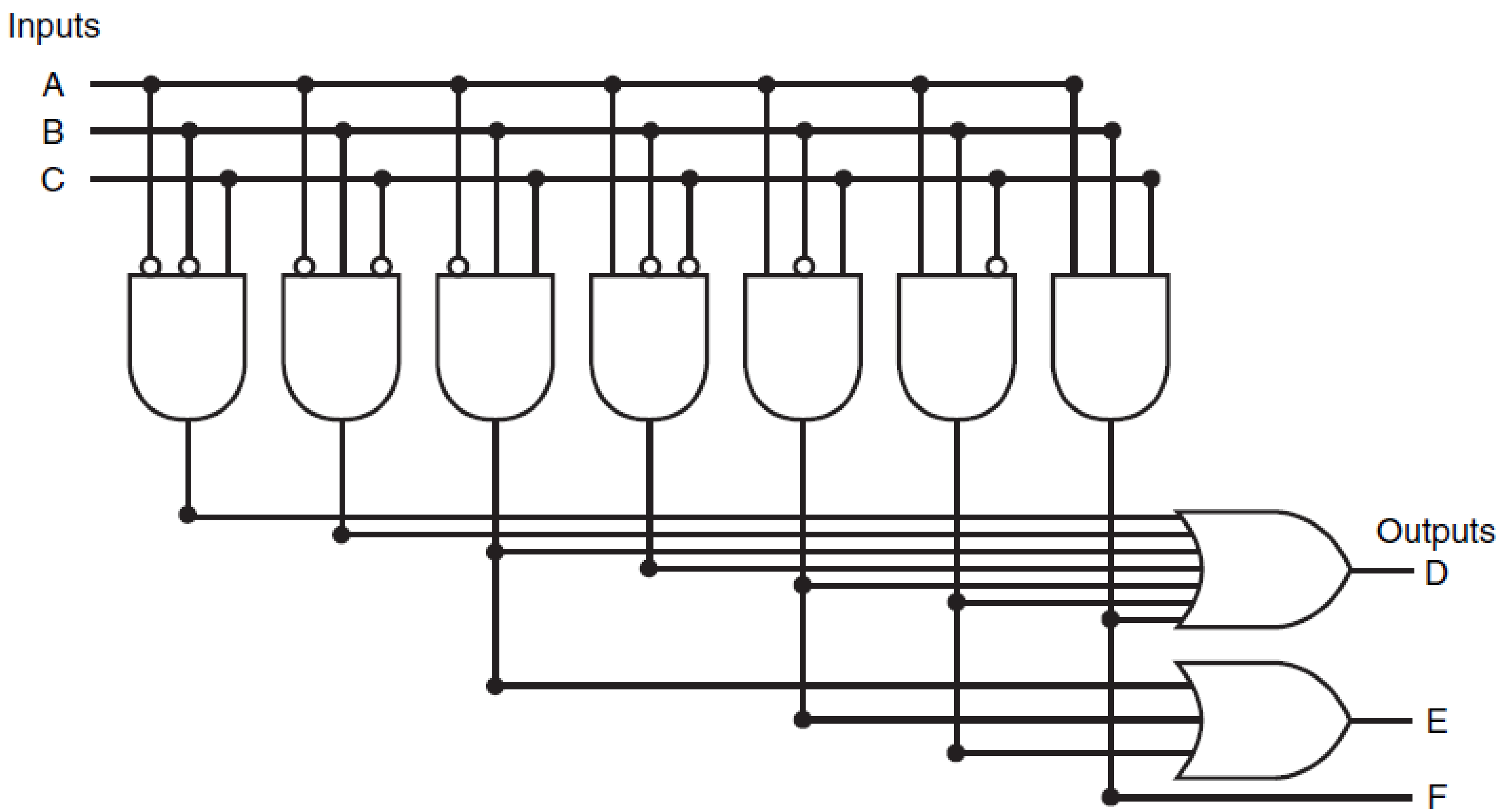
Programmable Logic Arrays

- PLAs can be used to realize combinational circuits





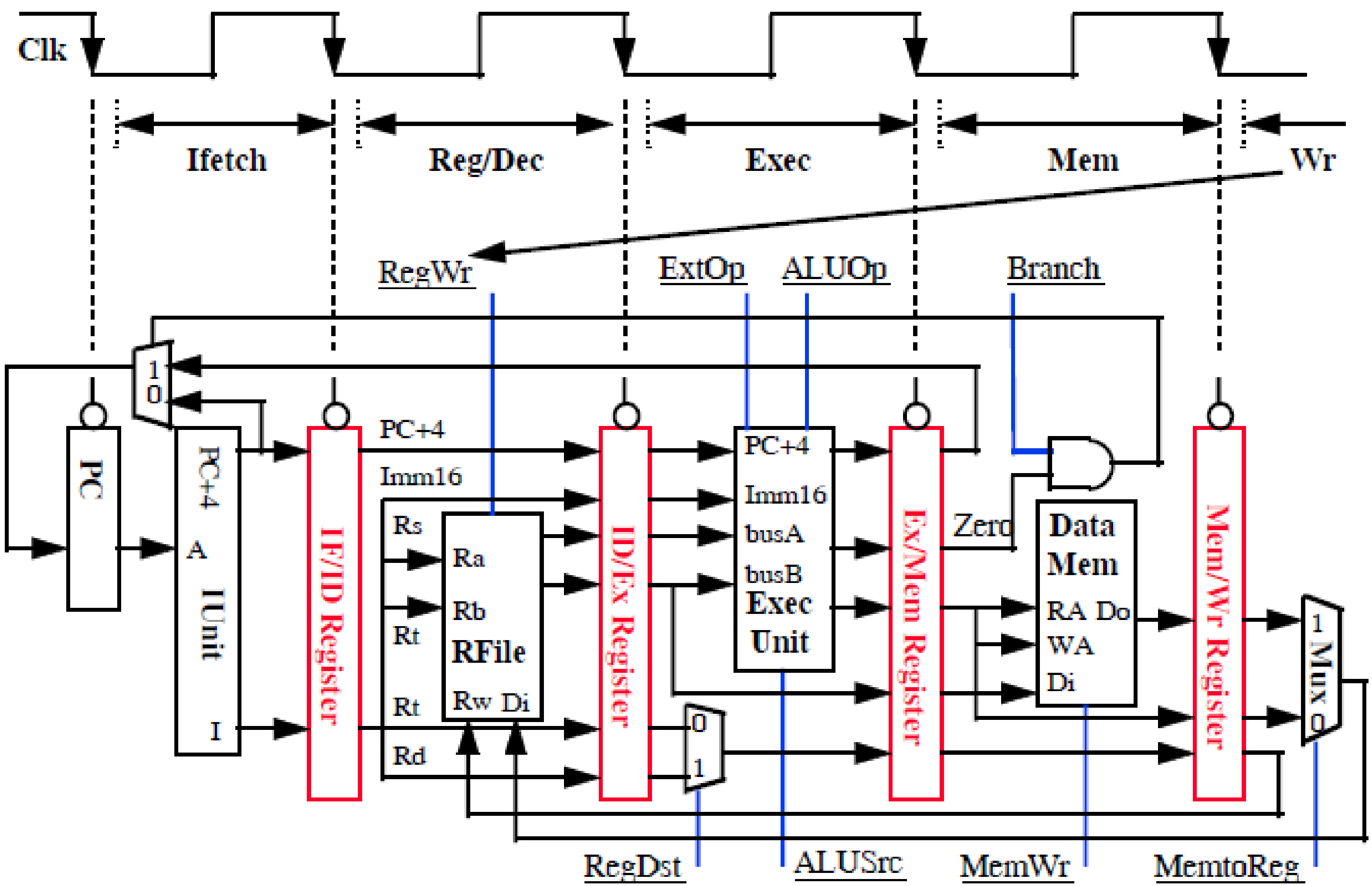
PLAs



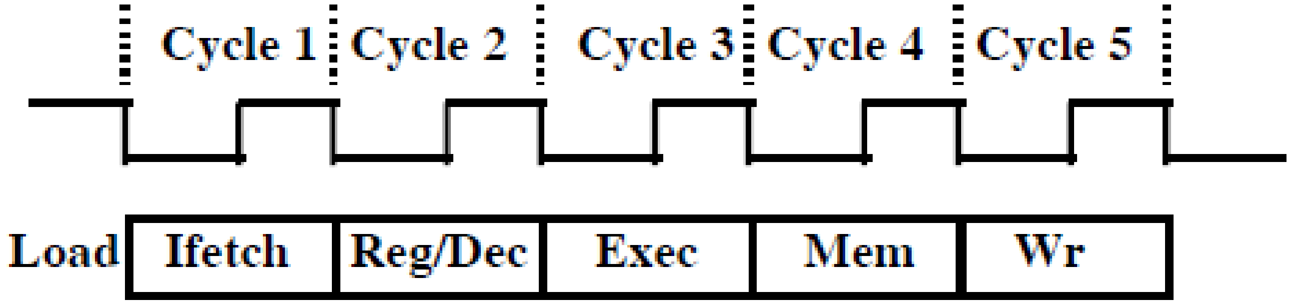
Acknowledgment: Almost all of these slides are based on Dave Patterson's CS152 Lecture Slides at UC, Berkeyley.

COMPUTER SYSTEMS ORGANIZATION

A Pipelined Datapath



The Five Stages of Load

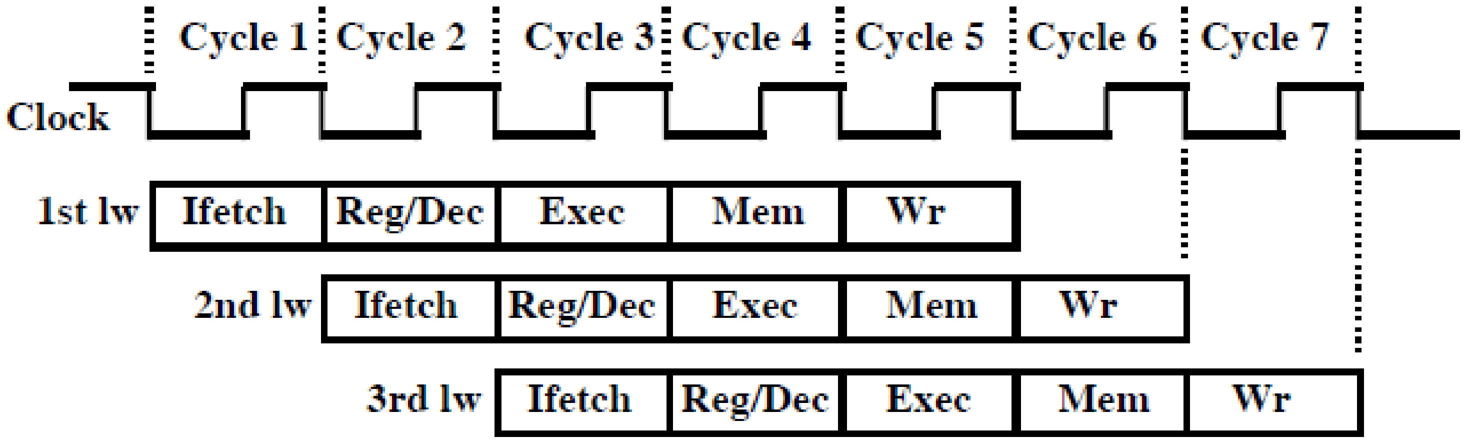


- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

Key Ideas Behind Pipelining

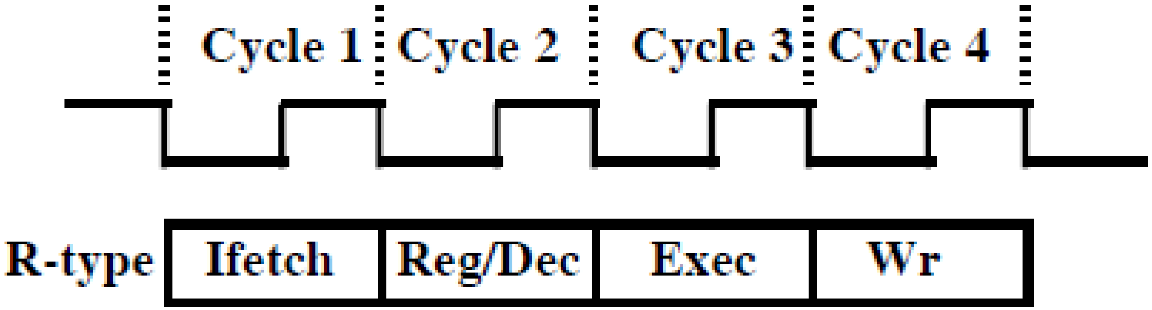
- The load instruction has 5 stages:
  - Five independent functional units to work on each stage
  - Each functional unit is used only once
- The 2nd load can start as soon as the 1st finishes its Ifetch stage
- Each load still takes five cycles to complete
- The throughput, however, is much higher

Pipelining the Load Instruction



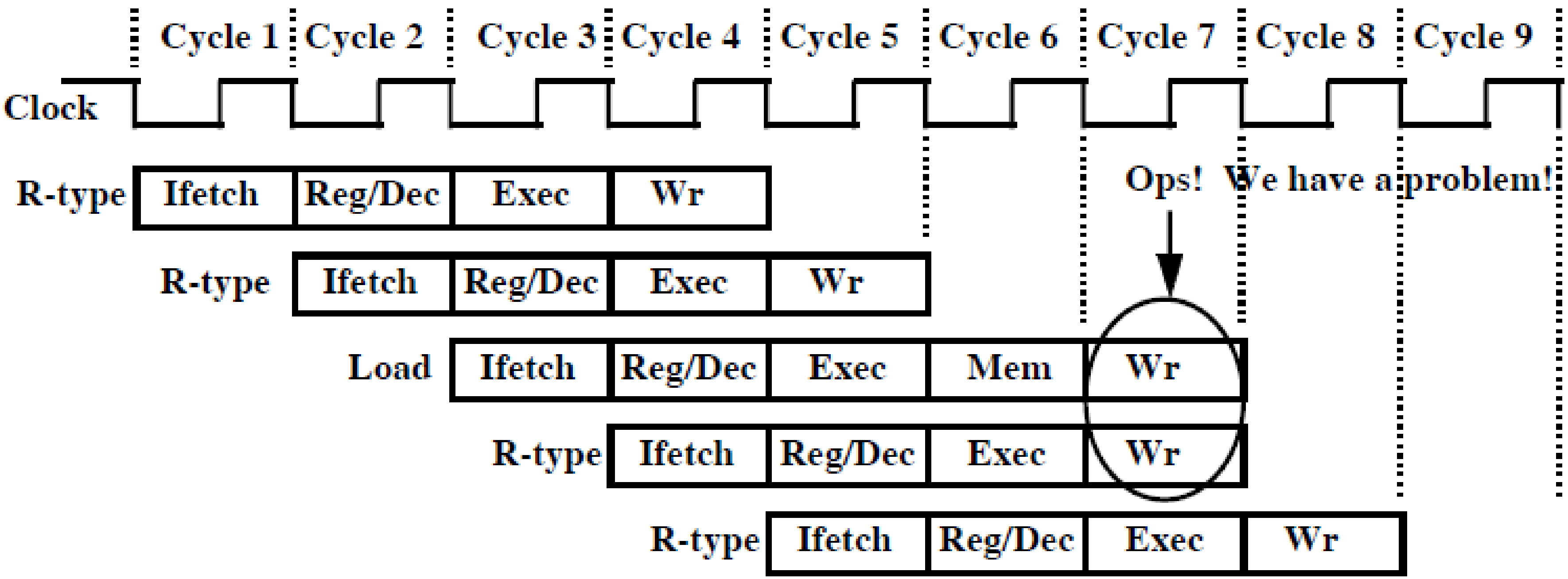
- The five independent functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File's Read ports (bus A and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the Mem stage
  - Register File's Write port (bus W) for the Wr stage
- One instruction enters the pipeline every cycle
  - One instruction comes out of the pipeline (complete) every cycle
  - The "Effective" Cycles per Instruction (CPI) is 1

The Four Stages of R-type



- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: ALU operates on the two register operands
- Wr: Write the ALU output back to the register file

Pipelining the R-type and Load Instruction



- We have a problem:
  - Two instructions try to write to the register file at the same time!

Structural Hazard: Two instructions require access to the same functional unit.

Important Observation

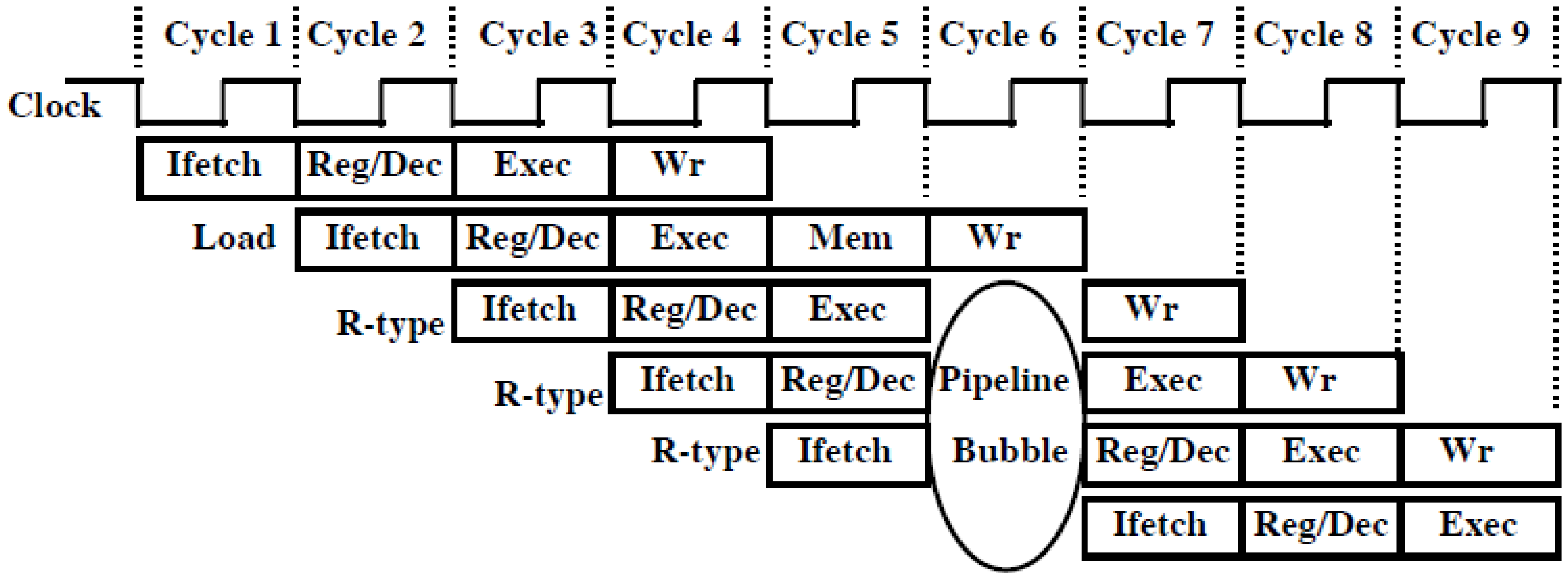
- Each functional unit can only be used once per instruction
- Each functional unit must be used at the same stage for all instructions:
  - Load uses Register File's Write Port during its 5th stage

	1	2	3	4	5
Load	Ifetch	Reg/Dec	Exec	Mem	Wr

  - R-type uses Register File's Write Port during its 4th stage

	1	2	3	4
R-type	Ifetch	Reg/Dec	Exec	Wr

Solution 1: Insert "Bubble" into the Pipeline

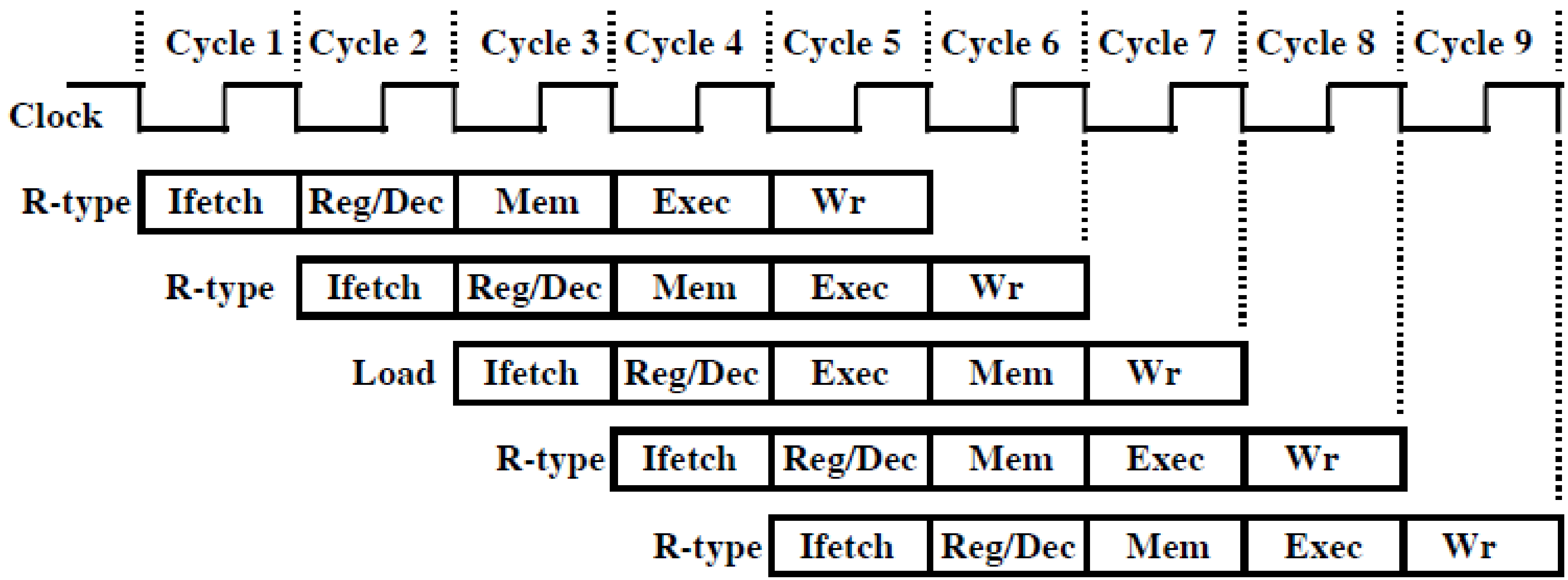


- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex
- No instruction is completed during Cycle 5:
  - The "Effective" CPI for load is 2

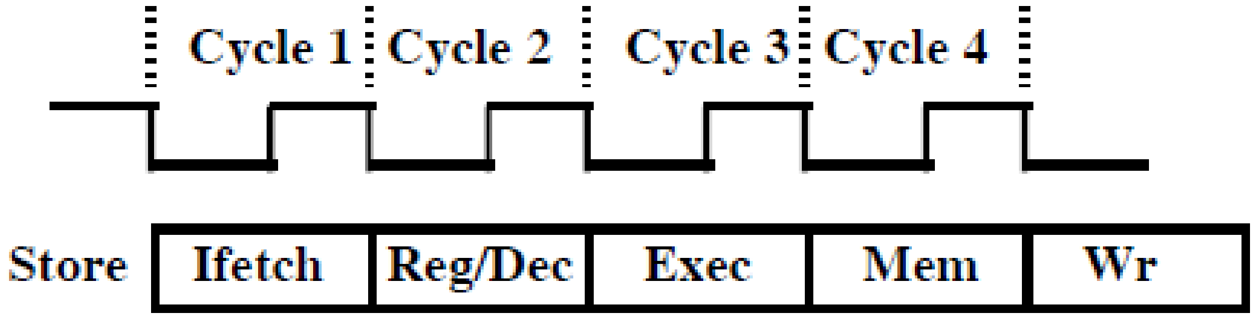
Solution 2: Delay R-type's Write by One Cycle

- Delay R-type's register write by one cycle:
  - Now R-type instructions also use Reg File's write port at Stage 5
  - Mem stage is a NOOP stage: nothing is being done

	1	2	3	4	5
R-type	Ifetch	Reg/Dec	Exec	Mem	Wr

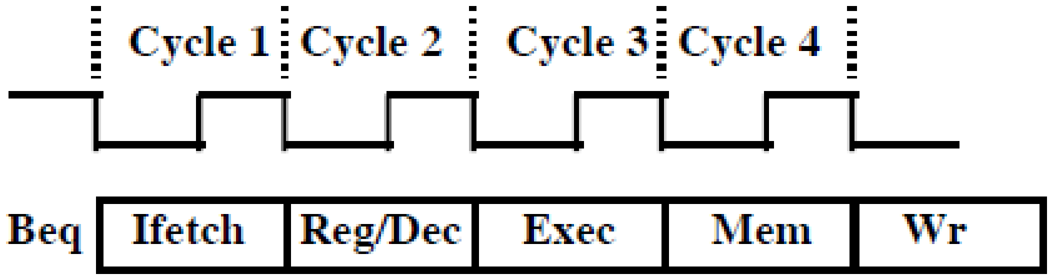


The Four Stages of Store



- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

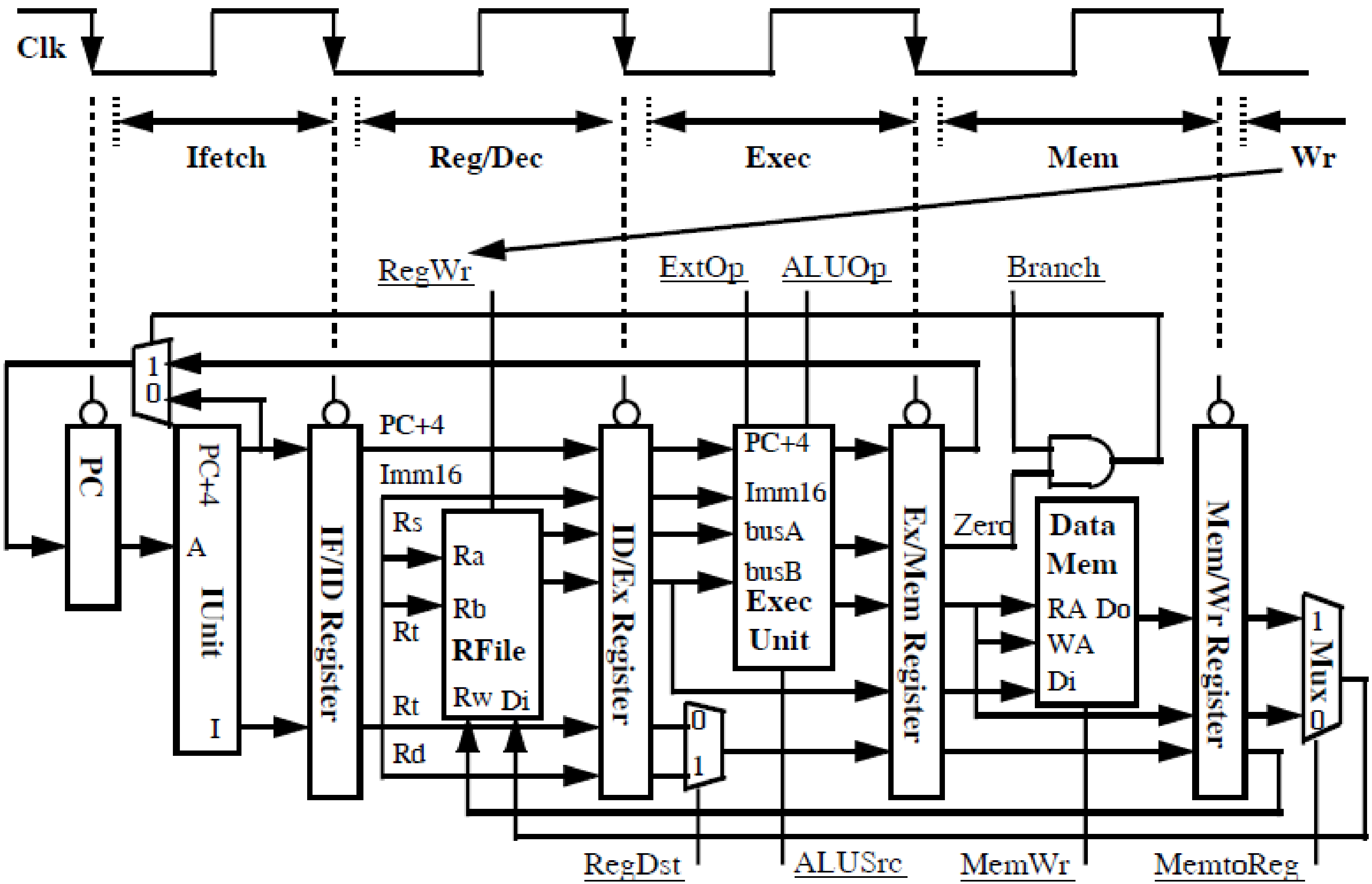
The Four Stages of Beq



- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: ALU compares the two register operands
  - Adder calculates the branch target address
- Mem: If the registers we compared in the Exec stage are the same,
  - Write the branch target address into the PC

Hey, this happens in second cycle in our Multi-Cycle CPU Design

A Pipelined Datapath

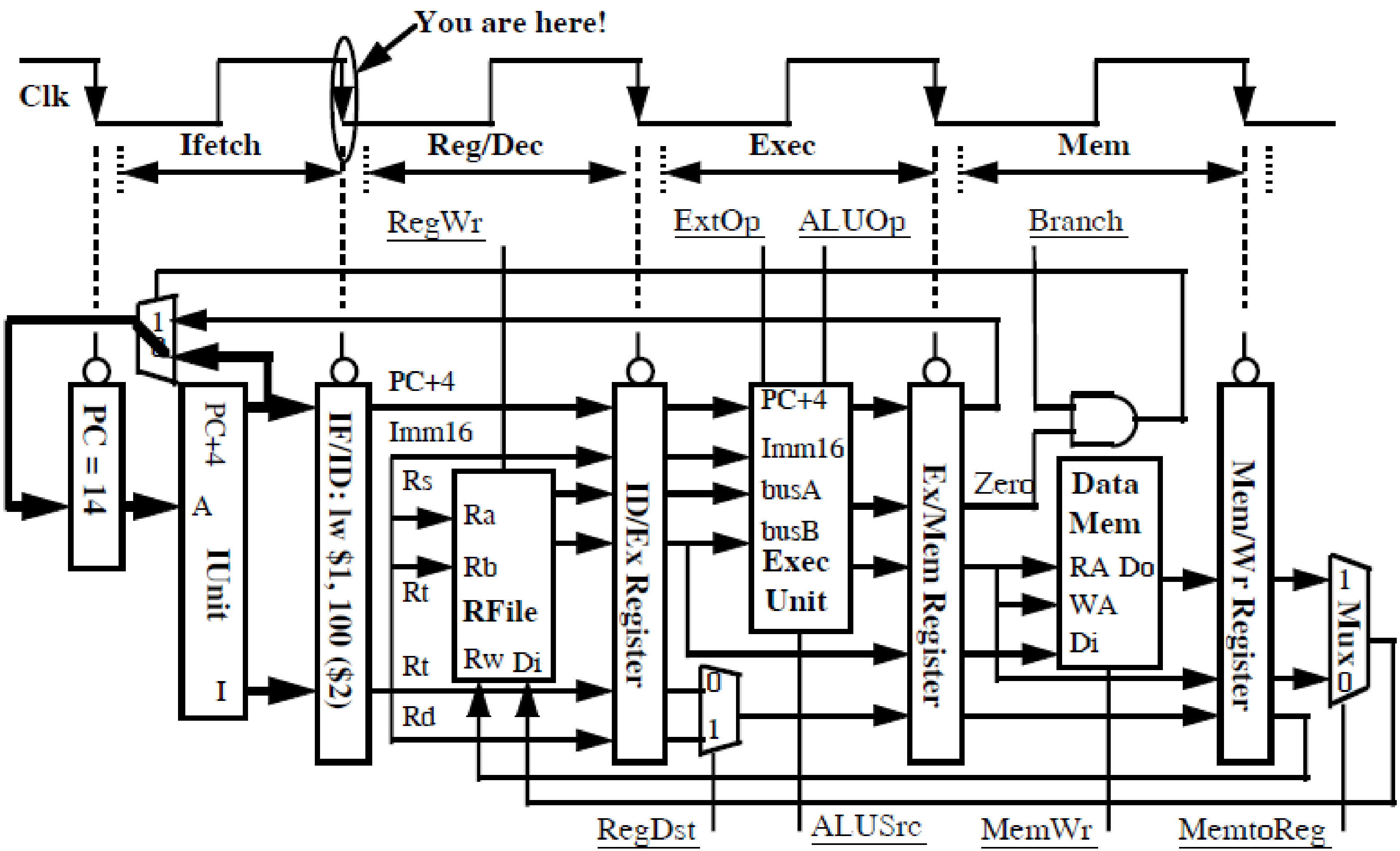


Fields of IF/ID Register:

1. 32-bits to store instruction
2. 32-bits to store PC+4

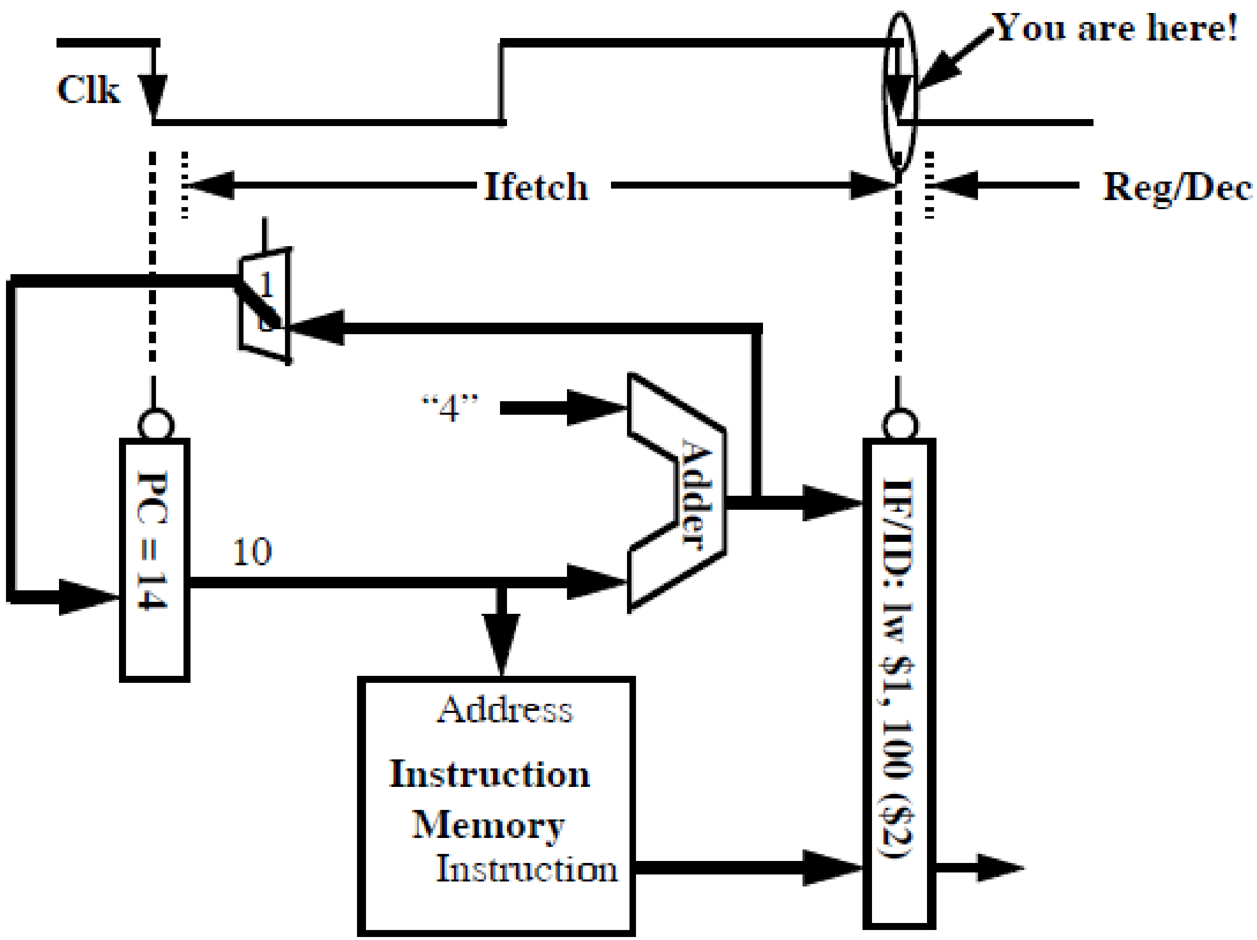
The Instruction Fetch Stage

- Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]



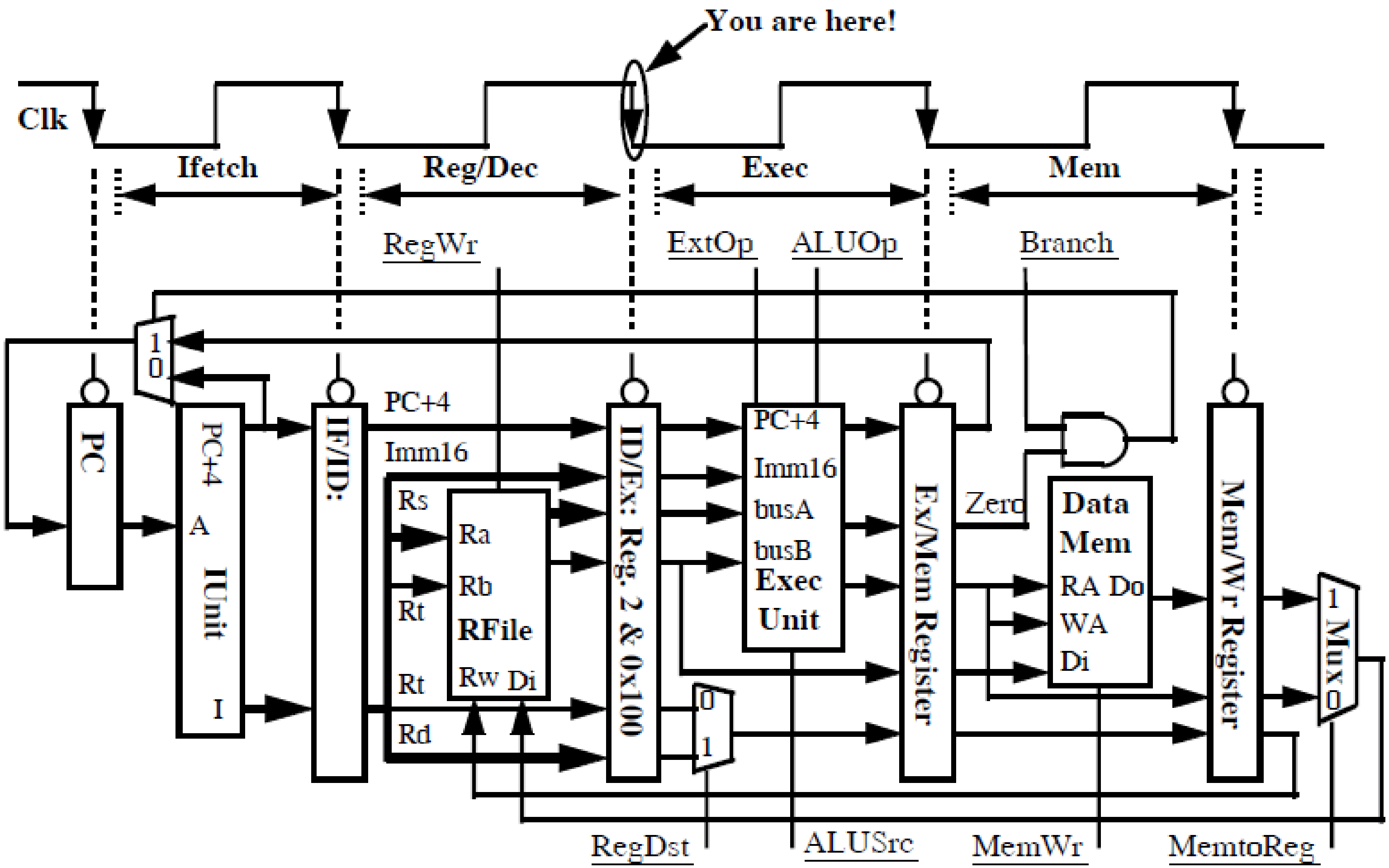
A Detail View of the Instruction Unit

- Location 10: lw $1, 0x100($2)



The Decode / Register Fetch Stage

- Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]



Fields of ID/Ex Register:

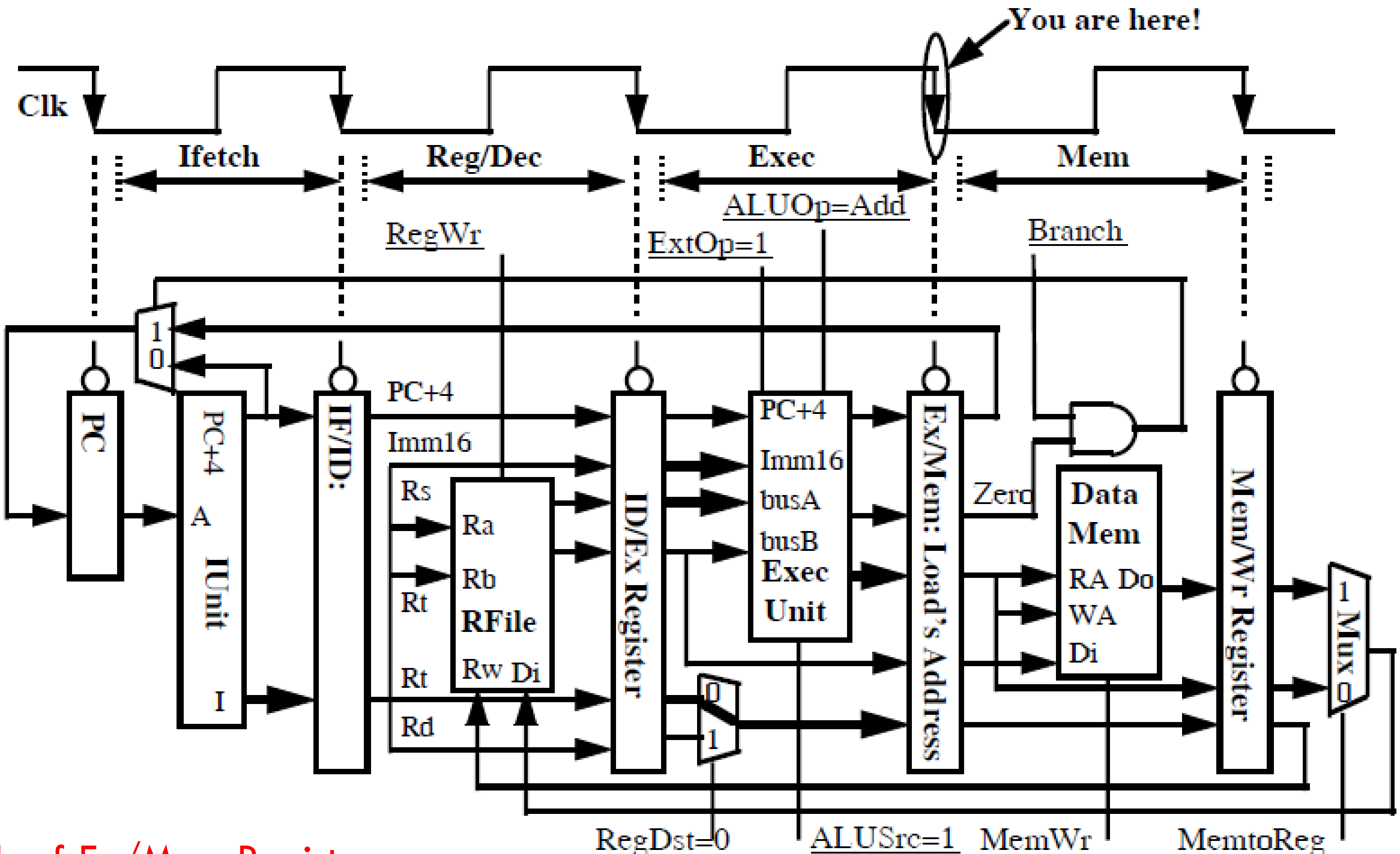
1. 32-bits for PC+4
2. 16-bits for Imm16
3. 32-bits for M[Rs]

Fields of ID/Ex Register (continued):

4. 32-bits for M[Rt]
5. 5-bits for Rt
6. 5-bits for Rd

Load's Address Calculation Stage

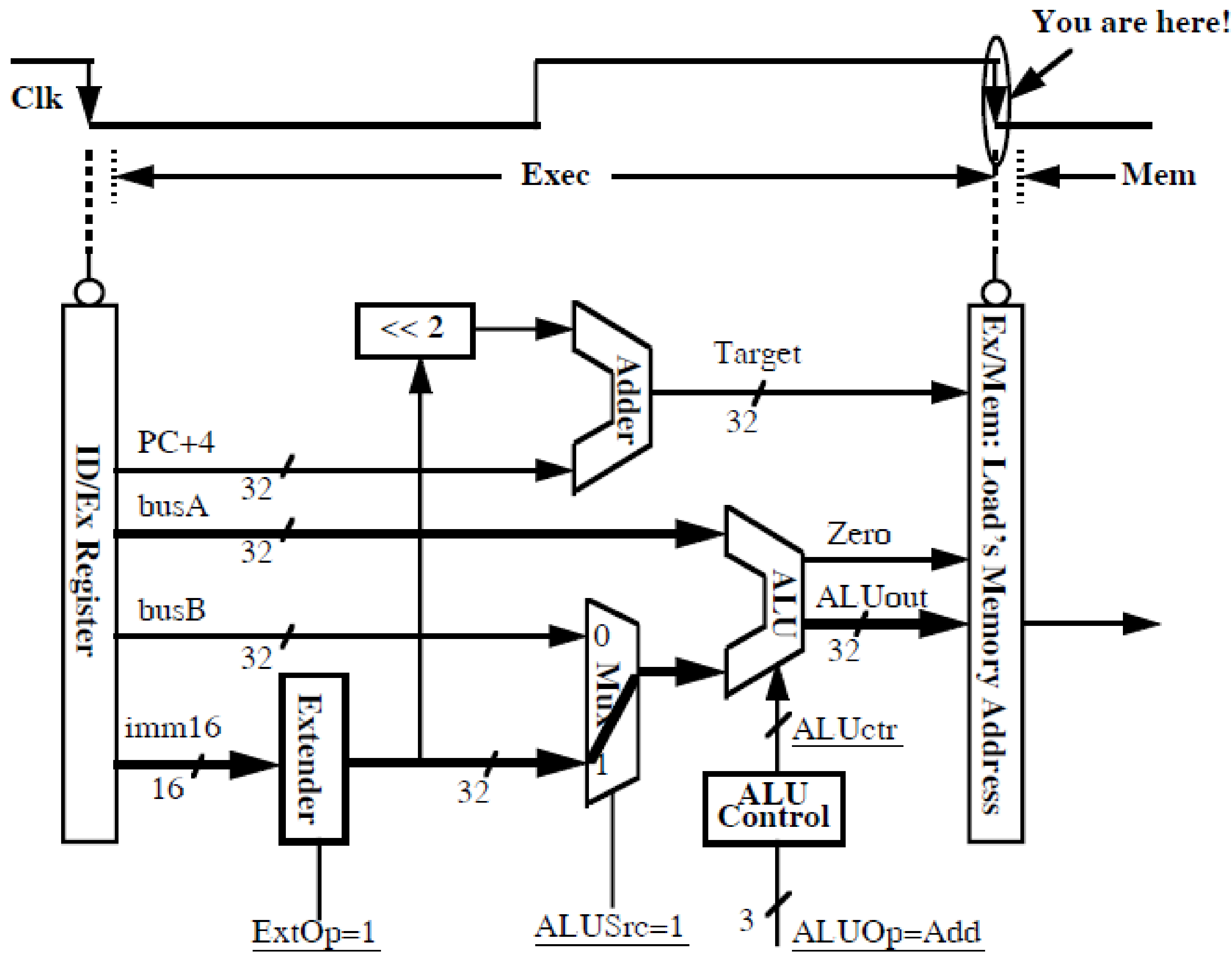
- Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]



Fields of Ex/Mem Register:

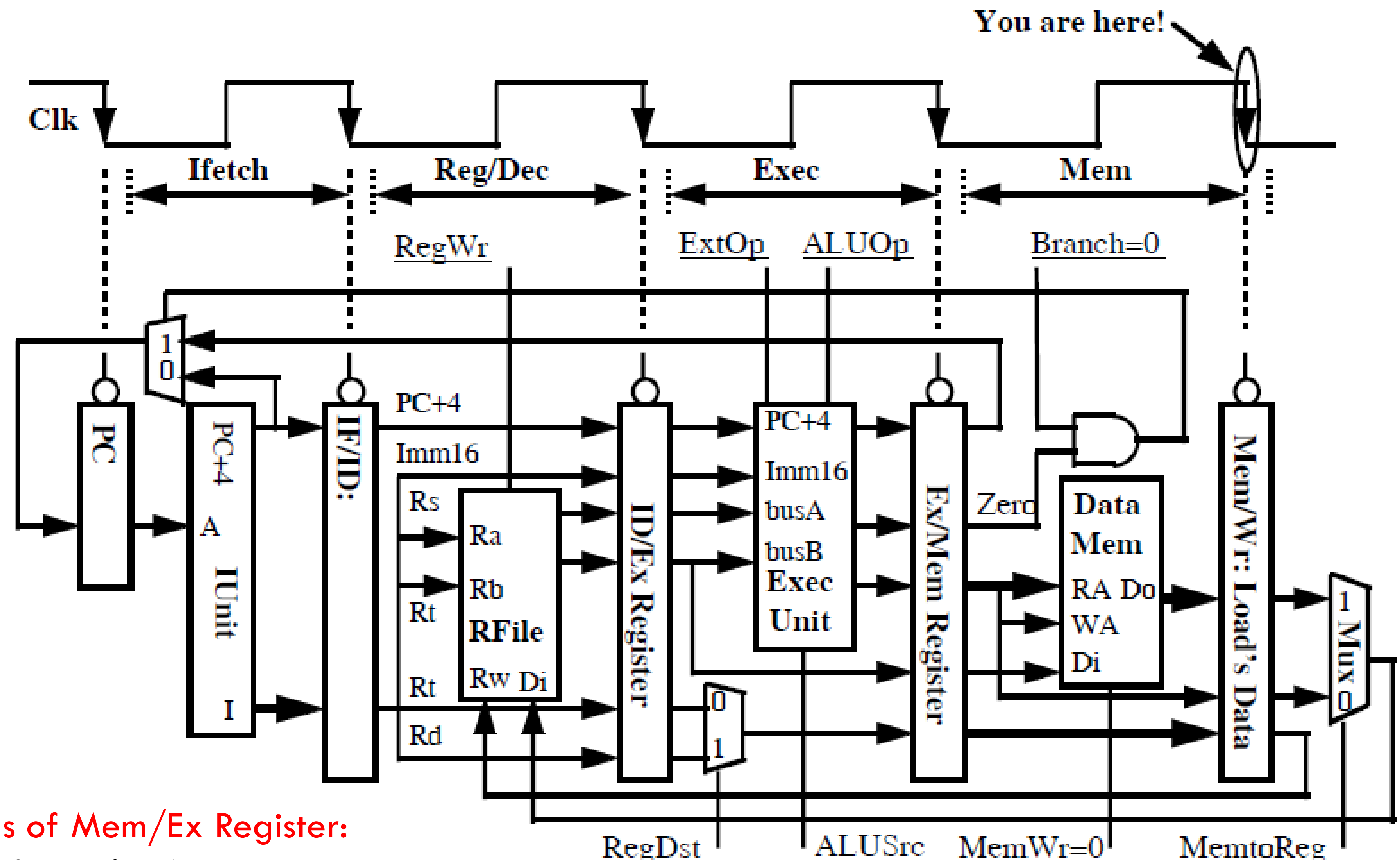
1. 32-bits for Branch Target Address
2. 32-bits for ALUout
3. 1-bit for Zero Flag
4. 32-bits for Mem[Rt]
5. 5-bits for RegDest: Rt or Rd

A Detail View of the Execution Unit



Load's Memory Access Stage

- Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]

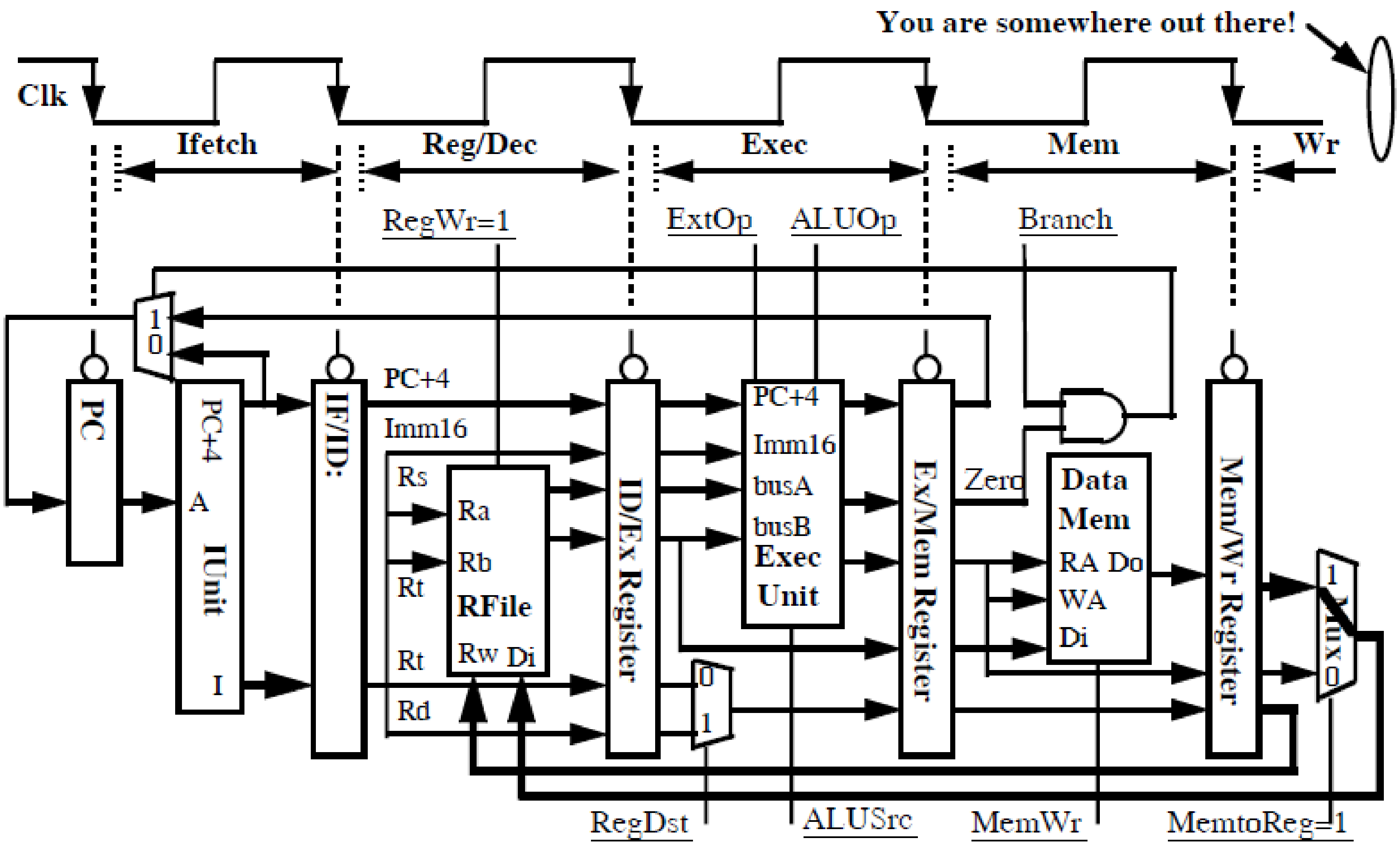


Fields of Mem/Ex Register:

1. 32-bits for ALUout
2. 32-bits for Data Memory
3. 5-bits for RegDest: Rt or Rd

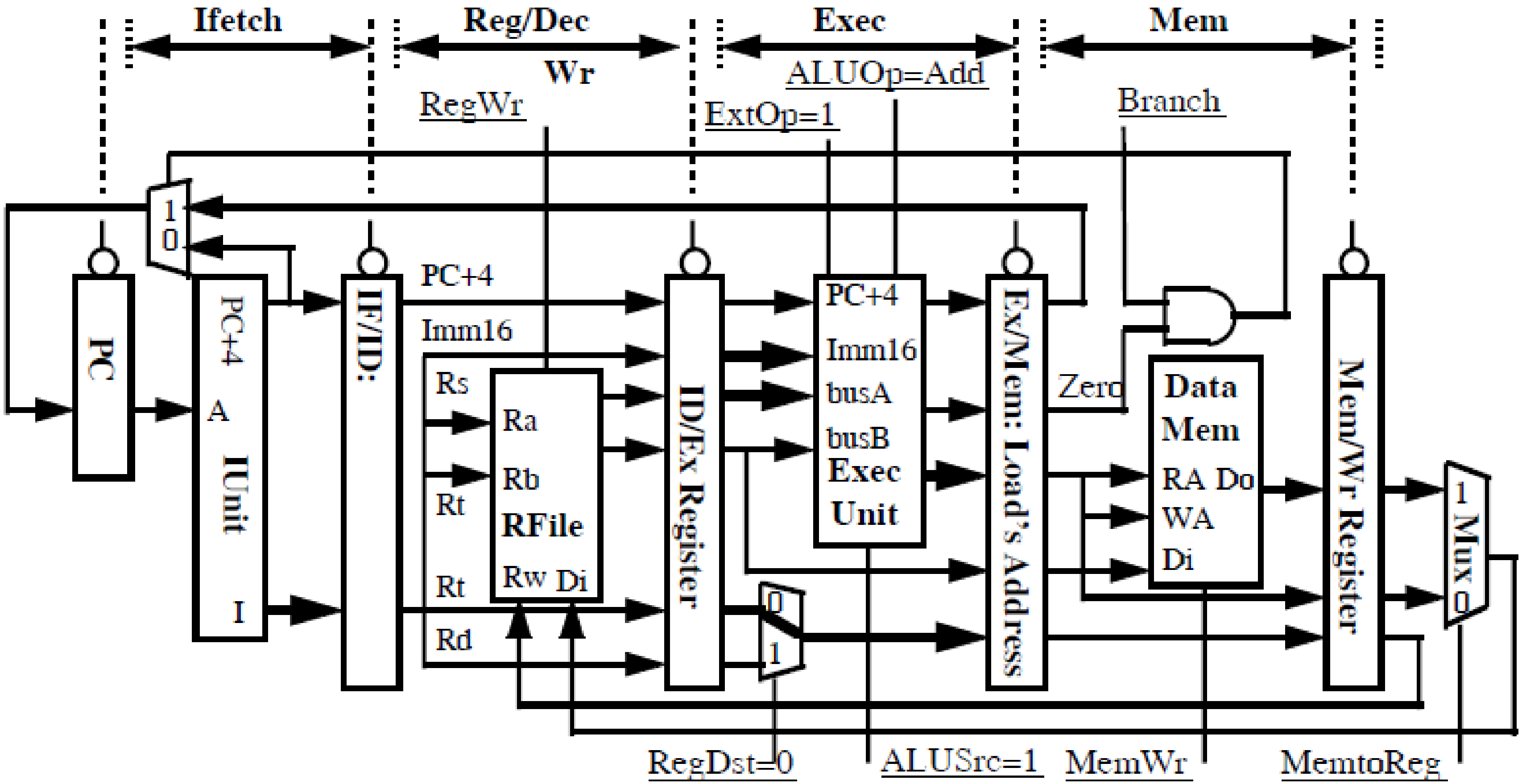
Load's Write Back Stage

- Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]



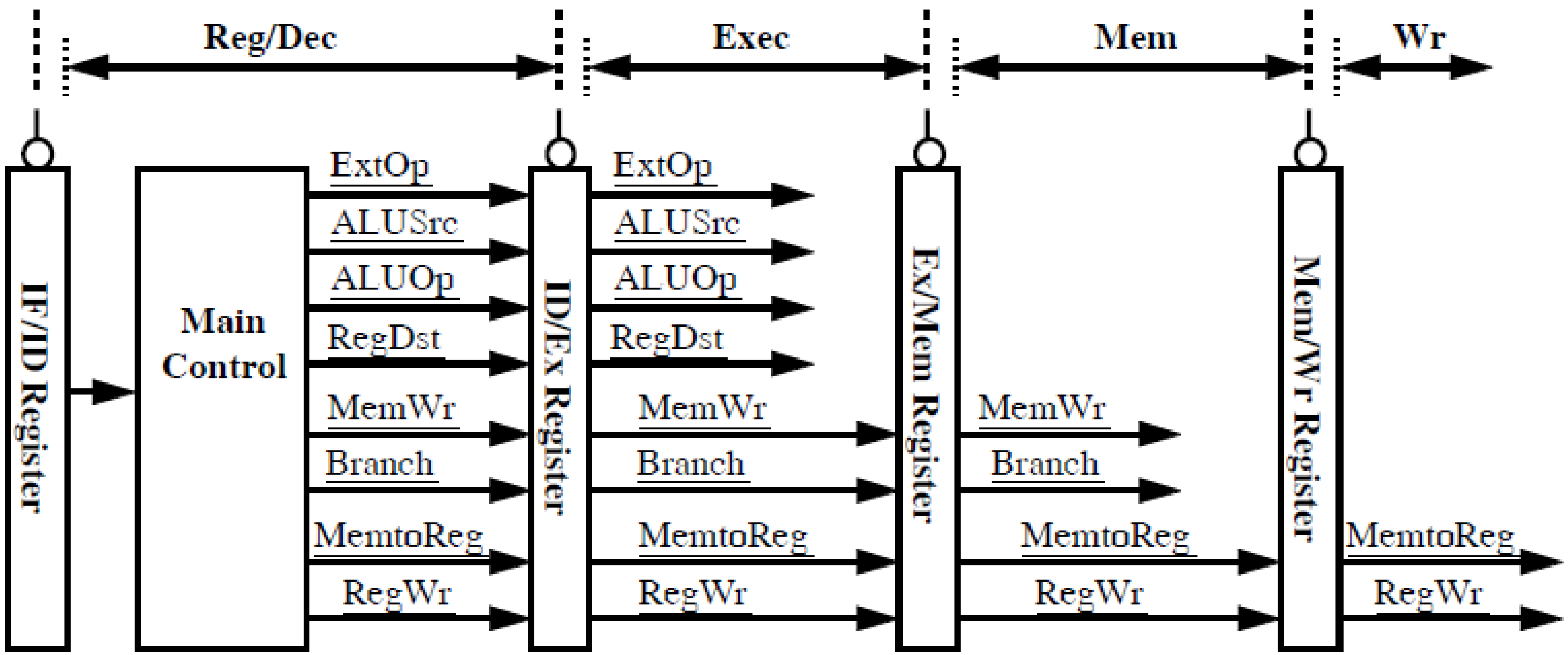
How About Control Signals?

- Key Observation: Control Signals at Stage N = Func (Instr. at Stage N)
  - N = Exec, Mem, or Wr
- Example: Controls Signals at Exec Stage = Func(Load's Exec)



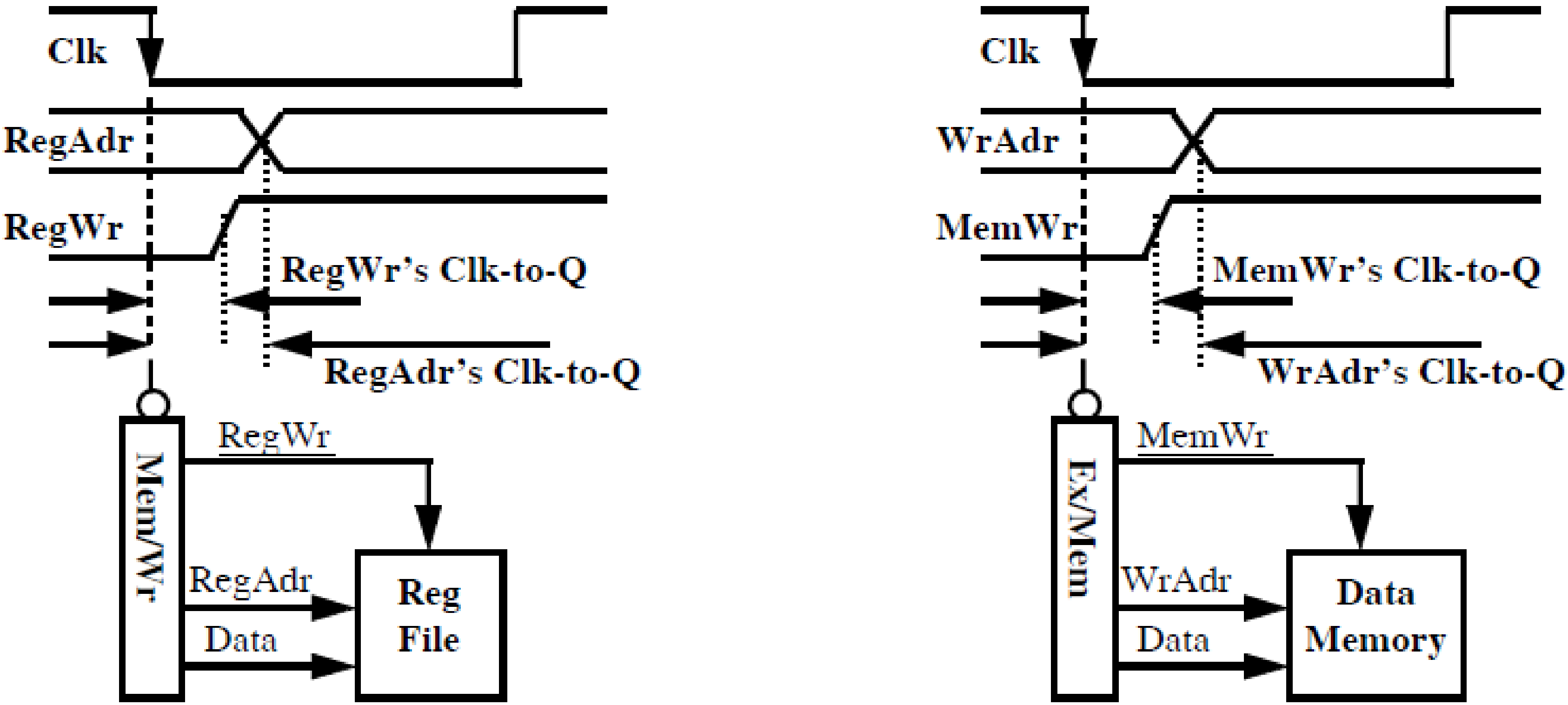
Pipeline Control

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



We need to add more fields for holding control bits to the intermediate register structures we defined in the previous fields.

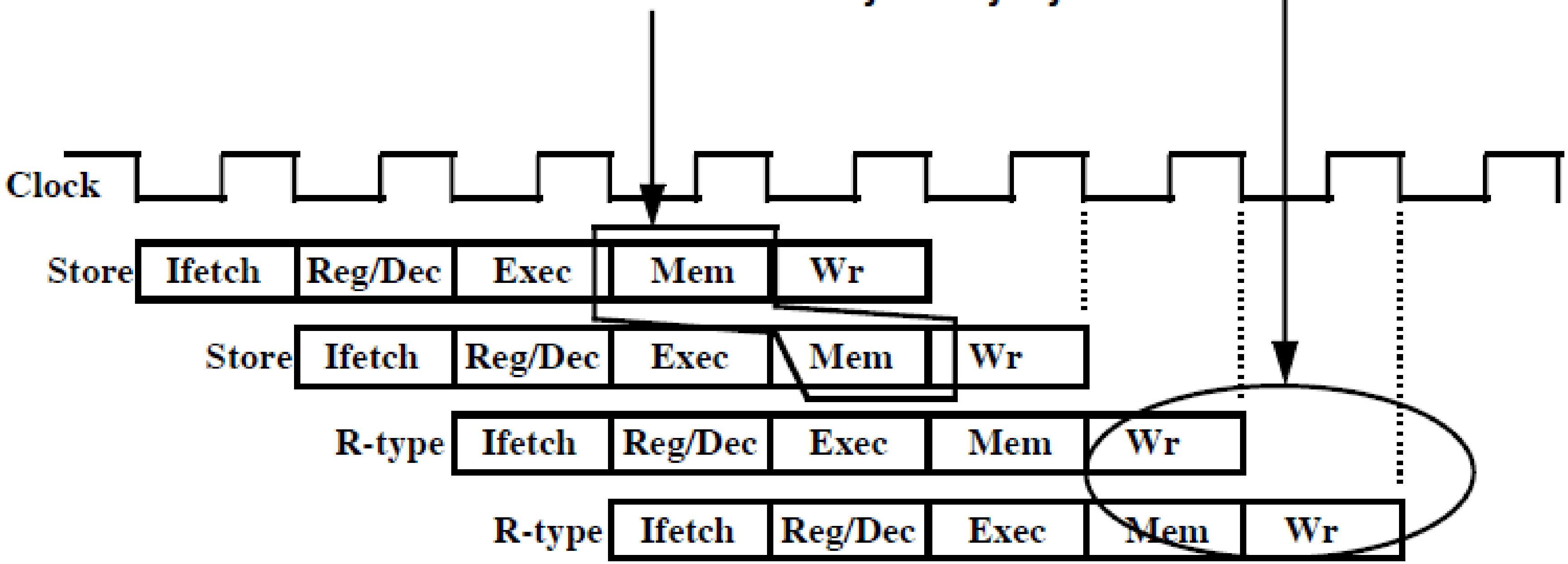
Beginning of the Wr's Stage: A Real World Problem



- At the beginning of the Wr stage, we have a problem if:
  - RegAdr's (Rd or Rt) Clk-to-Q > RegWr's Clk-to-Q
- Similarly, at the beginning of the Mem stage, we have a problem if:
  - WrAdr's Clk-to-Q > MemWr's Clk-to-Q
- We have a race condition between Address and Write Enable!

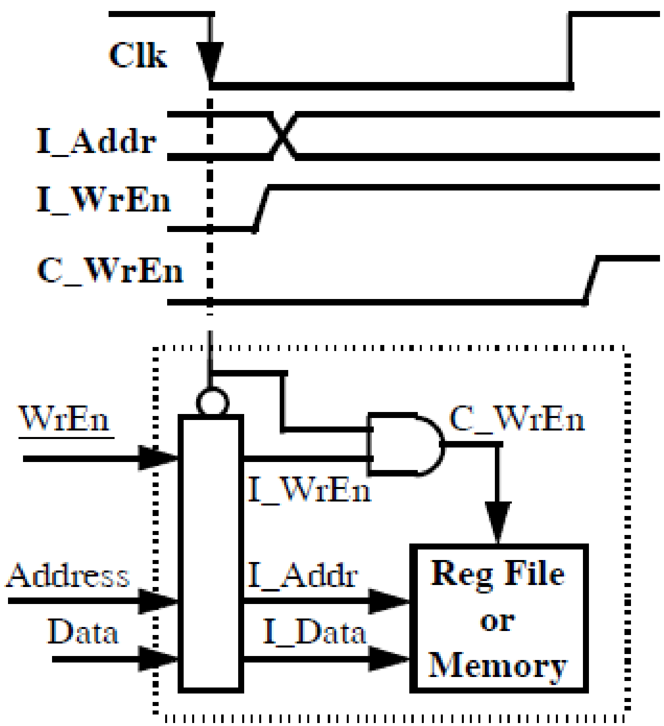
The Pipeline Problem

- Multiple Cycle design prevents race condition between Addr and WrEn:
  - Make sure Address is stable by the end of Cycle N
  - Asserts WrEn during Cycle N + 1
- This approach can NOT be used in the pipeline design because:
  - Must be able to write the register file every cycle
  - Must be able write the data memory every cycle



Synchronize Register File & Synchronize Memory

- Solution: And the Write Enable signal with the Clock
  - This is the ONLY place where gating the clock is used
  - MUST consult circuit expert to ensure no timing violation:
    - Example: Clock High Time > Write Access Delay

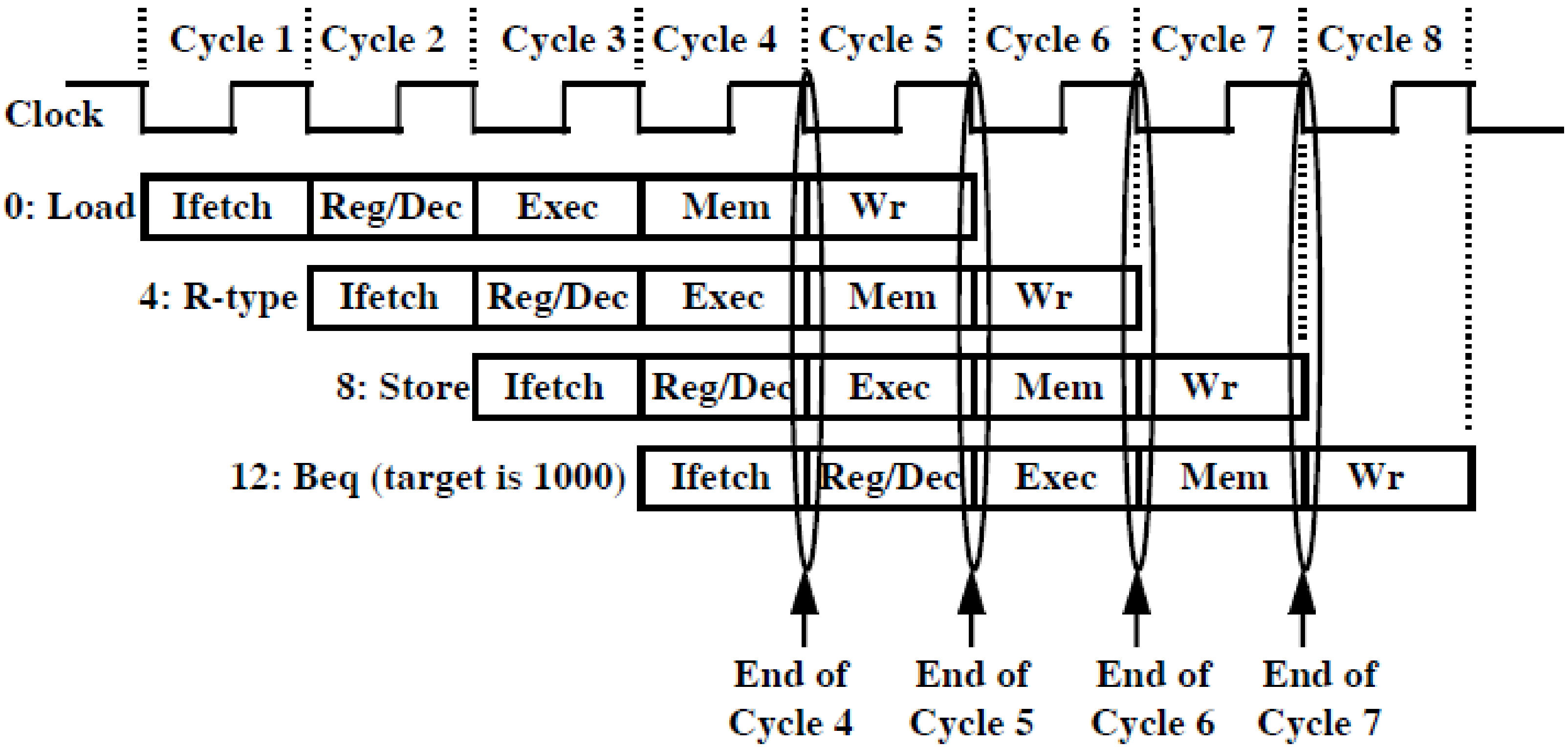


Synchronize Memory and Register File

Address, Data, and WrEn must be stable at least 1 set-up time before the Clk edge

Write occurs at the cycle following the clock edge that captures the signals

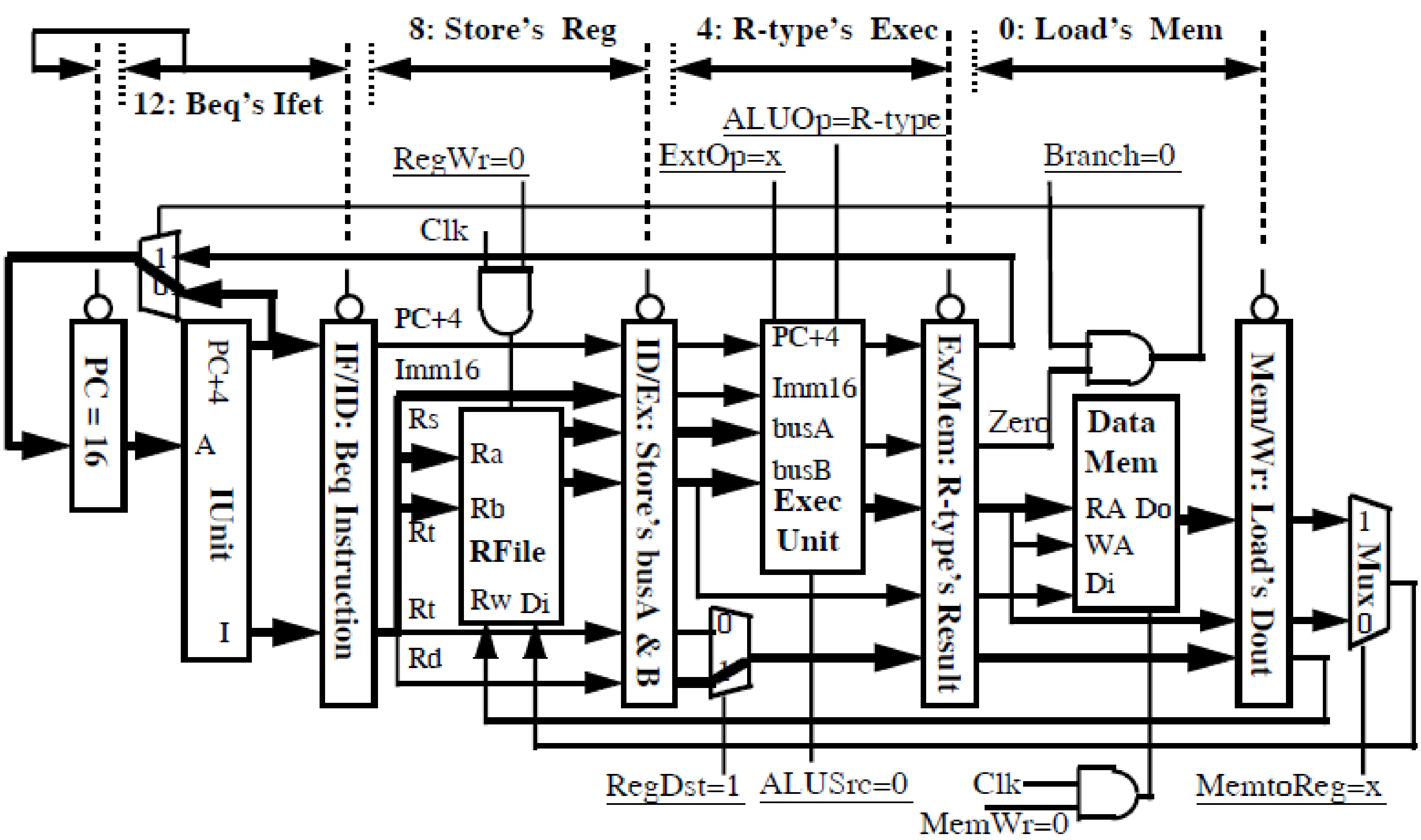
A More Extensive Pipelining Example



- End of Cycle 4: Load's Mem, R-type's Exec, Store's Reg, Beq's Ifetch
- End of Cycle 5: Load's Wr, R-type's Mem, Store's Exec, Beq's Reg
- End of Cycle 6: R-type's Wr, Store's Mem, Beq's Exec
- End of Cycle 7: Store's Wr, Beq's Mem

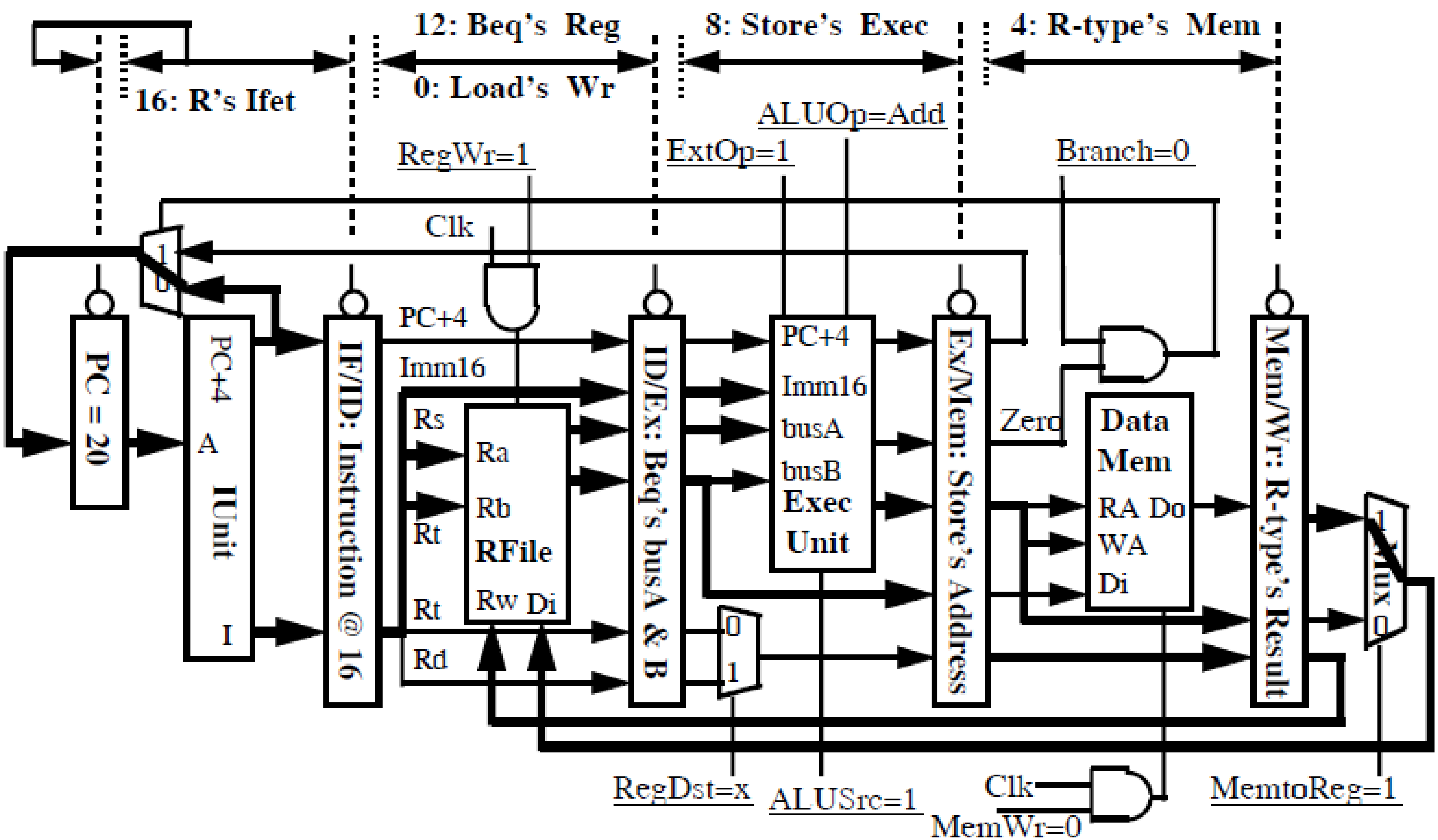
Pipelining Example: End of Cycle 4

- 0: Load's Mem   4: R-type's Exec   8: Store's Reg   12: Beq's Ifetch



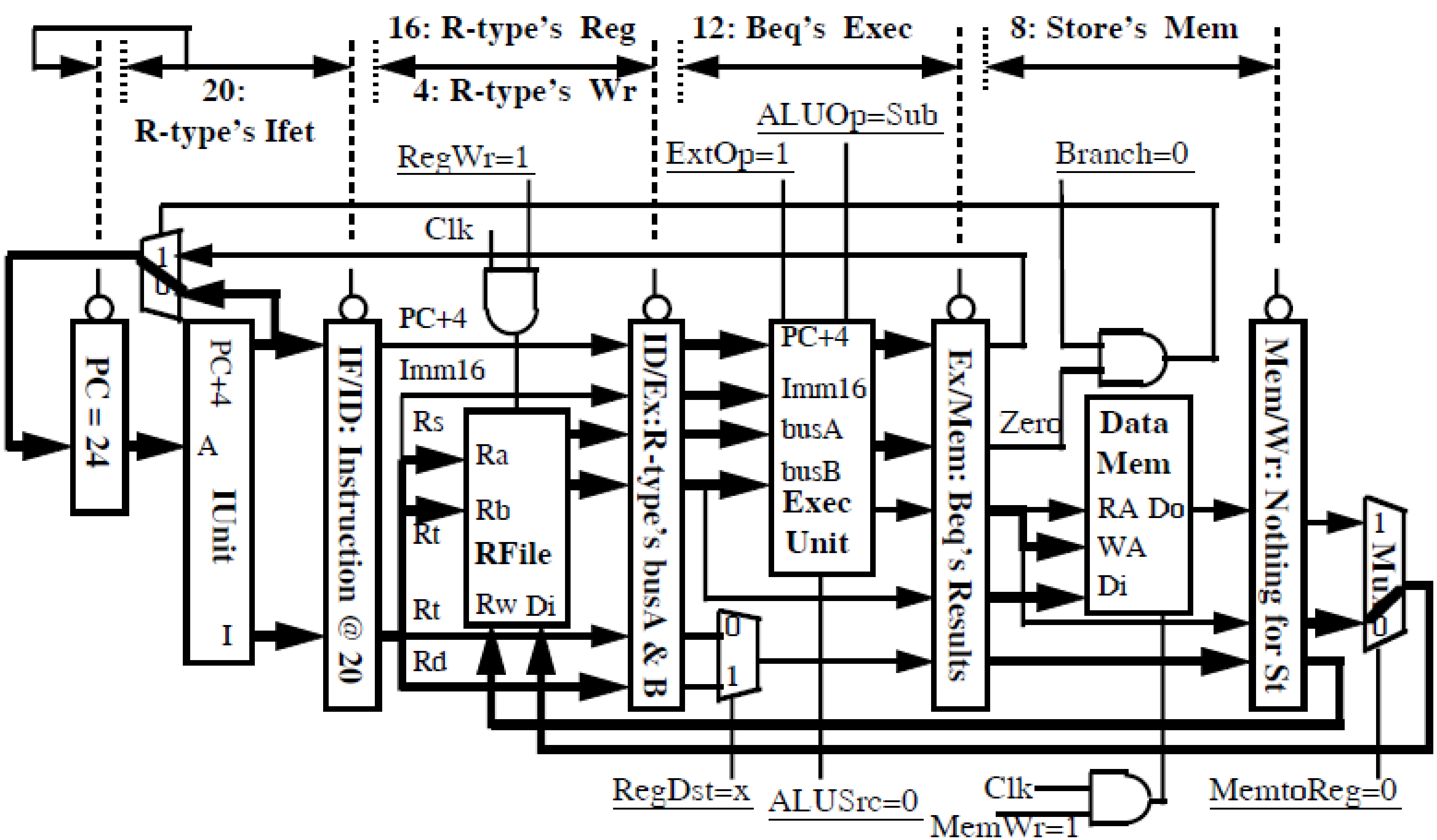
Pipelining Example: End of Cycle 5

- 0: Lw's Wr   4: R's Mem   8: Store's Exec   12: Beq's Reg   16: R's Ifetch



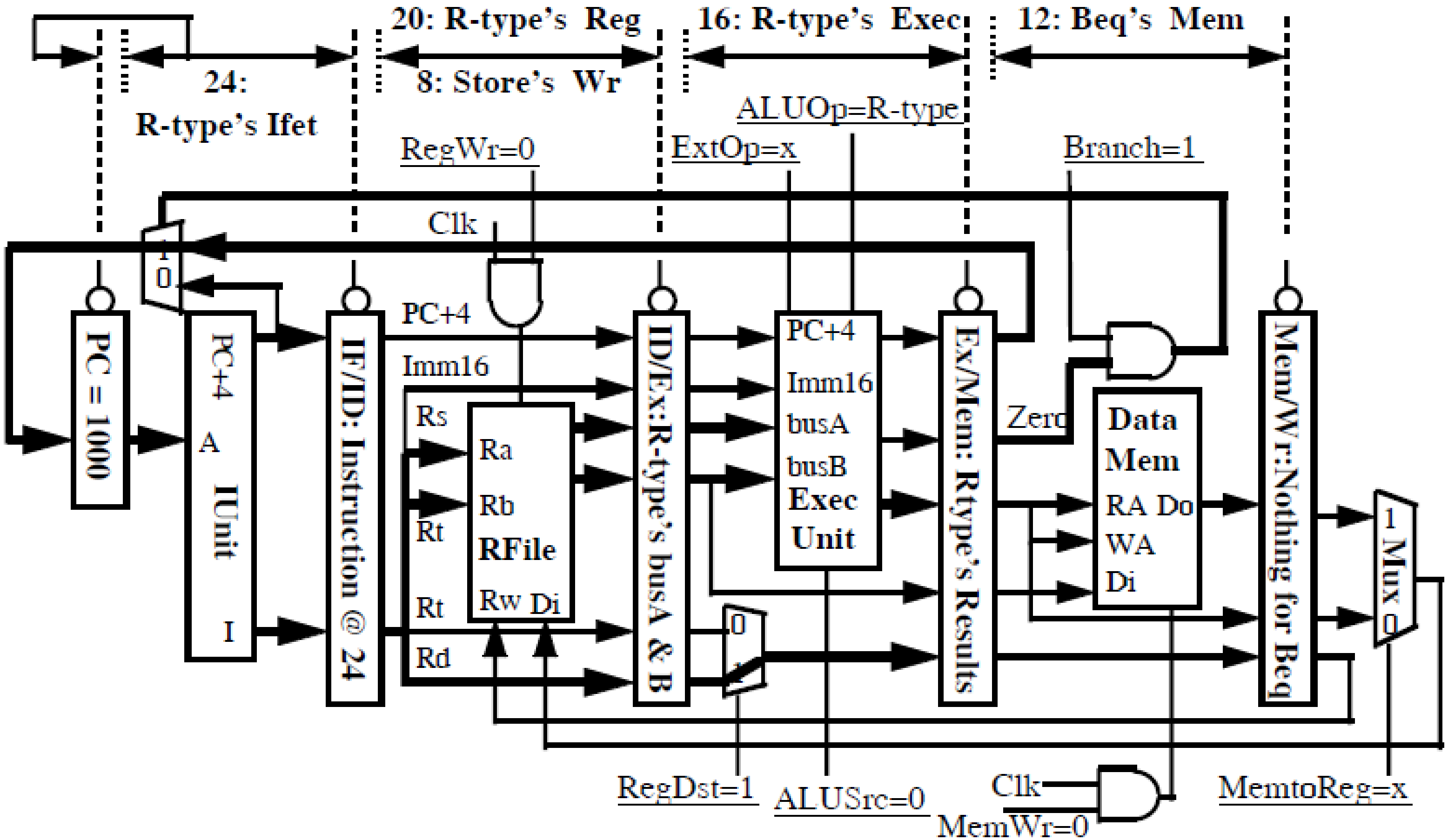
Pipelining Example: End of Cycle 6

° 4: R's Wr   8: Store's Mem   12: Beq's Exec   16: R's Reg   20: R's Ifet

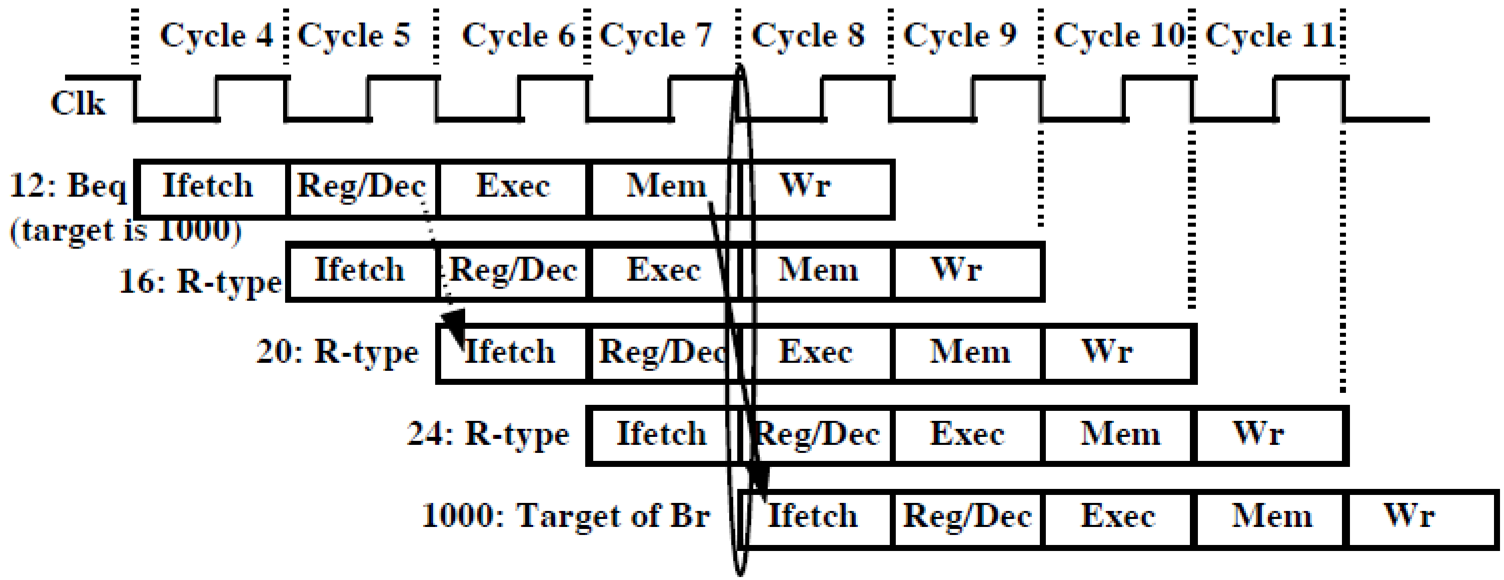


Pipelining Example: End of Cycle 7

- 8: Store's Wr 12: Beq's Mem 16: R's Exec 20: R's Reg 24: R's Ifet

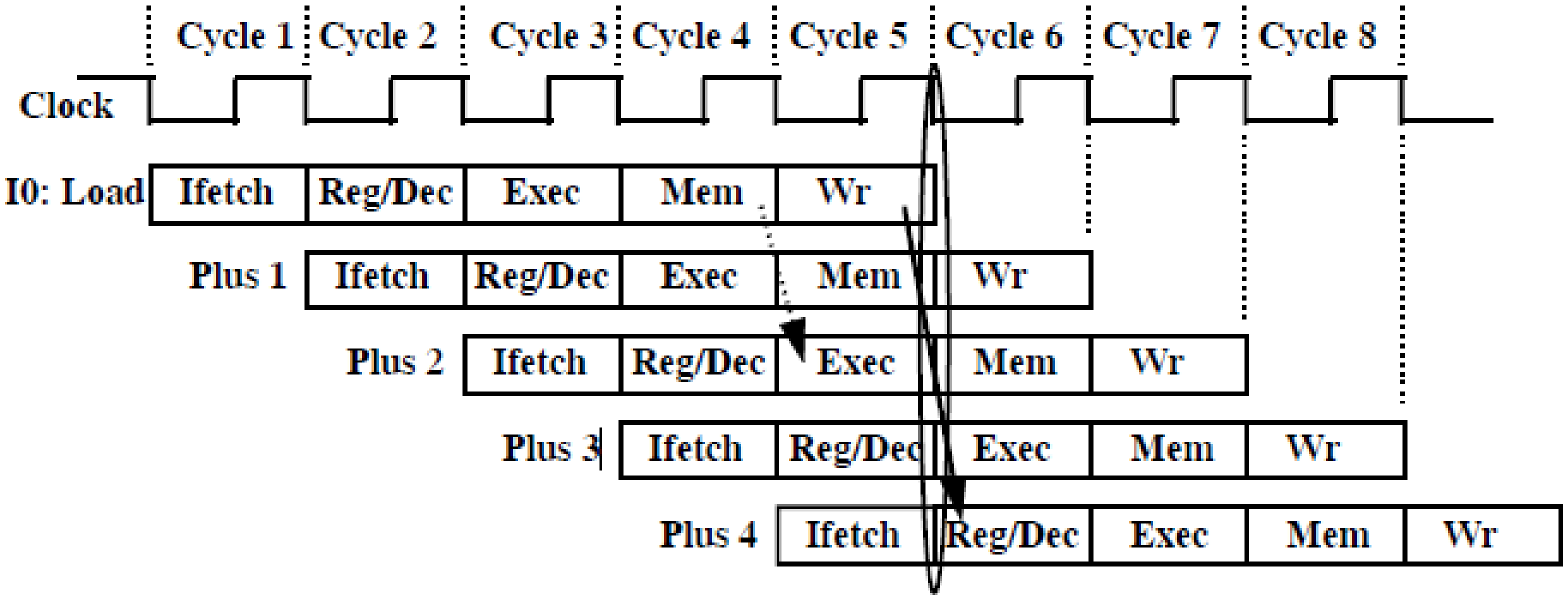


The Delay Branch Phenomenon



- Although Beq is fetched during Cycle 4:
  - Target address is NOT written into the PC until the end of Cycle 7
  - Branch's target is NOT fetched until Cycle 8
  - 3-instruction delay before the branch take effect
- This is referred to as Branch Hazard:
  - Clever design techniques can reduce the delay to ONE instruction

The Delay Load Phenomenon

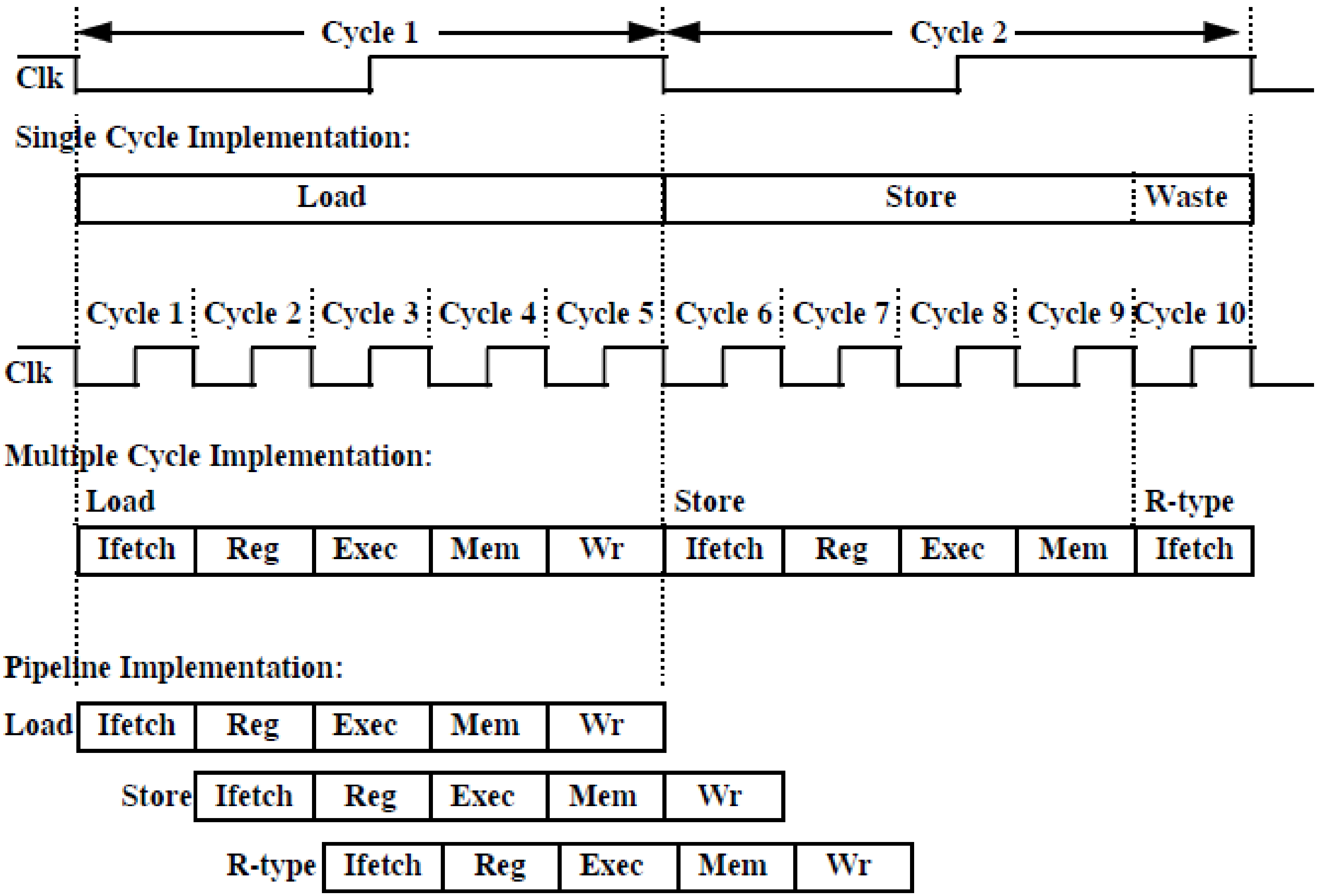


- Although Load is fetched during Cycle 1:
  - The data is NOT written into the Reg File until the end of Cycle 5
  - We cannot read this value from the Reg File until Cycle 6
  - 3-instruction delay before the load take effect
- This is referred to as Data Hazard:
  - Clever design techniques can reduce the delay to ONE instruction

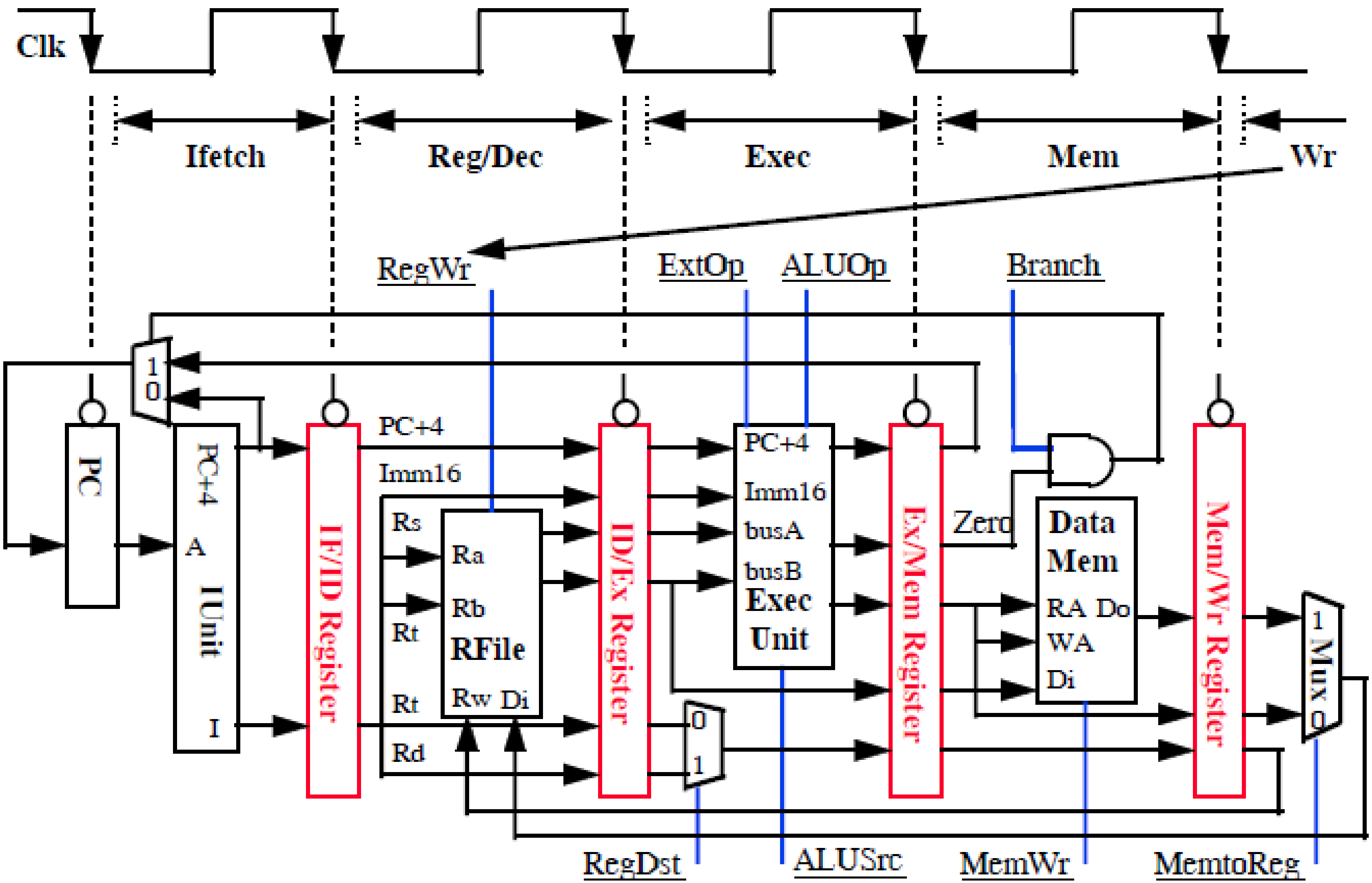
Summary

- Disadvantages of the Single Cycle Processor
  - Long cycle time
  - Cycle time is too long for all instructions except the Load
- Multiple Clock Cycle Processor:
  - Divide the instructions into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
- Pipeline Processor:
  - Natural enhancement of the multiple clock cycle processor
  - Each functional unit can only be used once per instruction
  - If a instruction is going to use a functional unit:
    - it must use it at the same stage as all other instructions
  - Pipeline Control:
    - Each stage's control signal depends ONLY on the instruction that is currently in that stage

Single Cycle, Multiple Cycle, vs. Pipeline

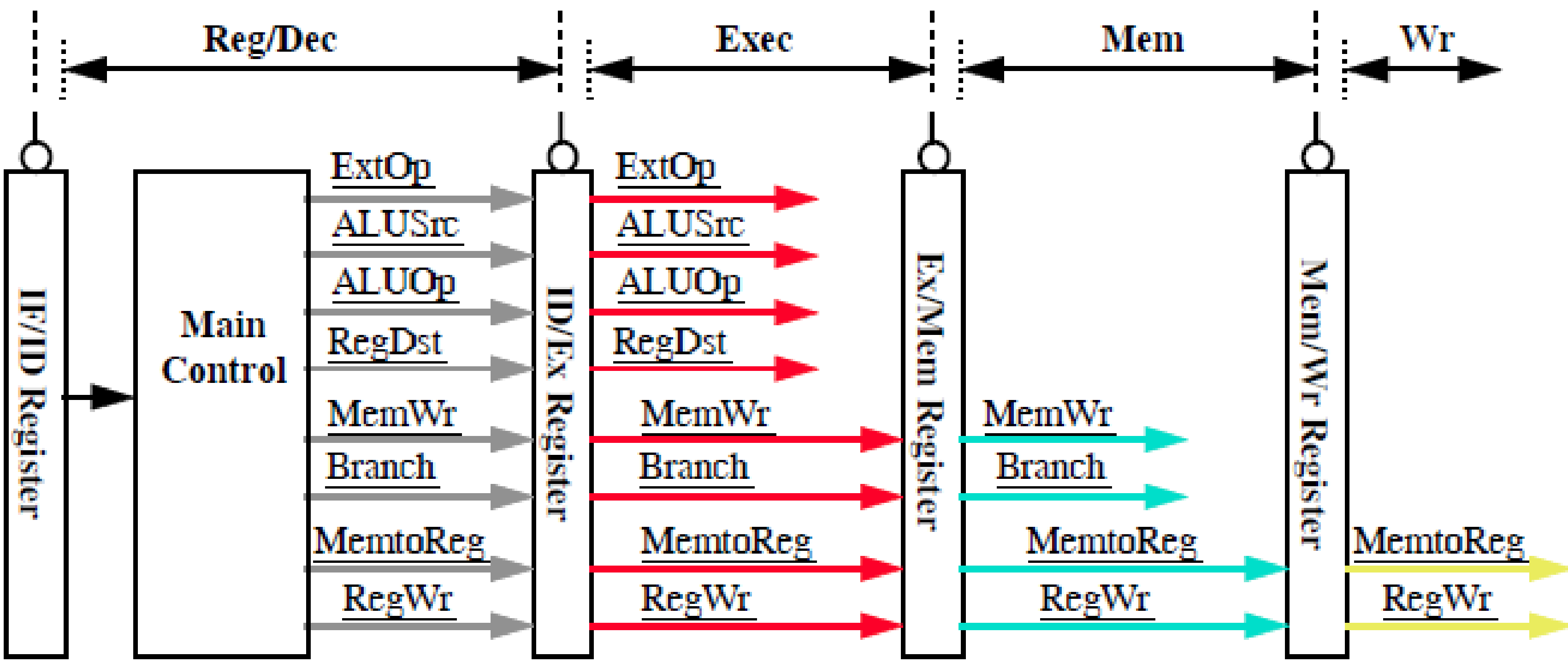


A Pipelined Datapath



Pipeline Control "Data Stationary Control"

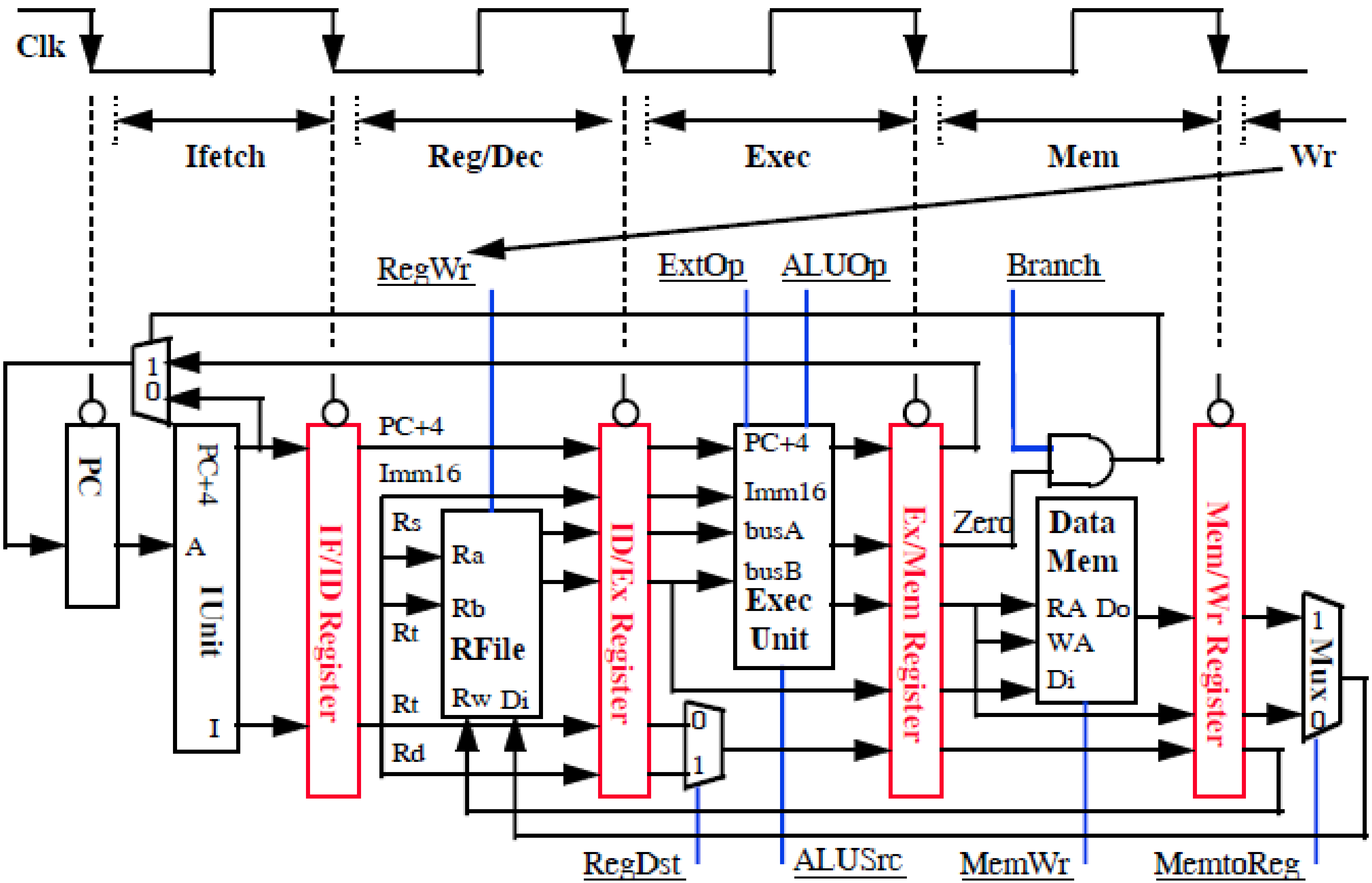
- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



Acknowledgment: Almost all of these slides are based on Dave Patterson’s CS152 Lecture Slides at UC, Berkeyley.

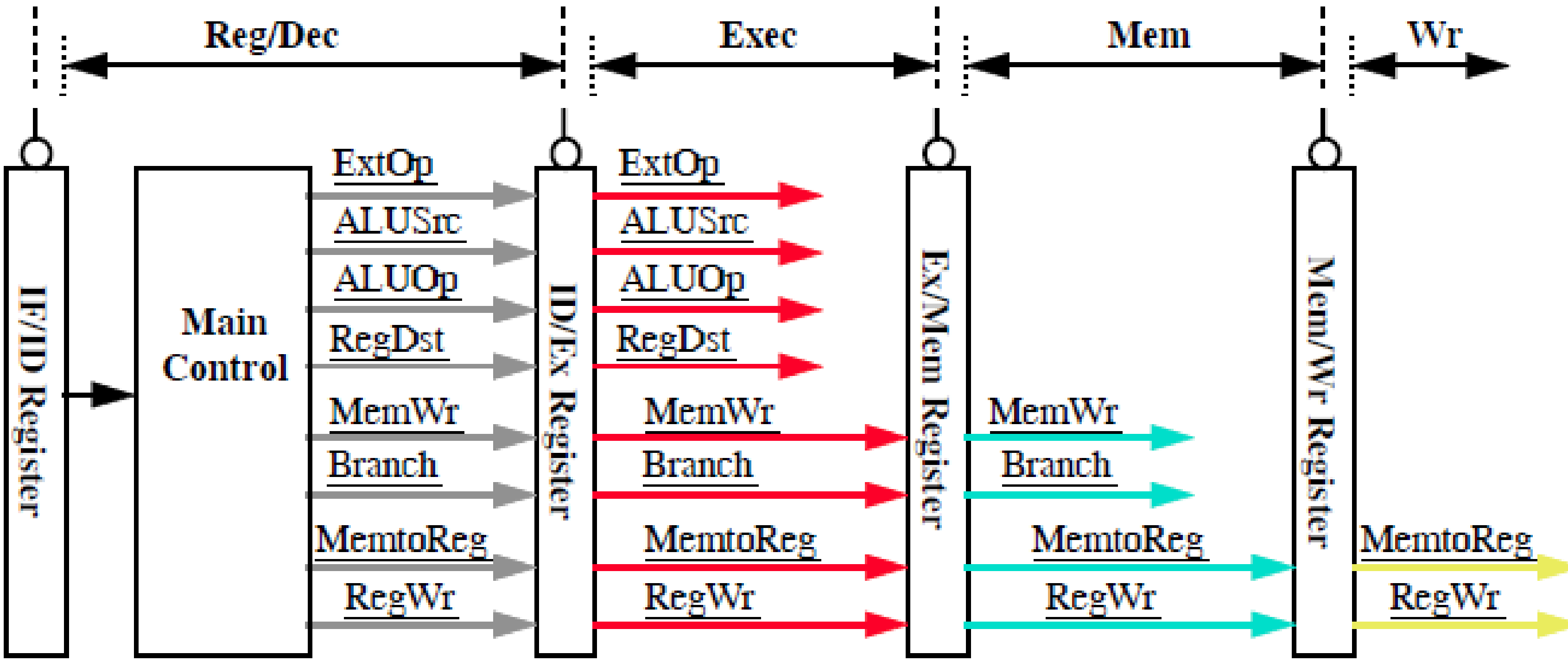
COMPUTER SYSTEMS ORGANIZATION

A Pipelined Datapath



Pipeline Control "Data Stationary Control"

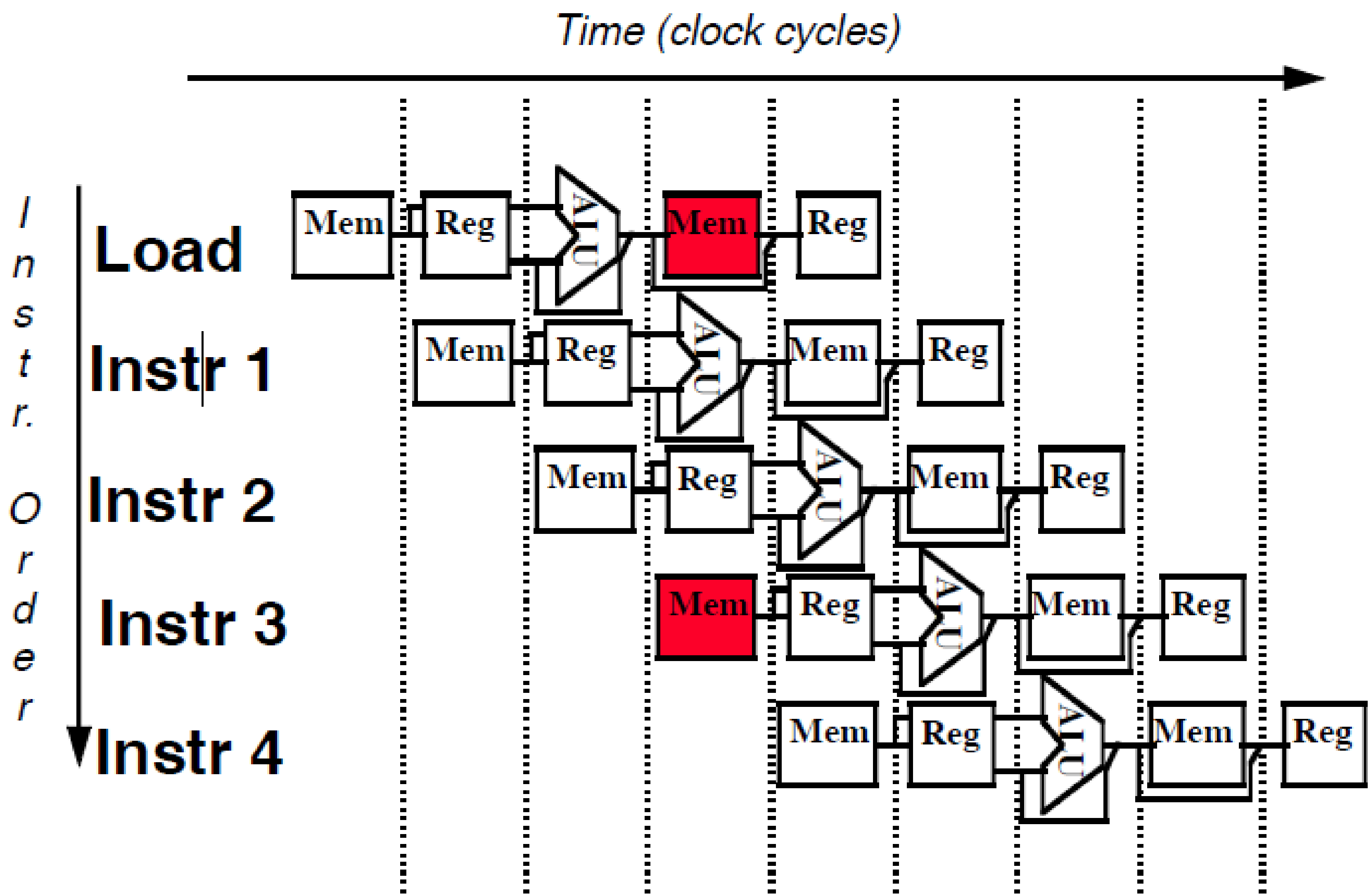
- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



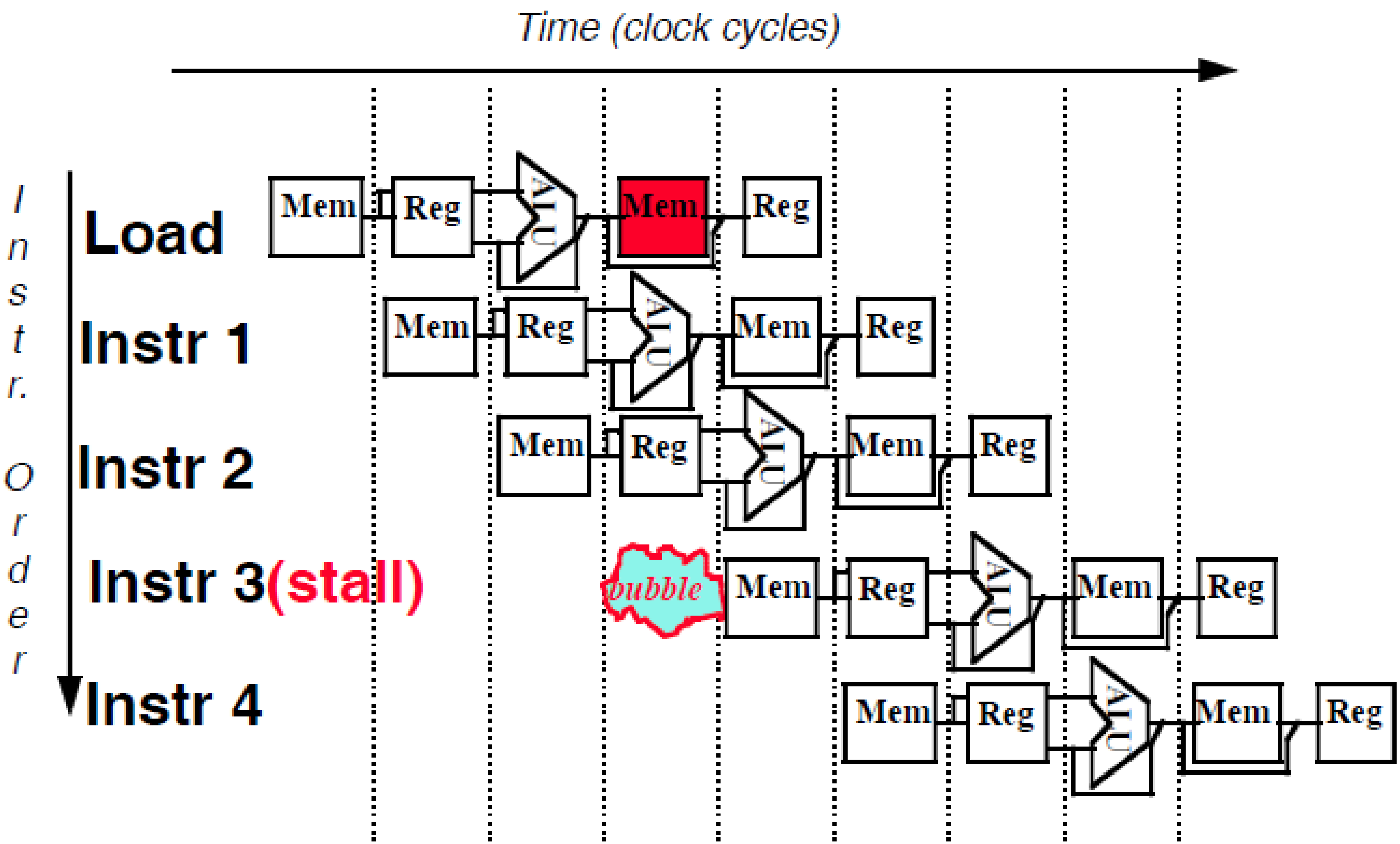
Its not that easy for computers

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - structural hazards: HW cannot support this combination of instructions
  - data hazards: instruction depends on result of prior instruction still in the pipeline
  - control hazards: pipelining of branches & other instructions that change the PC
- Common solution is to stall the pipeline until the hazard is resolved, inserting one or more "bubbles" in the pipeline

Single Memory is a Structural Hazard

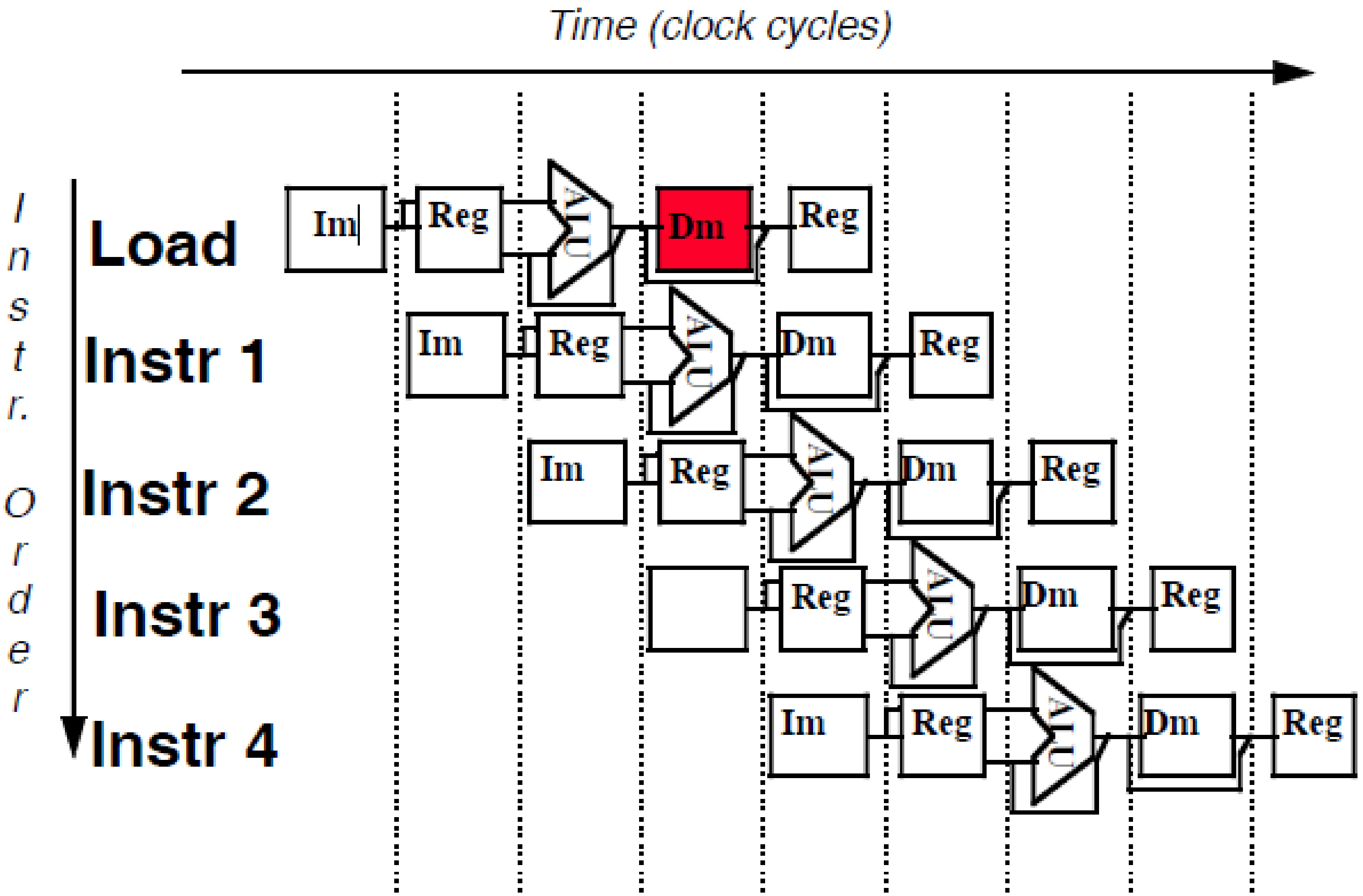


Option 1: Stall to resolve Memory Structural Hazard



Option 2: Duplicate to Resolve Structural Hazard

- Separate Instruction Cache (Im) & Data Cache (Dm)



Data Hazard on r1

add r1 ,r2,r3

sub r4, r1 ,r3

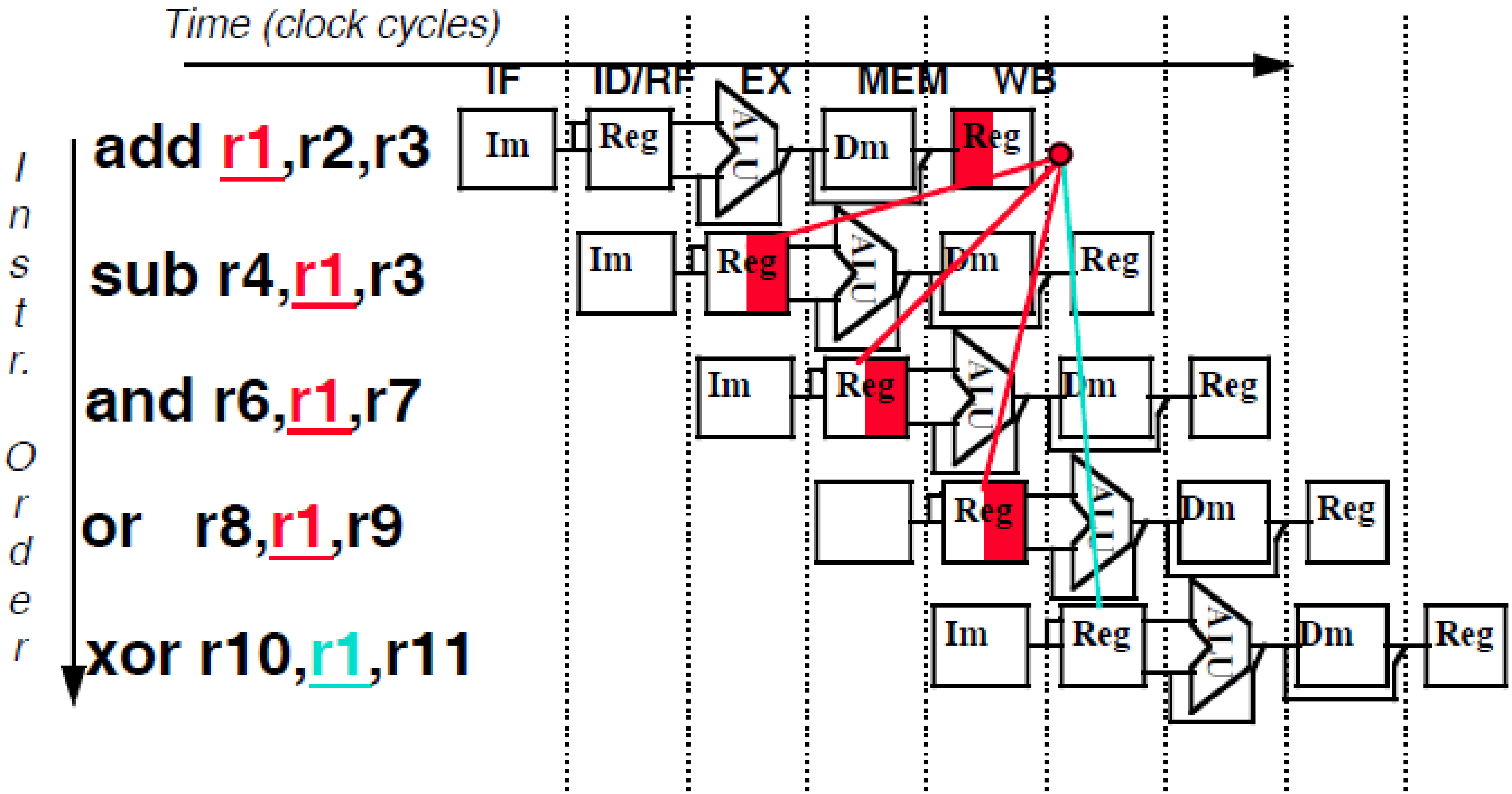
and r6, r1 ,r7

or   r8, r1 ,r9

xor r10, r1 ,r11

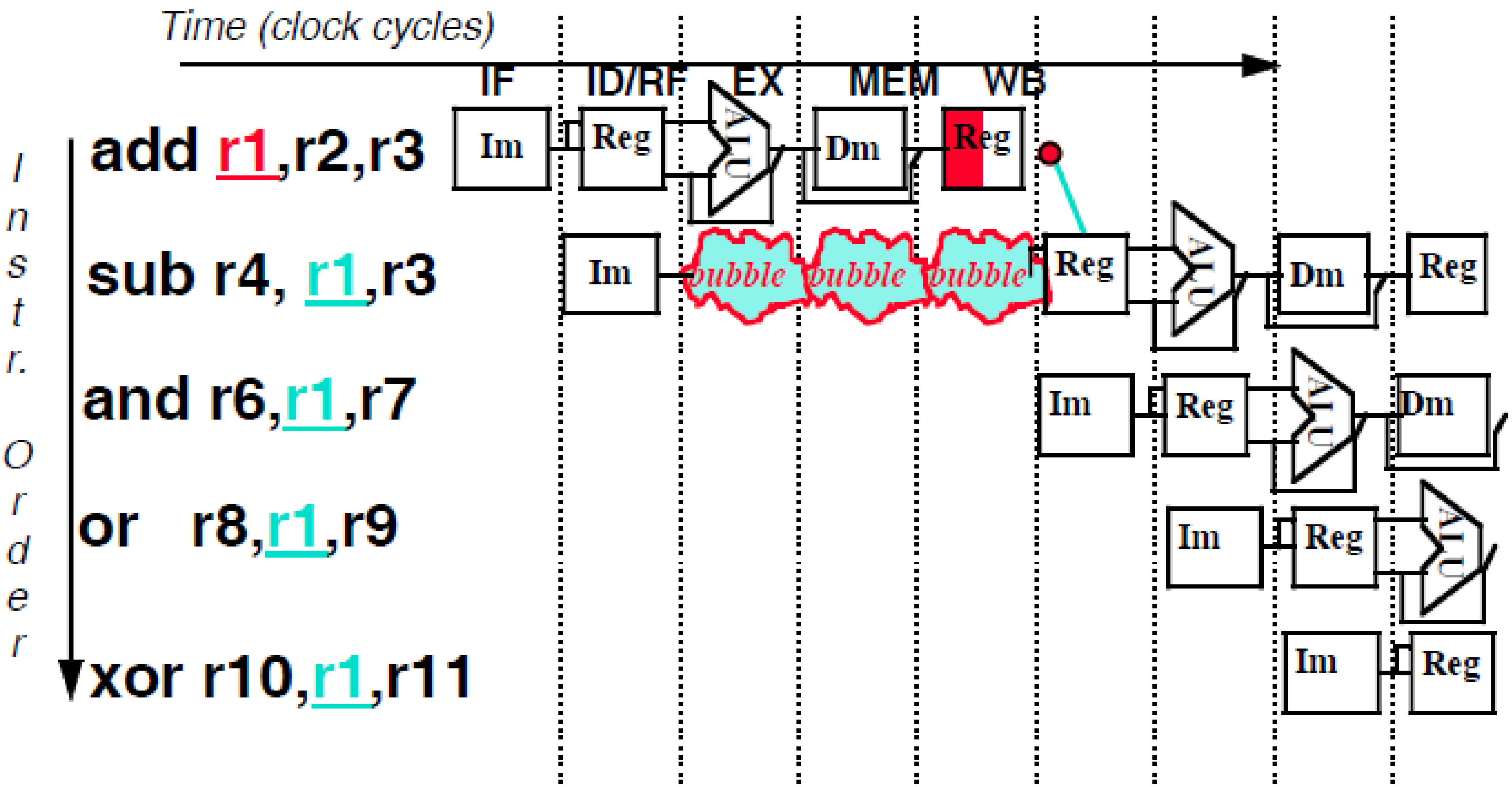
Data Hazard on r1:

- Dependencies backwards in time are hazards



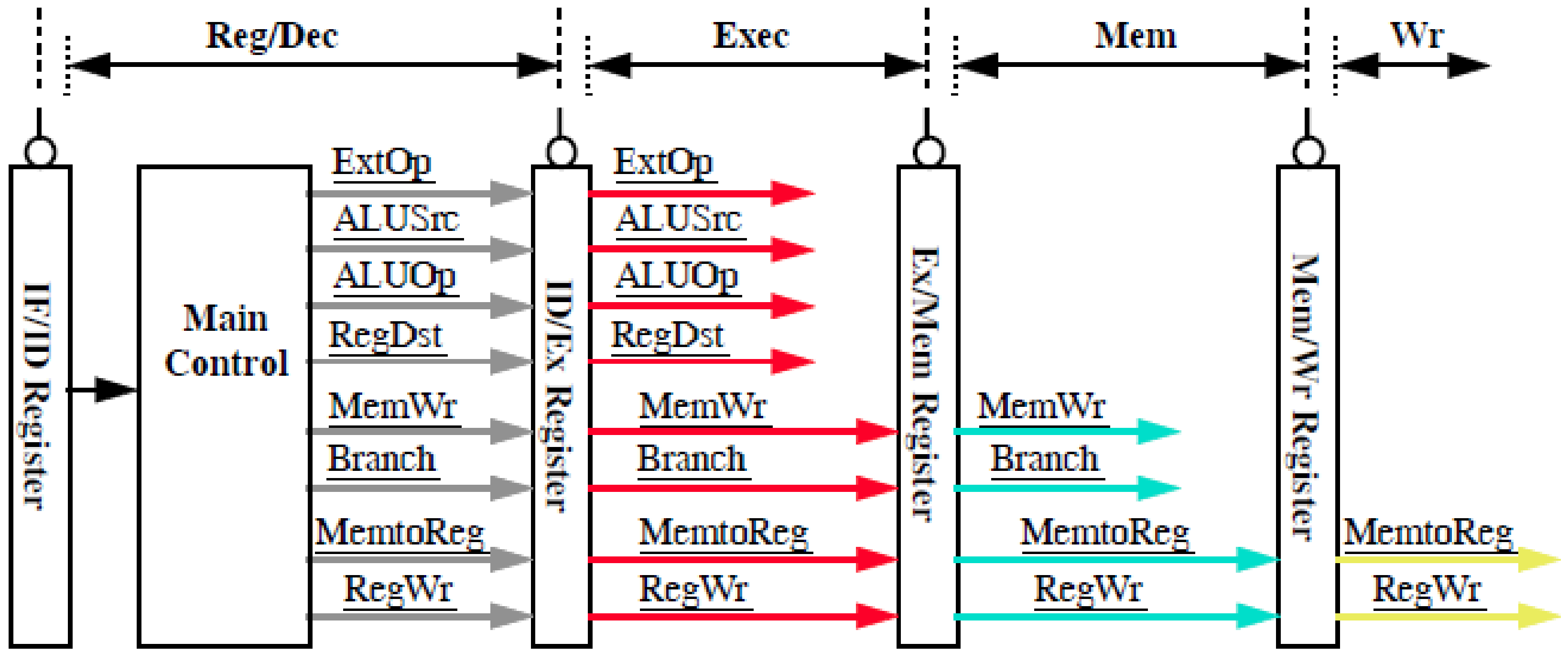
Option1: HW Stalls to Resolve Data Hazard

- Dependencies backwards in time are hazards



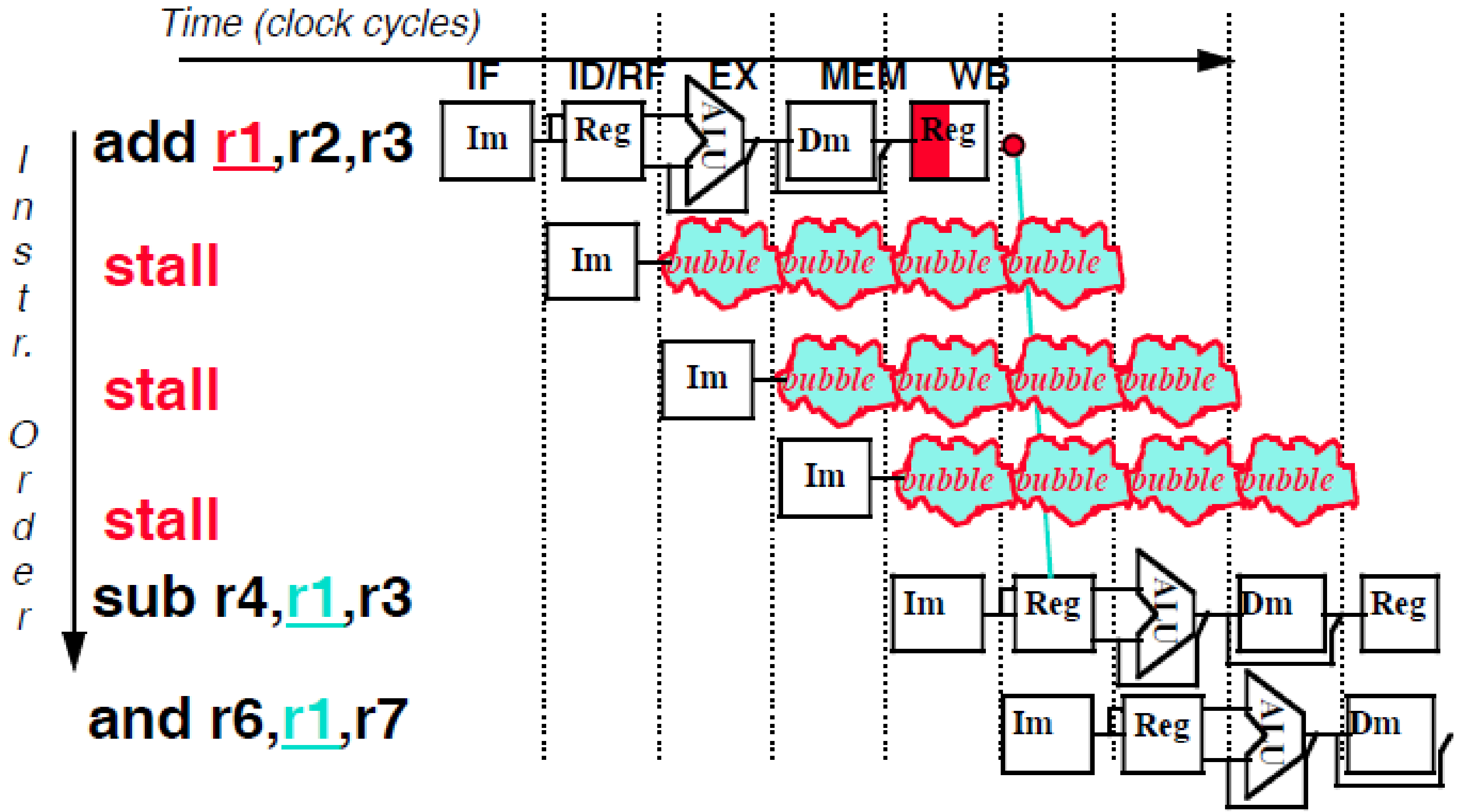
But recall use of “Data Stationary Control”

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



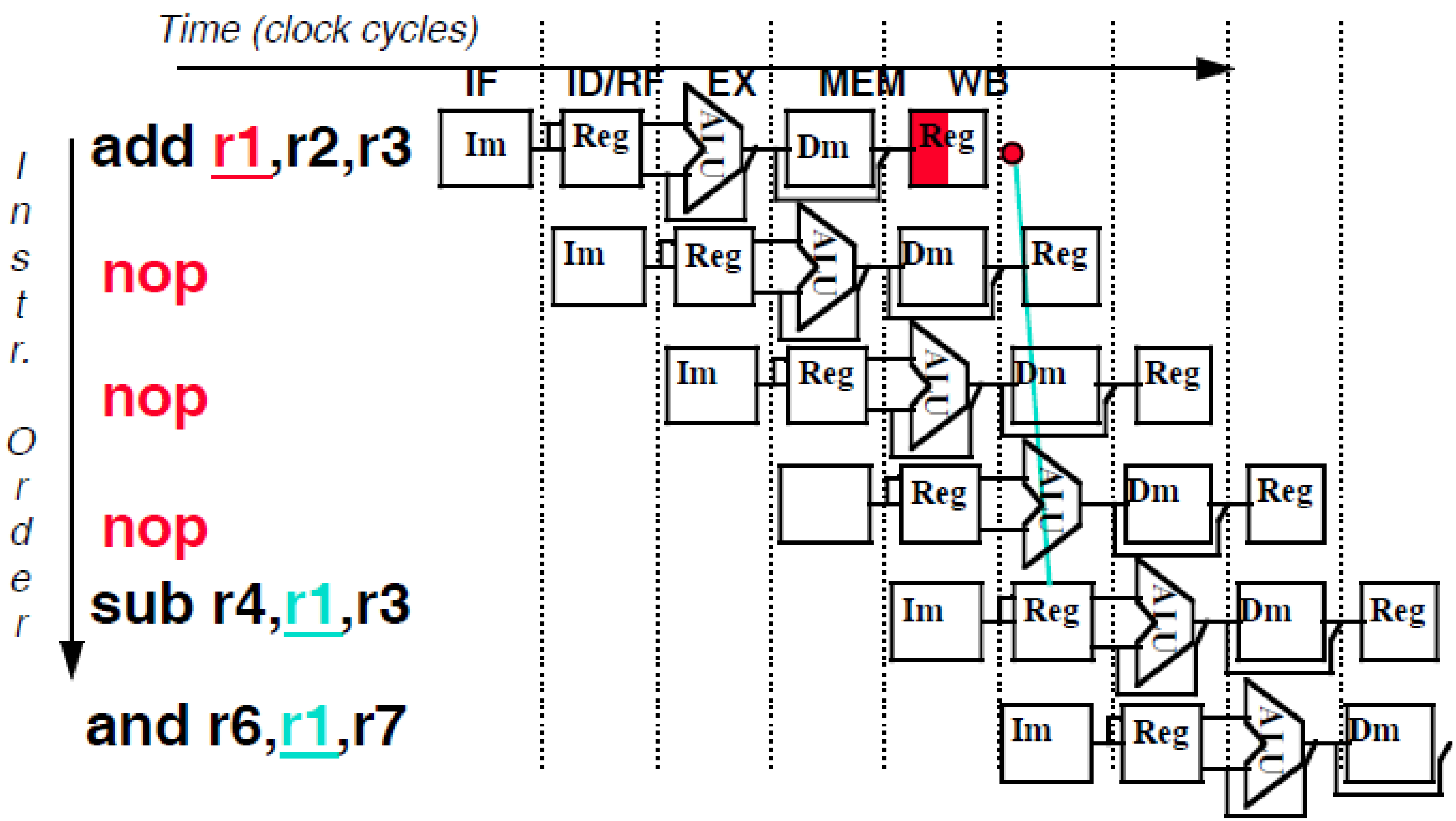
Option 1: How HW really stalls pipeline

- HW doesn't change PC => keeps fetching same instruction & sets control signals to to benign values (0)



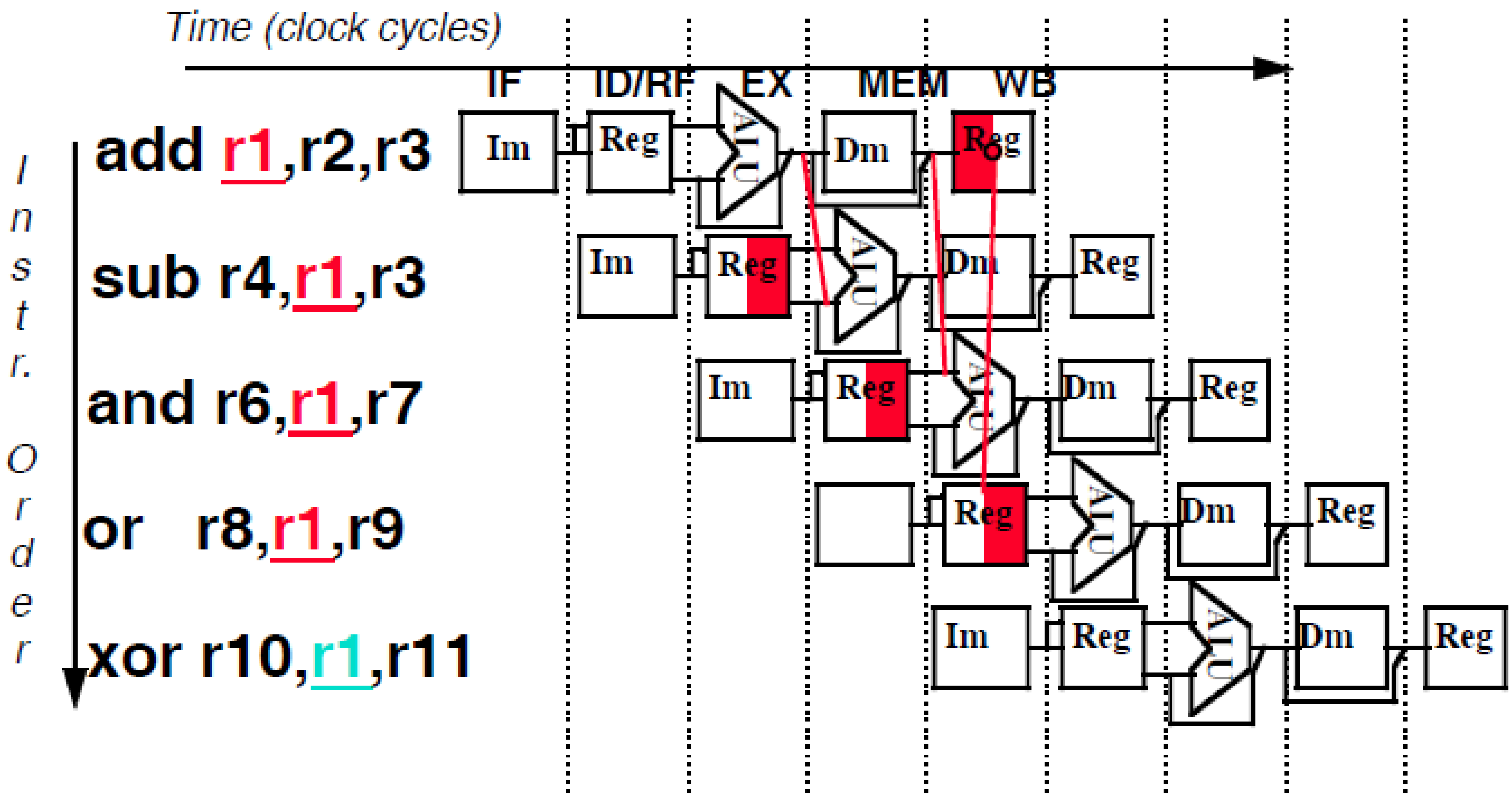
Option 2: SW inserts indepdendent instructions

- **Worst case inserts NOP instructions**



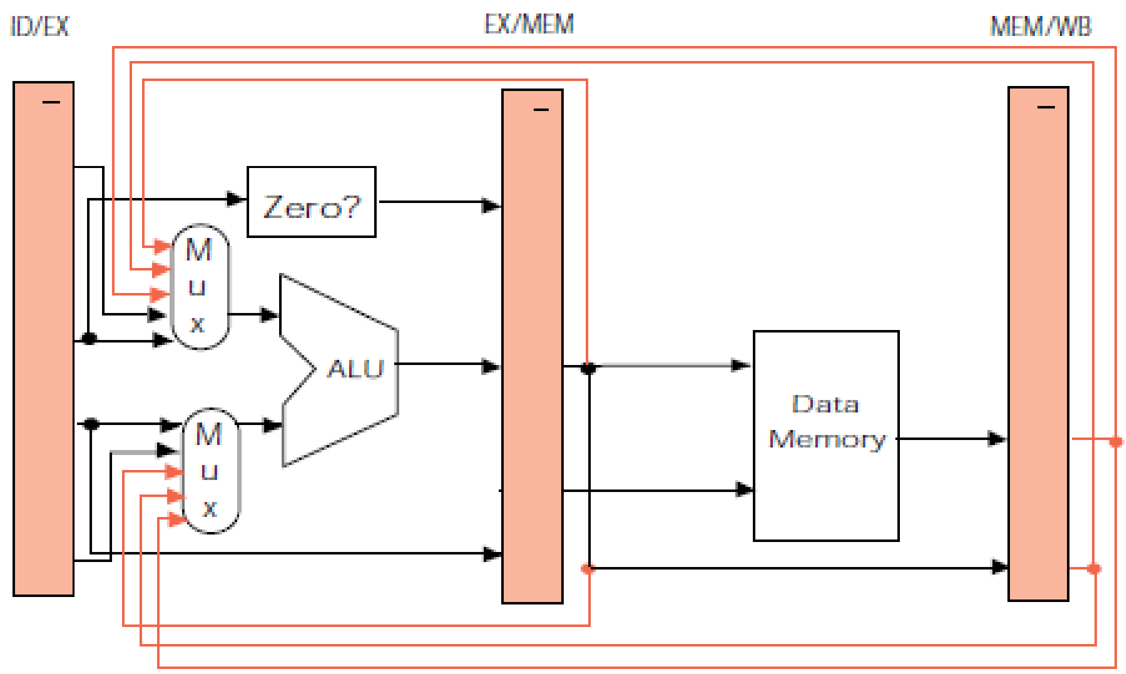
Option 3 Insight: Data is available!

- Pipeline registers already contain needed data

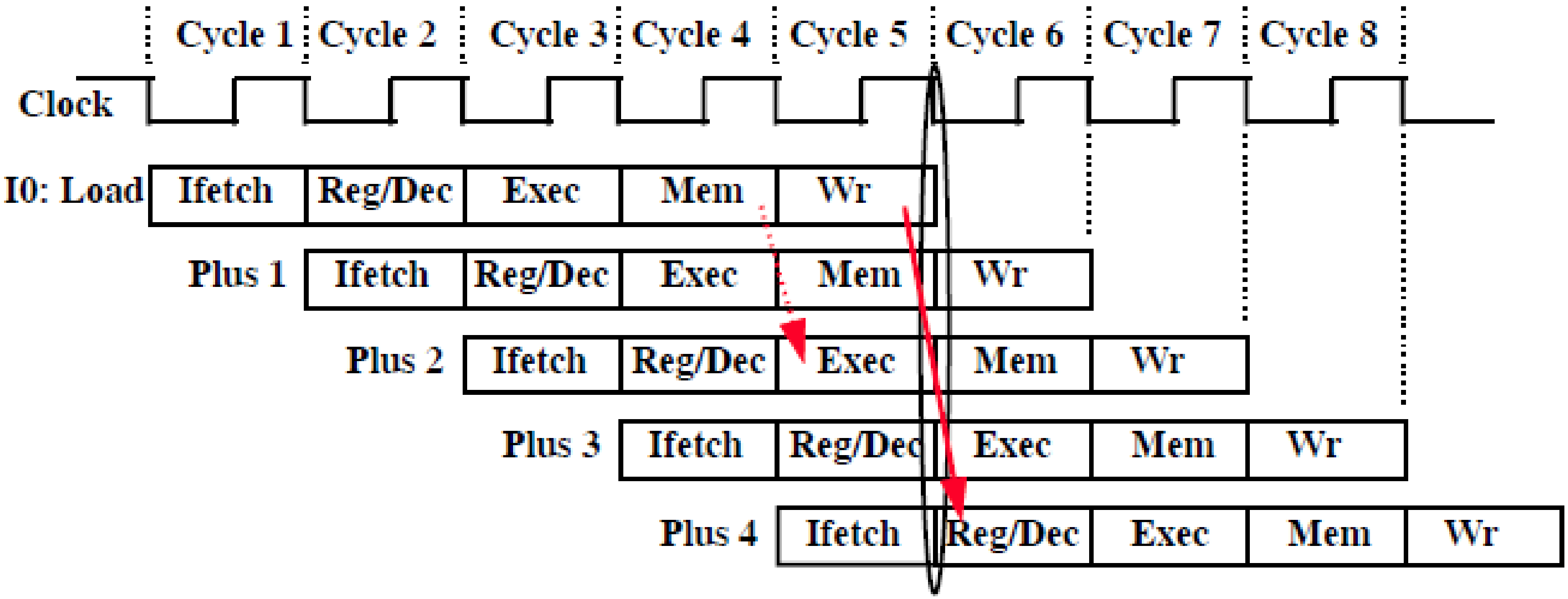


HW Change for "Forwarding" (Bypassing):

- Increase multiplexors to add paths from pipeline registers
- Assumes register read during write gets new value (otherwise more results to be forwarded)

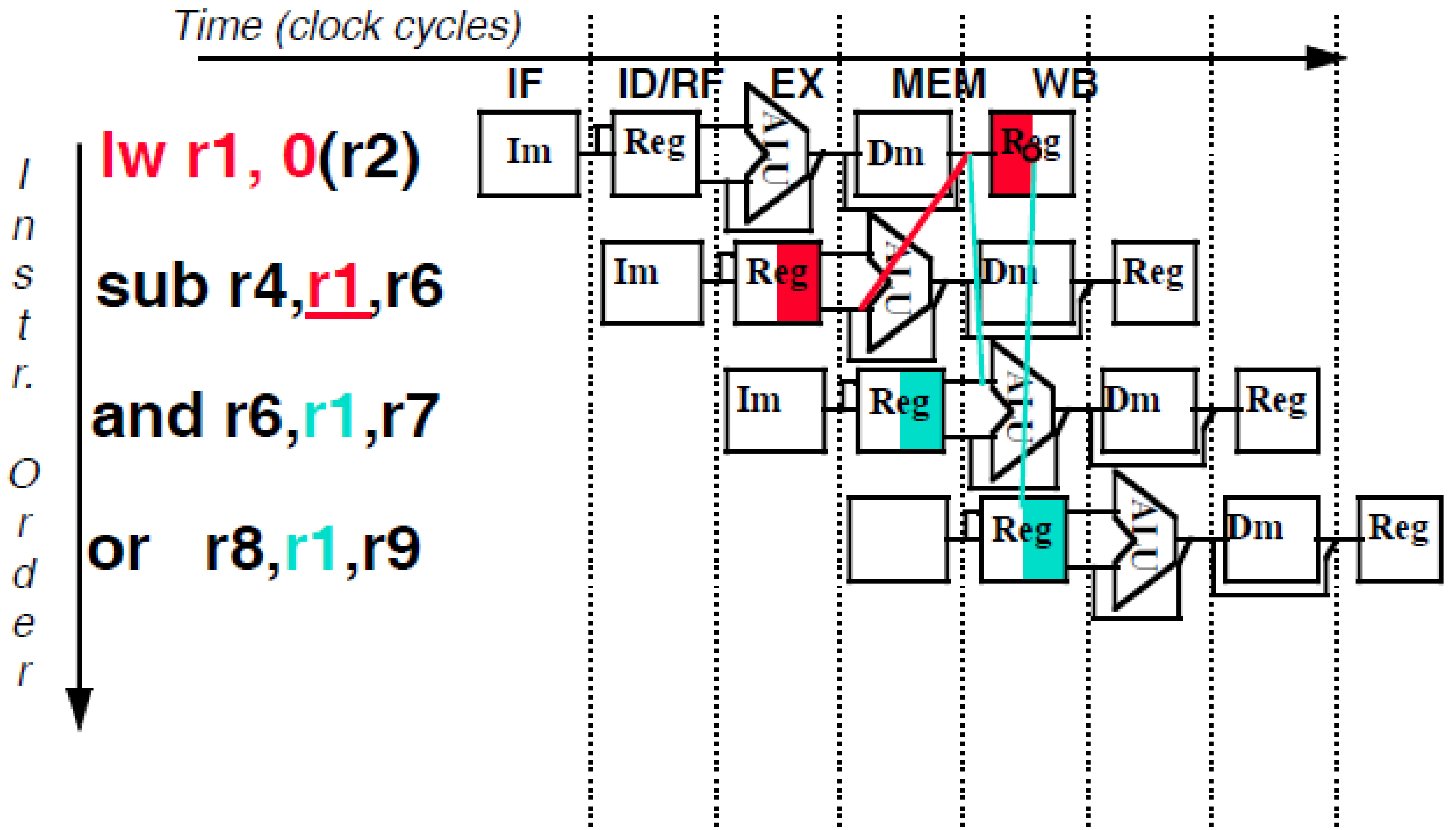


From Last Lecture: The Delay Load Phenomenon



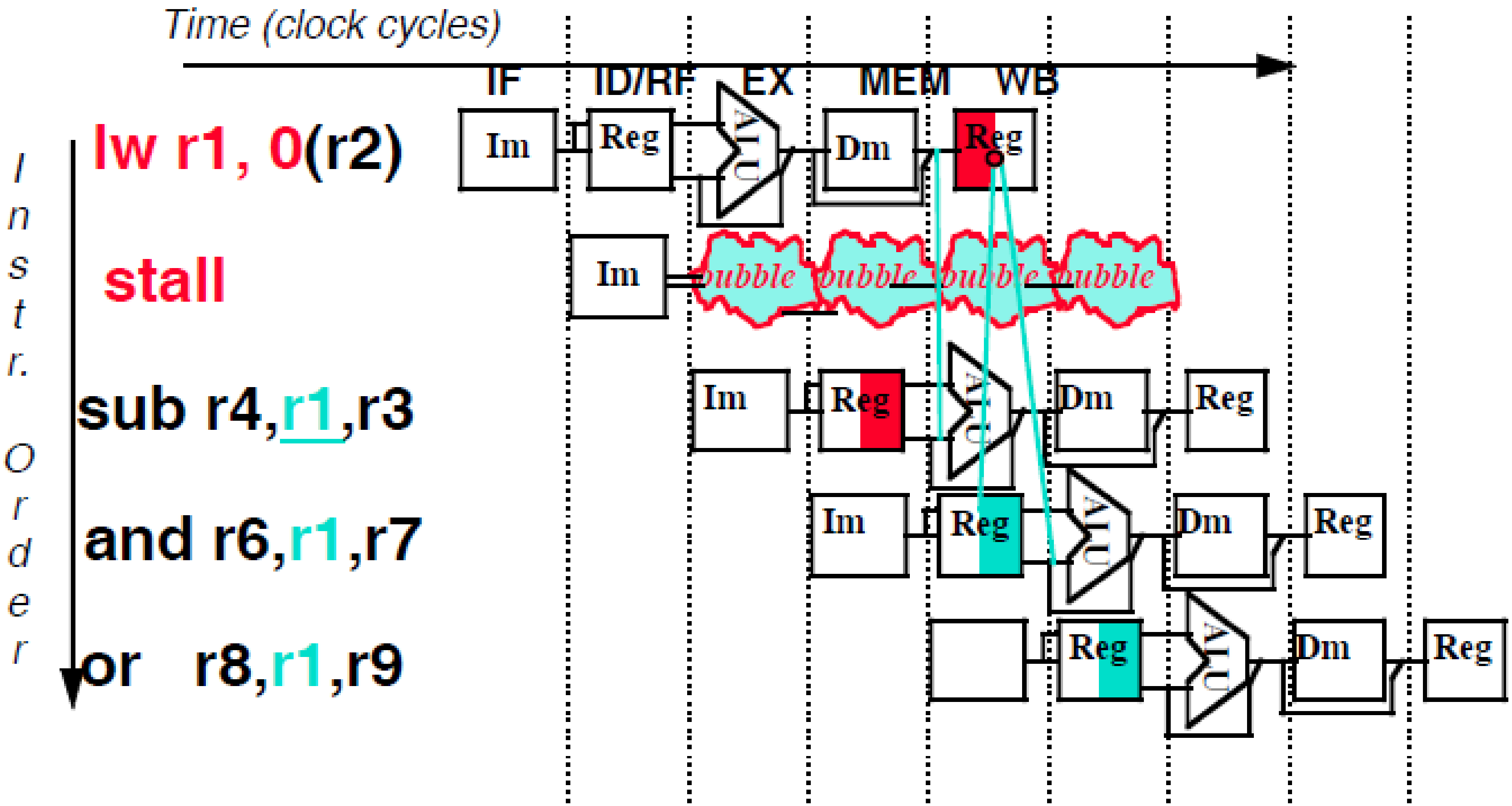
- Although Load is fetched during Cycle 1:
  - The data is NOT written into the Reg File until the end of Cycle 5
  - We cannot read this value from the Reg File until Cycle 6
  - 3-instruction delay before the load take effect

Forwarding reduces Data Hazard to 1 cycle:



Option1: HW Stalls to Resolve Data Hazard

- "Interlock": checks for hazard & stalls



Option 2: SW inserts independent instructions

- Worst case inserts NOP instructions
- MIPS I solution: No HW checking

Time (clock cycles)

Instr. Order

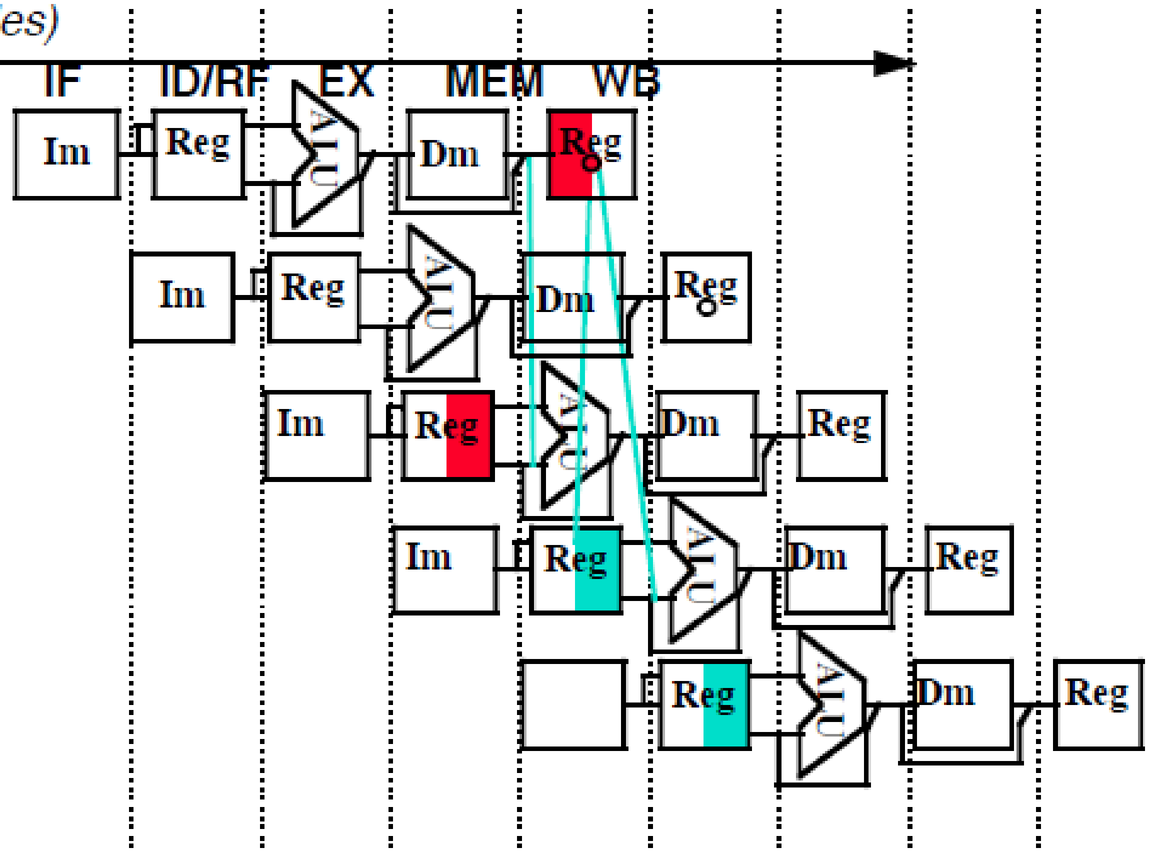
lw r1, 0(r2)

nop

sub r4,r1,r3

and r6,r1,r7

or r8,r1,r9



Software Scheduling to Avoid Load Hazards

Try producing fast code for

$$a = b + c;$$

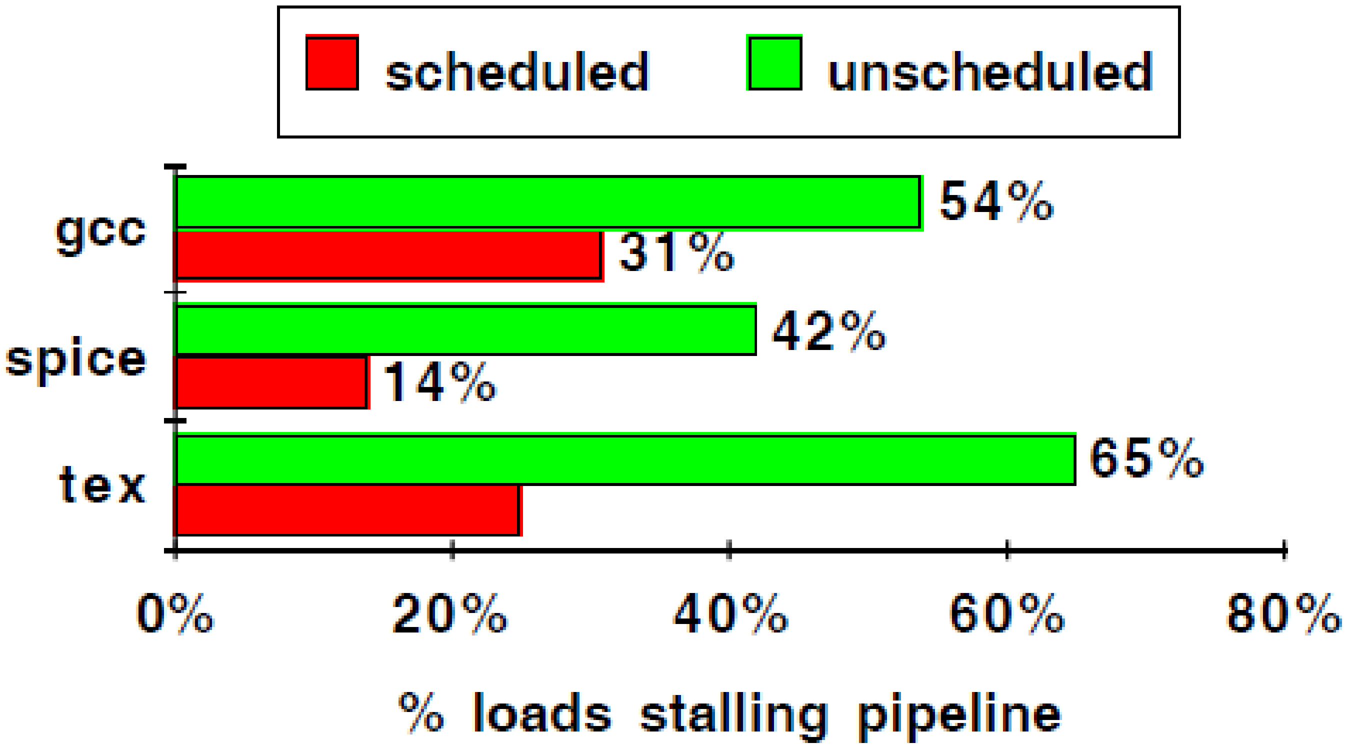
$$d = e - f;$$

assuming a, b, c, d ,e, and f in memory.

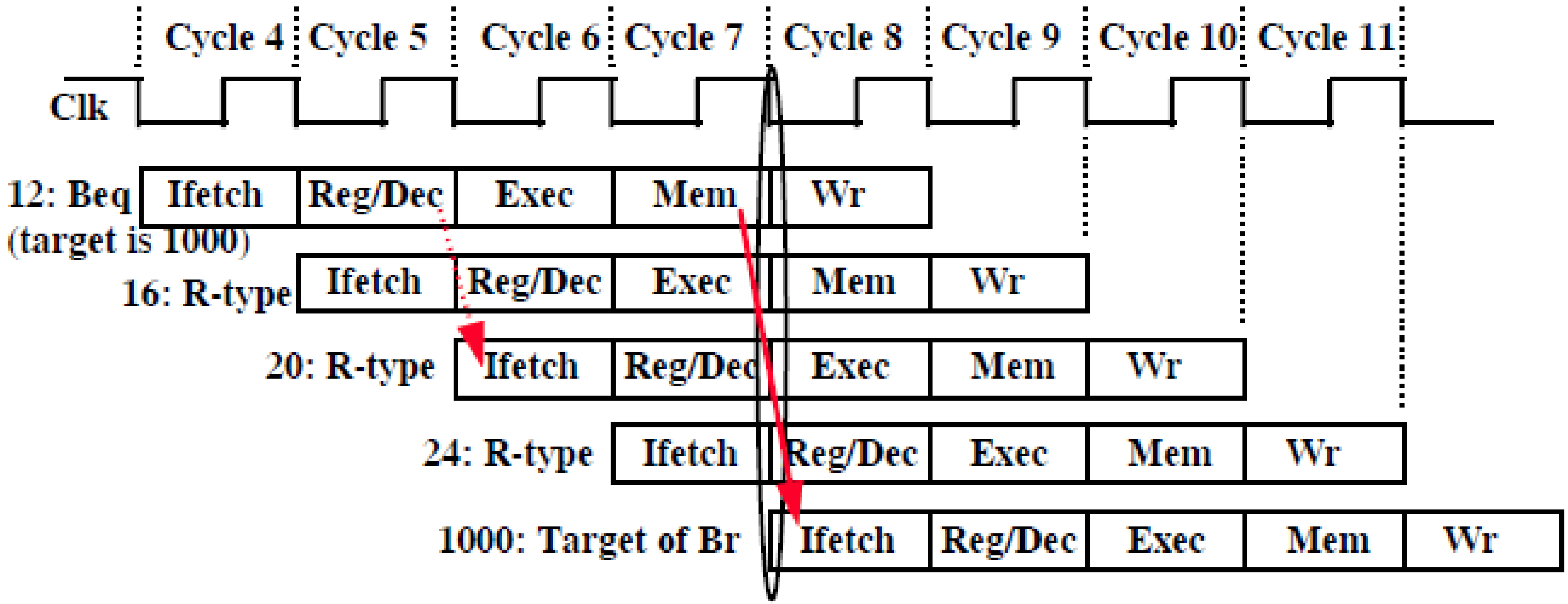
Slow code:

```
LW      Rb,b
LW      Rc,c
ADD     Ra,Rb,Rc
SW      a,Ra
LW      Re,e
LW      Rf,f
SUB     Rd,Re,Rf
SW      d,Rd
```

Compiler Avoiding Load Stalls:

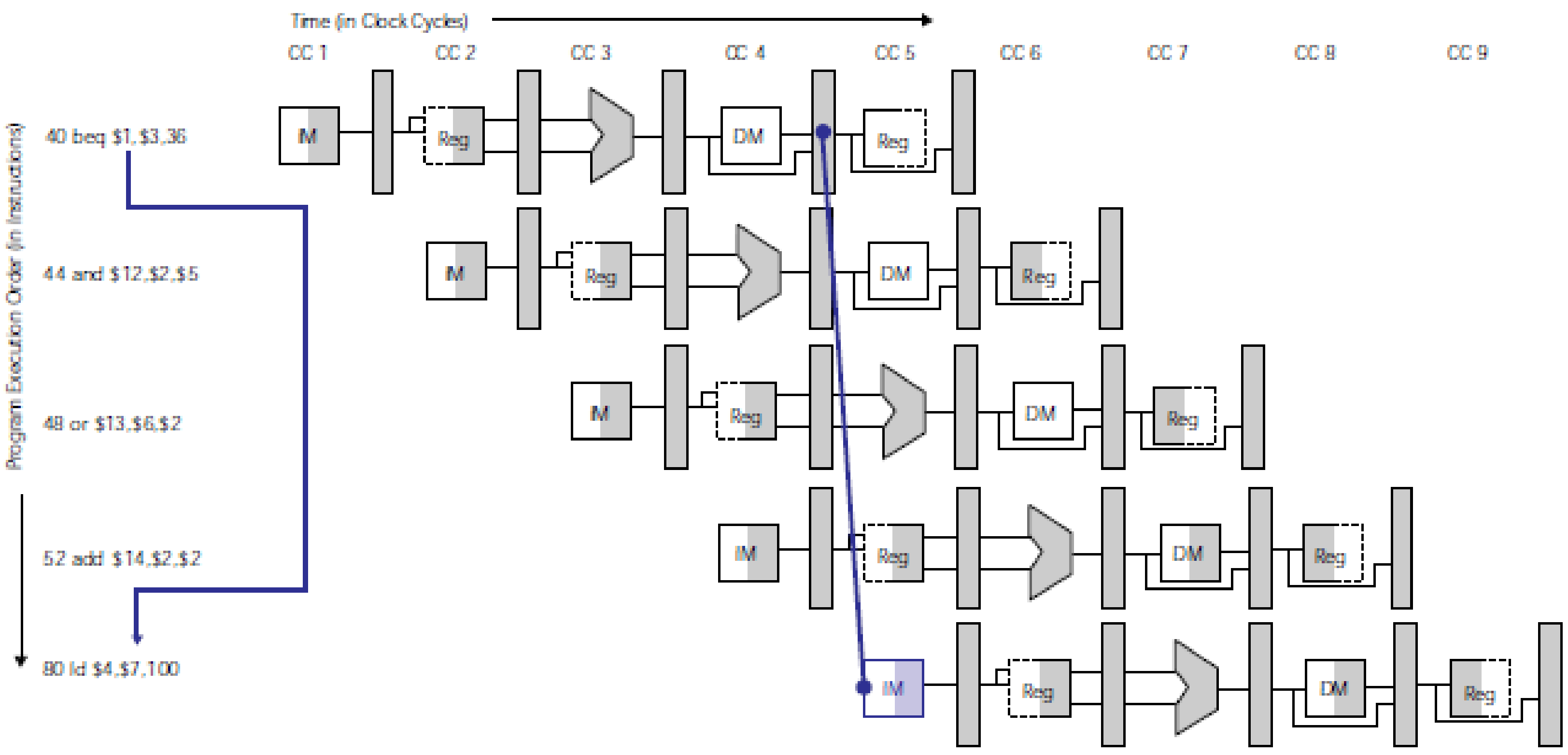


From Last Lecture: The Delay Branch Phenomenon



- Although Beq is fetched during Cycle 4:
  - Target address is NOT written into the PC until the end of Cycle 7
  - Branch's target is NOT fetched until Cycle 8
  - 3-instruction delay before the branch take effect

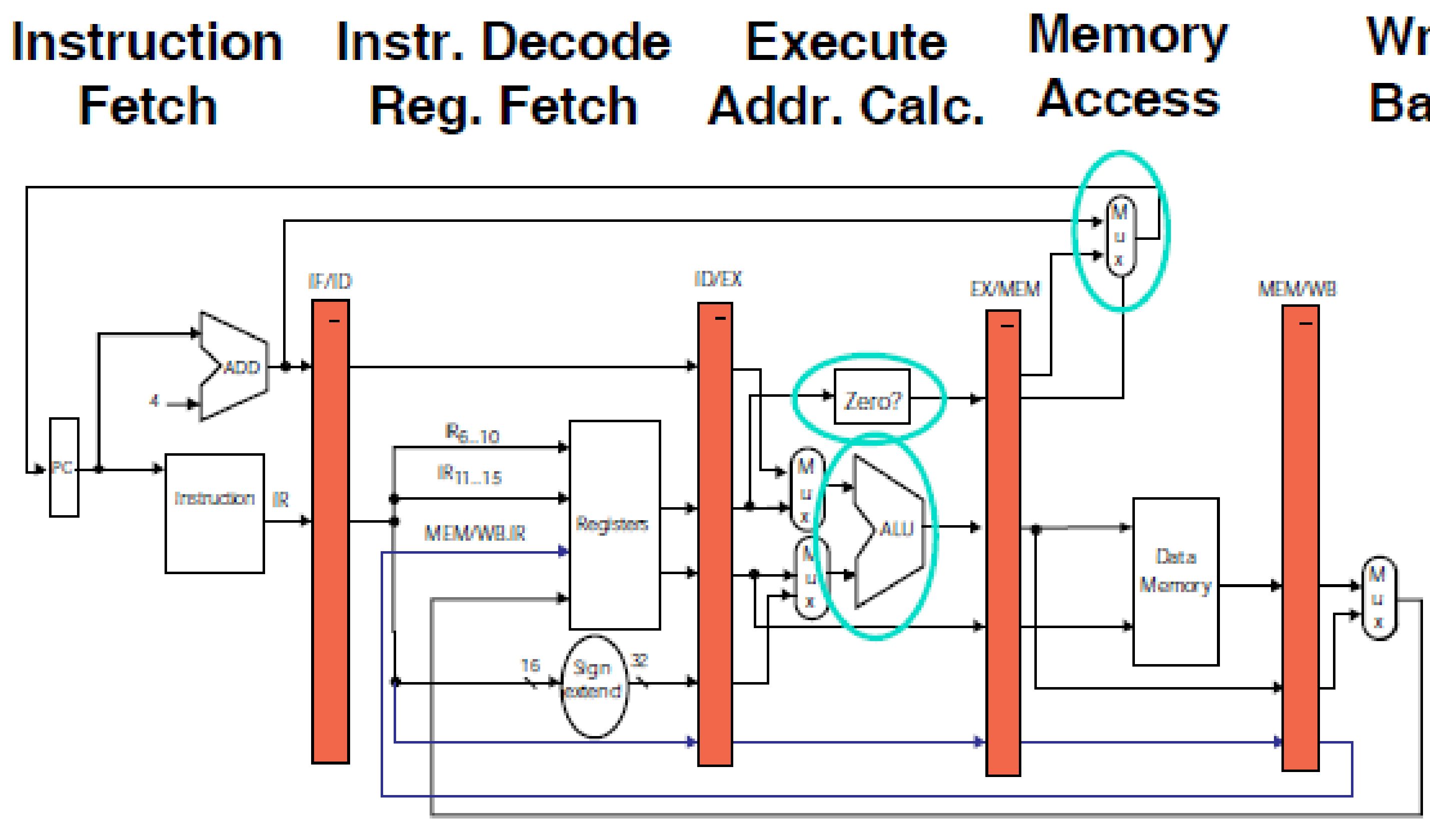
Control Hazard on Branches: 3 stage stall



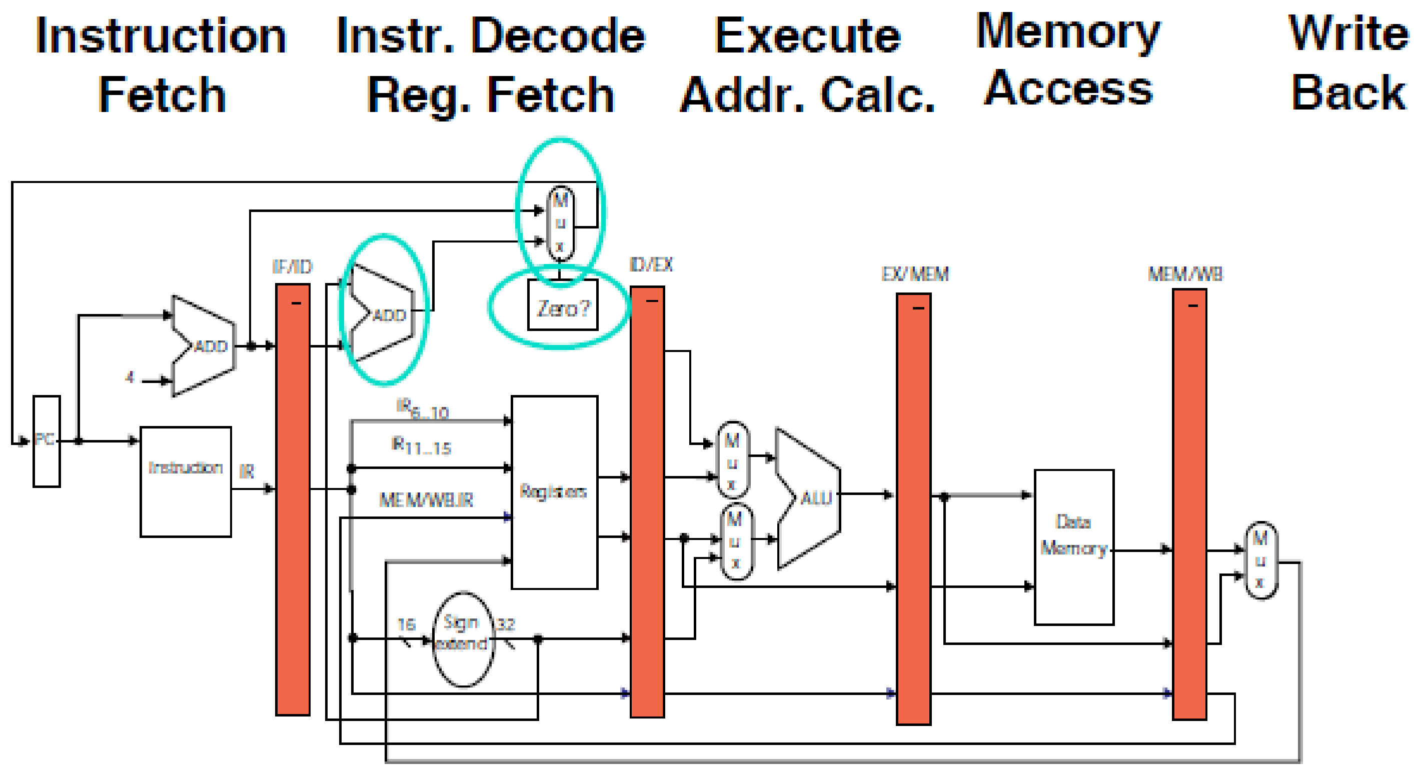
Branch Stall Impact

- If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!
- 2 part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests = 0 or ≠ 0
- Solution Option 1:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch vs. 3

Option 1: move HW forward to reduce branch delay



Branch Delay now 1 clock cycle



Option 2: Define Branch as Delayed

- Worst case, SW inserts NOP into branch delay
- Where get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: only valuable when branch
  - From fall through: only valuable when don't branch
- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - about 50% (60% x 80%) of slots usefully filled

When is pipelining hard?

- Interrupts: 5 instructions executing in 5 stage pipeline
  - How to stop the pipeline?
  - Restrart?
  - Who caused the interrupt?

Stage	*Problem interrupts occurring*
IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic interrupt
MEM	Page fault on data fetch; misaligned memory access; memory-protection violation

- Load with data page fault, Add with instruction page fault?
- Solution 1: interrupt vector/instruction, check last stage
- Solution 2: interrupt ASAP, restart everything incomplete

When is pipelining hard?

- Complex Addressing Modes and Instructions
- Address modes: Autoincrement causes register change during instruction execution
  - Interrupts?
  - Now worry about write hazards since write no longer last stage
    - Write After Read (WAR): Write occurs before independent read
    - Write After Write (WAW): Writes occur in wrong order, leaving wrong result in registers
    - (Previous data hazard called RAW, for Read After Write)
- Memory-memory Move instructions
  - Multiple page faults
  - make progress?

When is pipelining hard?

- Floating Point: long execution time
- Also, may pipeline FP execution unit so that can initiate new instructions without waiting full latency

FP Instruction	*Latency*	*Initiation Rate*	*(MIPS R4000)*
Add, Subtract	4	3	
Multiply	8	4	
Divide	36	35	
Square root	112	111	
Negate	2	1	
Absolute value	2	1	
FP compare	3	2	

- Divide, Square Root take ≈10X to ≈30X longer than Add
  - Exceptions?
  - Adds WAR and WAW hazards since pipelines are no longer same length

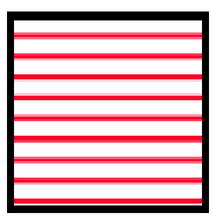
Hazard Detection

Suppose instruction *i* is about to be issued and a predecessor instruction *j* is in the instruction pipeline.

Rregs (*i*) = Registers read by instruction *i*

Wregs (*i*) = Registers written by instruction *i*

- A RAW hazard exists on register ρ if $\exists\, \rho,\ \rho \in$ Rregs(*i*) $\cap$ Wregs(*j*)
  - Keep a record of pending writes (for inst's in the pipe) and compare with operand regs of current instruction.
  - When instruction issues, reserve its result register.
  - When on operation completes, remove its write reservation.



- A WAW hazard exists on register ρ if $\exists\, \rho,\ \rho \in$ Wregs(*i*) $\cap$ Wregs(*j*)

- A WAR hazard exists on register ρ if $\exists\, \rho,\ \rho \in$ Wregs(*i*) $\cap$ Rregs(*j*)

First Generation RISC Pipelines

- All instructions follow same pipeline order (“static schedule”).
- Register write in last stage
  - Avoid WAW hazards
- All register reads performed in first stage after issue.
  - Avoid WAR hazards
- Memory access in stage 4
  - Avoid all memory hazards
- Control hazards resolved by delayed branch (with fast path)
- RAW hazards resolved by bypass, except on load results

which are resolved by fiat (delayed load).

Substantial pipelining with very little cost or complexity.

Machine organization is (slightly) exposed!

Relies very heavily on "hit assumption"of memory accesses in cache

Review: Summary of Pipelining Basics

- Speed Up $\leq$ Pipeline Depth; if ideal CPI is 1, then:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

- Hazards limit performance on computers:
  - structural: need more HW resources
  - data: need forwarding, compiler scheduling
  - control: early evaluation & PC, delayed branch, prediction
- Increasing length of pipe increases impact of hazards since pipelining helps instruction bandwidth, not latency
- Compilers key to reducing cost of data and control hazards
  - load delay slots
  - branch delay slots
- Exceptions, Instruction Set, FP makes pipelining harder
- Longer pipelines => Branch prediction, more instruction parallelism?

CS252
Graduate Computer Architecture
Lecture 6

Scoreboard, Tomasulo, Register Renaming
February 7th, 2011

John Kubiatowicz

Electrical Engineering and Computer Sciences

University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs252

Recall: Revised FP Loop Minimizing Stalls

```
1 Loop: LD    F0,0(R1)
2       stall
3       ADDD  F4,F0,F2
4       SUBI  R1,R1,8
5       BNEZ  R1,Loop    ;delayed branch
6       SD    8(R1),F4   ;altered when move past SUBI
```

Swap BNEZ and SD by changing address of SD

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

6 clocks: Unroll loop 4 times code to make faster?

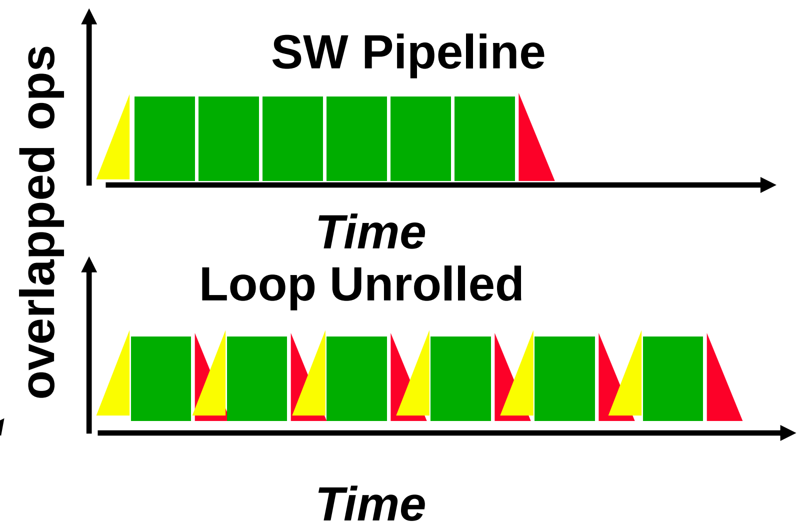
Recall: Software Pipelining Example

Before: Unrolled 3 times

```
1   LD    F0,0(R1)
2   ADDD  F4,F0,F2
3   SD    0(R1),F4
4   LD    F6,-8(R1)
5   ADDD  F8,F6,F2
6   SD    -8(R1),F8
7   LD    F10,-16(R1)
8   ADDD  F12,F10,F2
9   SD    -16(R1),F12
10  SUBI  R1,R1,#24
11  BNEZ  R1,LOOP
```

After: Software Pipelined

```
1   SD    0(R1),F4 ;  Stores M[i]
2   ADDD  F4,F0,F2 ;  Adds to M[i-1]
3   LD    F0,-16(R1); Loads M[i-2]
4   SUBI  R1,R1,#8
5   BNEZ  R1,LOOP
```



- **Symbolic Loop Unrolling**
  - Maximize result-use distance
  - Less code space than unrolling
  - Fill & drain pipe only once per loop vs. once per each unrolled iteration in loop unrolling

5 cycles per iteration

Can we use HW to get CPI closer to 1?

- **Why in HW at run time?**
  - **Works when can't know real dependence at compile time**
  - **Compiler simpler**
  - **Code for one machine runs well on another**

- **Key idea: Allow instructions behind stall to proceed**

```
DIVD   F0,F2,F4
ADDD   F10,F0,F8
SUBD   F12,F8,F14
```

- **Out-of-order execution => out-of-order completion.**

Problems?

- **How do we prevent WAR and WAW hazards?**
- **How do we deal with variable latency?**
  - **Forwarding for RAW hazards harder.**

	Clock Cycle Number																
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD F6,34(R2)	IF	ID	EX	MEM	WB												
LD F2,45(R3)		IF	ID	EX	MEM	WB											
MULTD F0,F2,F4			IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	MEM	WB
SUBD F8,F6,F2				IF	ID	A1	A2	MEM	WB								
DIVD F10,F0,F6					IF	ID	stall	stall	stall	stall	stall	stall	stall	stall	stall	D1	D2
ADDD F6,F8,F2						IF	ID	A1	A2	MEM	WB						

RAW

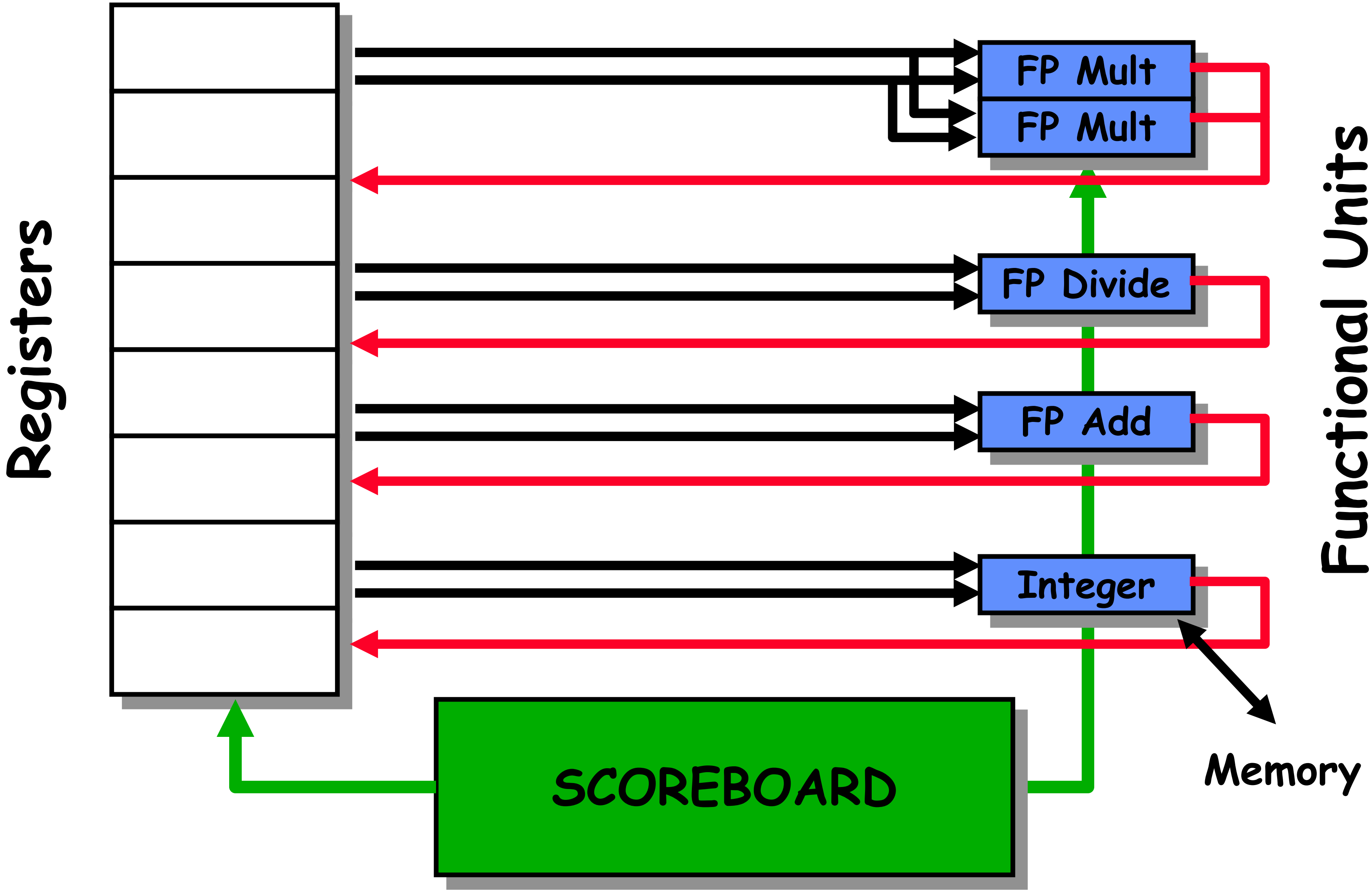
WAR

- **How to get precise exceptions?**

Scoreboard: a bookkeeping technique

- **Out-of-order execution divides ID stage:**
  1. **Issue—decode instructions, check for structural hazards**
  2. **Read operands—wait until no data hazards, then read operands**
- **Scoreboards date to CDC6600 in 1963**
  - **Readings for Monday include one on CDC6600**
- **Instructions execute whenever not dependent on previous instructions and no hazards.**
- **CDC 6600: In order issue, out-of-order execution, out-of-order commit (or completion)**
  - **No forwarding!**
  - **Imprecise interrupt/exception model for now**

Scoreboard Architecture (CDC 6600)



Scoreboard Implications

- Out-of-order completion => WAR, WAW hazards?
- Solutions for WAR:
  - Stall writeback until registers have been read
  - Read registers only during Read Operands stage
- Solution for WAW:
  - Detect hazard and stall issue of new instruction until other instruction completes
- No register renaming (next time)
- Need to have multiple instructions in execution phase => multiple execution units or pipelined execution units
- Scoreboard keeps track of dependencies between instructions that have already issued.
- Scoreboard replaces ID, EX, WB with 4 stages

Four Stages of Scoreboard Control

- **Issue—decode instructions & check for structural hazards (ID1)**
  - **Instructions issued in program order (for hazard checking)**
  - **Don’t issue if structural hazard**
  - **Don’t issue if instruction is output dependent on any previously issued but uncompleted instruction (no WAW hazards)**
- **Read operands—wait until no data hazards, then read operands (ID2)**
  - **All real dependencies (RAW hazards) resolved in this stage, since we wait for instructions to write back data.**
  - **No forwarding of data in this model!**

Four Stages of Scoreboard Control

- **Execution—operate on operands (EX)**
  - The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution.
- **Write result—finish execution (WB)**
  - Stall until no WAR hazards with previous instructions:

Example:

```
DIVD   F0,F2,F4
ADDD   F10,F0,F8
SUBD   F8,F8,F14
```

CDC 6600 scoreboard would stall SUBD until ADDD reads operands

Three Parts of the Scoreboard

- **Instruction status:**
  Which of 4 steps the instruction is in

- **Functional unit status:—Indicates the state of the functional unit (FU). 9 fields for each functional unit**
  - **Busy:** Indicates whether the unit is busy or not
  - **Op:** Operation to perform in the unit (e.g., + or –)
  - **Fi:** Destination register
  - **Fj,Fk:** Source-register numbers
  - **Qj,Qk:** Functional units producing source registers Fj, Fk
  - **Rj,Rk:** Flags indicating when Fj, Fk are ready

- **Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions will write that register**

Scoreboard Example

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2				
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
	FU									

Detailed Scoreboard Pipeline Control

Instruction status	Wait until	Bookkeeping
Issue	Not busy (FU) and not result(D)	Busy(FU)← yes; Op(FU)← op; Fi(FU)← `D'; Fj(FU)← `S1'; Fk(FU)← `S2'; Qj← Result('S1'); Qk← Result(`S2'); Rj← not Qj; Rk← not Qk; Result('D')← FU;
Read operands	Rj and Rk	Rj← No; Rk← No
Execution complete	Functional unit done	
Write result	∀f((Fj(f)≠Fi(FU) or Rj(f)=No) & (Fk(f)≠Fi(FU) or Rk(f)=No))	∀f(if Qj(f)=FU then Rj(f)← Yes); ∀f(if Qk(f)=FU then Rj(f)← Yes); Result(Fi(FU))← 0; Busy(FU)← No

Scoreboard Example: Cycle 1

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1			
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock

1

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU				Integer					

Scoreboard Example: Cycle 2

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2		
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	*FU*				Integer					

- **Issue 2nd LD?**

Scoreboard Example: Cycle 3

Instruction status:

Instruction		j	k	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	Yes	Load	F6		R2				No
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
3	*FU*				Integer					

- **Issue MULT?**

Scoreboard Example: Cycle 4

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	*FU*				Integer					

Scoreboard Example: Cycle 5

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5			
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	*FU*		Integer							

Scoreboard Example: Cycle 6

Instruction status:

Instruction		j	k	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6		
MULTD	F0	F2	F4	6			
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	Yes	Load	F2		R3				Yes
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
6	*FU*	Mult1	Integer							

Scoreboard Example: Cycle 7

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	
MULTD	F0	F2	F4	6			
SUBD	F8	F6	F2	7			
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				No
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	*FU*	Mult1	Integer			Add				

- **Read multiply operands?**

Scoreboard Example: Cycle 8a
(First half of clock cycle)

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	
MULTD	F0	F2	F4	6			
SUBD	F8	F6	F2	7			
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	Yes	Load	F2		R3				No
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
8	*FU*	Mult1	Integer			Add	Divide			

Scoreboard Example: Cycle 8b
(Second half of clock cycle)

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6			
SUBD	F8	F6	F2	7			
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
8	*FU*	Mult1				Add	Divide			

Scoreboard Example: Cycle 9

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9		
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Note Remaining →

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
10	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
2	Add	Yes	Sub	F8	F6	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
9	*FU*	Mult1				Add	Divide			

- **Read operands for MULT & SUB? Issue ADDD?**

Scoreboard Example: Cycle 10

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9		
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
9	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
1	Add	Yes	Sub	F8	F6	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
10	*FU*	Mult1				Add	Divide			

Scoreboard Example: Cycle 11

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
8	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
0	Add	Yes	Sub	F8	F6	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
11	*FU*	Mult1				Add	Divide			

Scoreboard Example: Cycle 12

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
7	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	No								
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
12	*FU*	Mult1					Divide			

- **Read operands for DIVD?**

Scoreboard Example: Cycle 13

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13			

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
6	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
13	*FU*	Mult1			Add		Divide			

Scoreboard Example: Cycle 14

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14		

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
5	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
2	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
14	*FU*	Mult1			Add		Divide			

Scoreboard Example: Cycle 15

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14		

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
4	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
1	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
15	*FU*	Mult1			Add		Divide			

Scoreboard Example: Cycle 16

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
3	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
0	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
16	FU	Mult1			Add		Divide			

Scoreboard Example: Cycle 17

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

WAR Hazard!

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
2	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
17	FU	Mult1			Add		Divide			

- Why not write result of ADD???

Scoreboard Example: Cycle 18

Instruction status:

Instruction		j	k	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
1	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
18	*FU*	Mult1			Add		Divide			

Scoreboard Example: Cycle 19

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
0	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
19	*FU*	Mult1			Add		Divide			

Scoreboard Example: Cycle 20

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6			Yes	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
20	*FU*				Add		Divide			

Scoreboard Example: Cycle 21

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21		
ADDD	F6	F8	F2	13	14	16	

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6			Yes	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
21	*FU*				Add		Divide			

- **WAR Hazard is now gone...**

Scoreboard Example: Cycle 22

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21		
ADDD	F6	F8	F2	13	14	16	22

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
39	Divide	Yes	Div	F10	F0	F6			No	No

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
22	*FU*						Divide			

Faster than light computation (skip a couple of cycles)

Scoreboard Example: Cycle 61

Instruction status:

Instruction		j	k	Read Oper Issue	Exec Comp Oper	Write Result Comp	Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	
ADDD	F6	F8	F2	13	14	16	22

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
0	Divide	Yes	Div	F10	F0	F6			No	No

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
61	FU						Divide			

Scoreboard Example: Cycle 62

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	62
ADDD	F6	F8	F2	13	14	16	22

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
62	*FU*									

Review: Scoreboard Example: Cycle 62

Instruction status:

Instruction		j	k	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	62
ADDD	F6	F8	F2	13	14	16	22

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
62	*FU*									

- **In-order issue; out-of-order execute & commit**

CDC 6600 Scoreboard

- Speedup 1.7 from compiler; 2.5 by hand BUT slow memory (no cache) limits benefit
- Limitations of 6600 scoreboard:
  - No forwarding hardware
  - Limited to instructions in basic block (small *window*)
  - Small number of functional units (structural hazards), especially integer/load store units
  - Do not issue on structural hazards
  - Wait for WAR hazards
  - Prevent WAW hazards

CS 252 Administrivia

- **Interesting Resource: http://bitsavers.org**
  - Has digital versions of users manuals for old machines
  - Quite interesting!
  - I'll link in some of them to your reading pages when it is appropriate
  - Very limited bandwidth: use mirrors such as: http://bitsavers.vt100.net
- **Textbook Reading for next few lectures:**
  - Computer Architecture: A Quantitative Approach, Chapter 2
- **Midterm I: March 16th**
  - Currently exam on 16th is scheduled for 2:30-5:30. Would this work for people? Room is available up until 7:00, so could do 4:00-7:00.
  - Pizza afterwards…

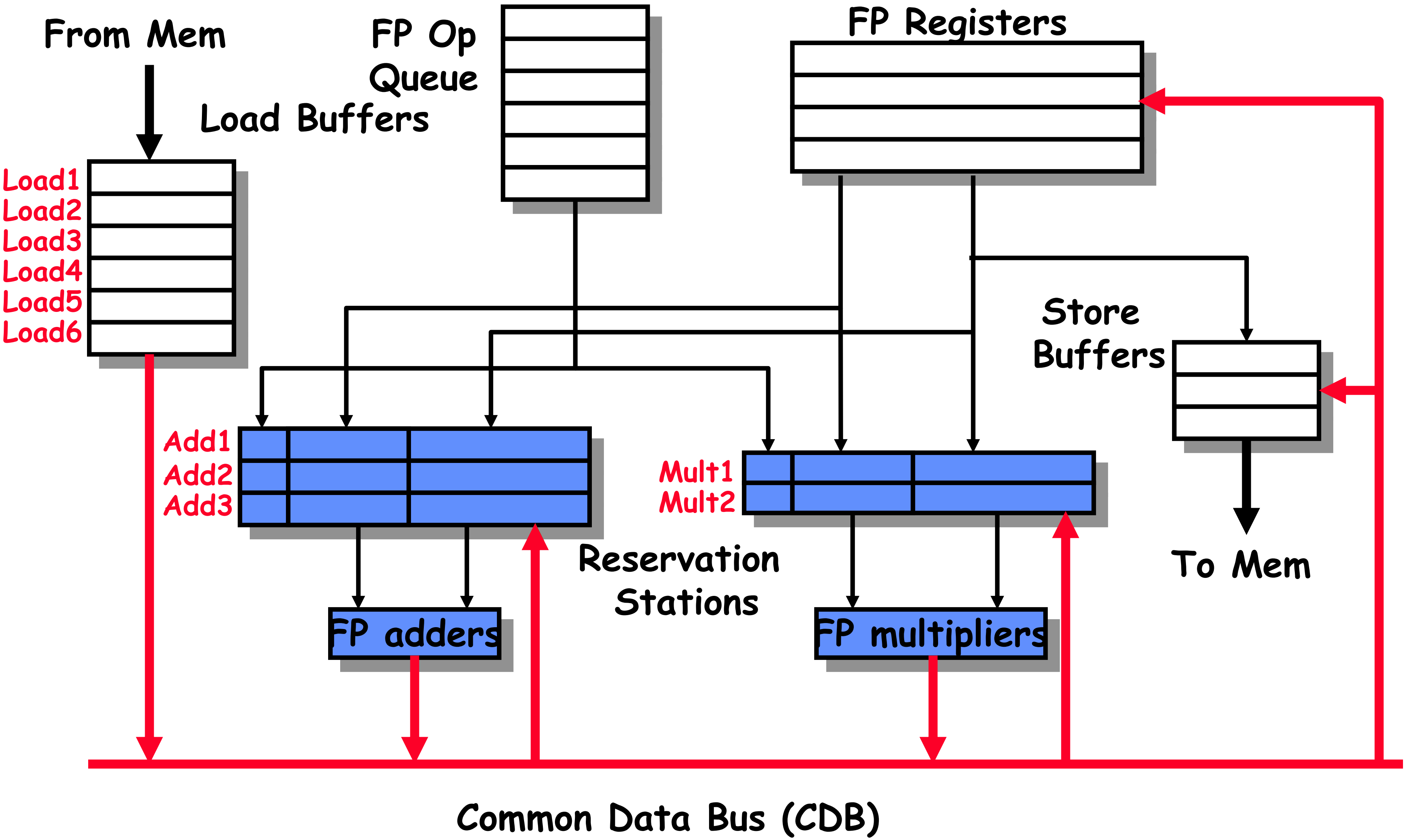
Paper Discussion (Reading #4)

- "Parallel Operation in the CDC 6600," James E. Thorton. *AFIPS Proc*. FJCC, pt. 2 vol. 03, pp. 33-40, 1964
  - Pushed the Load-Store architecture that became staple of RISC
  - Scoreboard for OOO execution (last lecture)
  - Separation of I/O processors from main processor
    - » Memory-mapped communication between them
    - » Very modern ideas
- "The CRAY-1 Computer System," Richard Russel. *Communications of the ACM*, 21(1) 63-72, January 1978
  - Very successful Vector Machine
  - Highly tuned physical implementation
  - No Virtual Memory: segmented protection + relocatable code

Another Dynamic Algorithm: Tomasulo Algorithm

- For IBM 360/91 about 3 years after CDC 6600 (1966)
- Goal: High Performance without special compilers
- Differences between IBM 360 & CDC 6600 ISA
  - IBM has only 2 register specifiers/instr vs. 3 in CDC 6600
  - IBM has 4 FP registers vs. 8 in CDC 6600
  - IBM has memory-register ops
- Why Study? lead to Alpha 21264, HP 8000, MIPS 10000, Pentium II, PowerPC 604, …

Tomasulo Organization



Tomasulo Algorithm vs. Scoreboard

- **Control & buffers distributed with Function Units (FU) vs. centralized in scoreboard;**
  - **FU buffers called "reservation stations"; have pending operands**
- **Registers in instructions replaced by values or pointers to reservation stations(RS); called register renaming ;**
  - **avoids WAR, WAW hazards**
  - **More reservation stations than registers, so can do optimizations compilers can't**
- **Results to FU from RS, not through registers, over Common Data Bus that broadcasts results to all FUs**
- **Load and Stores treated as FUs with RSs as well**
- **Integer instructions can go past branches, allowing FP ops beyond basic block in FP queue**

Reservation Station Components

- **Op:** Operation to perform in the unit (e.g., + or –)
- **Vj, Vk:** **Value** of Source operands
  - Store buffers has V field, result to be stored
- **Qj, Qk:** Reservation stations producing source registers (value to be written)
  - Note: No ready flags as in Scoreboard; Qj,Qk=0 => ready
  - Store buffers only have Qi for RS producing result
- **Busy:** Indicates reservation station or FU is busy

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Three Stages of Tomasulo Algorithm

1. **Issue—get instruction from FP Op Queue**

   If reservation station free (no structural hazard), control issues instr & sends operands (renames registers).

2. **Execution—operate on operands (EX)**

   When both operands ready then execute; if not ready, watch Common Data Bus for result

3. **Write result—finish execution (WB)**

   Write on Common Data Bus to all awaiting units; mark reservation station available

- **Normal data bus: data + destination ("go to" bus)**
- **Common data bus: data + source ("come from" bus)**
  - **64 bits of data + 4 bits of Functional Unit source address**
  - **Write if matches expected Functional Unit (produces result)**
  - **Does the broadcast**

Tomasulo Example

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
0	*FU*									

Tomasulo Example Cycle 1

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
1	*FU*				Load1					

Tomasulo Example Cycle 2

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3	2		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	FU		Load2		Load1					

Note: Unlike 6600, can have multiple loads outstanding

Tomasulo Example Cycle 3

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU	Mult1	Load2		Load1					

- Note: registers names are removed ("renamed") in Reservation Stations; MULT issued vs. scoreboard
- Load1 completing; what is waiting for Load1?

Tomasulo Example Cycle 4

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	Yes	SUBD	M(A1)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
4	*FU*	Mult1	Load2		M(A1)	Add1				

- **Load2 completing; what is waiting for Load2?**

Tomasulo Example Cycle 5

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU	Mult1	M(A2)		M(A1)	Add1	Mult2			

Tomasulo Example Cycle 6

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
6	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

- **Issue ADDD here vs. scoreboard?**

Tomasulo Example Cycle 7

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

- Add1 completing; what is waiting for it?

Tomasulo Example Cycle 8

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
2	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
8	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 9

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
9	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 10

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
10	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

- **Add2 completing; what is waiting for it?**

Tomasulo Example Cycle 11

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
11	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- **Write result of ADDD here vs. scoreboard?**
- **All quick instructions complete in this cycle!**

Tomasulo Example Cycle 12

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
12	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 13

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
13	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 14

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
14	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 15

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
15	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 16

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
16	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

Faster than light computation (skip a couple of cycles)

Tomasulo Example Cycle 55

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
1	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
55	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 56

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
56	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- **Mult2 is completing; what is waiting for it?**

Tomasulo Example Cycle 57

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	57
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
56	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Result			

- **Once again: In-order issue, out-of-order execution and completion.**

Compare to Scoreboard Cycle 62

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4	1	3	4
LD	F2	45+	R3	5	6	7	8	2	4	5
MULTD	F0	F2	F4	6	9	19	20	3	15	16
SUBD	F8	F6	F2	7	9	11	12	4	7	8
DIVD	F10	F0	F6	8	21	61	62	5	56	57
ADDD	F6	F8	F2	13	14	16	22	6	10	11

- **Why take longer on scoreboard/6600?**
  - **Structural Hazards**
  - **Lack of forwarding**

Tomasulo v. Scoreboard (IBM 360/91 v. CDC 6600)

Pipelined Functional Units (6 load, 3 store, 3 +, 2 x/÷)	**Multiple Functional Units (1 load/store, 1 + , 2 x, 1 ÷)**
window size: ≤ 14 instructions	**≤ 5 instructions**
No issue on structural hazard	**same**
WAR: renaming avoids	**stall completion**
WAW: renaming avoids	**stall issue**
Broadcast results from FU	**Write/read registers**
Control: reservation stations	**central scoreboard**

Recall: Unrolled Loop That Minimizes Stalls

```
Loop:LD     F0,0(R1)
     LD     F6,-8(R1)
     LD     F10,-16(R1)
     LD     F14,-24(R1)
     ADDD   F4,F0,F2
     ADDD   F8,F6,F2
     ADDD   F12,F10,F2
     ADDD   F16,F14,F2
     SD     0(R1),F4
     SD     -8(R1),F8
     SD     -16(R1),F12
     SUBI   R1,R1,#32
     BNEZ   R1,LOOP
     SD     8(R1),F16      ; 8-32 = -24
```

- **What assumptions made when moved code?**
  - OK to move store past SUBI even though changes register
  - OK to move loads before stores: get right data?
  - When is it safe for compiler to do such changes?

14 clock cycles, or 3.5 per iteration

Tomasulo Loop Example

```
Loop: LD      F0   0     R1
      MULTD   F4   F0    F2
      SD      F4   0     R1
      SUBI    R1   R1    #8
      BNEZ    R1   Loop
```

- Assume Multiply takes 4 clocks
- Assume first load takes 8 clocks (cache miss), second load takes 1 clock (hit)
- To be clear, will show clocks for SUBI, BNEZ
- Reality: integer instructions ahead

Loop Example

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1			
1	MULTD	F4	F0	F2			
1	SD	F4	0	R1			
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
0	**80**	*Fu*									

Loop Example Cycle 1

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2			
1	SD	F4	0	R1			
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
1	**80**	*Fu*	Load1								

Loop Example Cycle 2

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1			
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	Addr	Fu
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F4)	Load1	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
2	80	Fu	Load1		Mult1						

Loop Example Cycle 3

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F4)	Load1	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
3	**80**	*Fu*	Load1		Mult1						

- **Implicit renaming sets up “DataFlow” graph**

Loop Example Cycle 4

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F4)	Load1	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
4	**80**	*Fu*	Load1		Mult1						

- **Dispatching SUBI Instruction**

Loop Example Cycle 5

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1			
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	Addr	Fu
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F4)	Load1	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
5	72	Fu	Load1		Mult1						

- **And, BNEZ instruction**

Loop Example Cycle 6

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2			
2	SD	F4	0	R1			

	Busy	Addr	Fu
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F4)	Load1	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
6	72	Fu	Load2		Mult1						

- **Notice that F0 never sees Load from location 80**

Loop Example Cycle 7

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1			

	Busy	Addr	Fu
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
7	72	Fu	Load2		Mult2						

- **Register file completely detached from computation**
- **First and Second iteration completely overlapped**

Loop Example Cycle 8

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
8	**72**	*Fu*	Load2		Mult2						

Loop Example Cycle 9

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
9	72	Fu	Load2		Mult2						

- **Load1 completing: who is waiting?**
- **Note: Dispatching SUBI**

Loop Example Cycle 10

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	Multd	M[80]	R(F2)		
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
10	**64**	*Fu*	Load2		Mult2						

- **Load2 completing: who is waiting?**
- **Note: Dispatching BNEZ**

Loop Example Cycle 11

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	Multd	M[80]	R(F2)		
4	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
11	**64**	*Fu*	Load3		Mult2						

- **Next load in sequence**

Loop Example Cycle 12

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	Multd	M[80]	R(F2)		
3	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
12	**64**	*Fu*	Load3		Mult2						

- **Why not issue third multiply?**

Loop Example Cycle 13

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	Multd	M[80]	R(F2)		
2	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	*...*	*F30*
13	**64**	*Fu*	Load3		Mult2						

Loop Example Cycle 14

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result		Busy	Addr	Fu
1	LD	F0	0	R1	1	9	10	Load1	No		
1	MULTD	F4	F0	F2	2	14		Load2	No		
1	SD	F4	0	R1	3			Load3	Yes	64	
2	LD	F0	0	R1	6	10	11	Store1	Yes	80	Mult1
2	MULTD	F4	F0	F2	7			Store2	Yes	72	Mult2
2	SD	F4	0	R1	8			Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	Multd	M[80]	R(F2)		
1	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
14	64	Fu	Load3		Mult2						

- **Mult1 completing. Who is waiting?**

Loop Example Cycle 15

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
15	**64**	*Fu*	Load3		Mult2						

- **Mult2 completing. Who is waiting?**

Loop Example Cycle 16

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	[72]*R2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
16	64	Fu	Load3		Mult1						

Loop Example Cycle 17

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	[72]*R2
Store3	Yes	64	Mult1

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
17	**64**	*Fu*	Load3		Mult1						

Loop Example Cycle 18

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8		

	Busy	*Addr*	*Fu*
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	[72]*R2
Store3	Yes	64	Mult1

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1 Vk*	*S2 Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
18	**64**	*Fu*	Load3		Mult1						

Loop Example Cycle 19

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result		Busy	Addr	Fu
1	LD	F0	0	R1	1	9	10	Load1	No		
1	MULTD	F4	F0	F2	2	14	15	Load2	No		
1	SD	F4	0	R1	3	18	19	Load3	Yes	64	
2	LD	F0	0	R1	6	10	11	Store1	No		
2	MULTD	F4	F0	F2	7	15	16	Store2	Yes	72	[72]*R2
2	SD	F4	0	R1	8	19		Store3	Yes	64	Mult1

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
19	**64**	*Fu*	Load3		Mult1						

Loop Example Cycle 20

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*		*Busy*	*Addr*	*Fu*
1	LD	F0	0	R1	1	9	10	Load1	No		
1	MULTD	F4	F0	F2	2	14	15	Load2	No		
1	SD	F4	0	R1	3	18	19	Load3	Yes	64	
2	LD	F0	0	R1	6	10	11	Store1	No		
2	MULTD	F4	F0	F2	7	15	16	Store2	No		
2	SD	F4	0	R1	8	19	20	Store3	Yes	64	Mult1

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

LD	F0	0	R1
MULTD	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
20	**64**	*Fu*	Load3		Mult1						

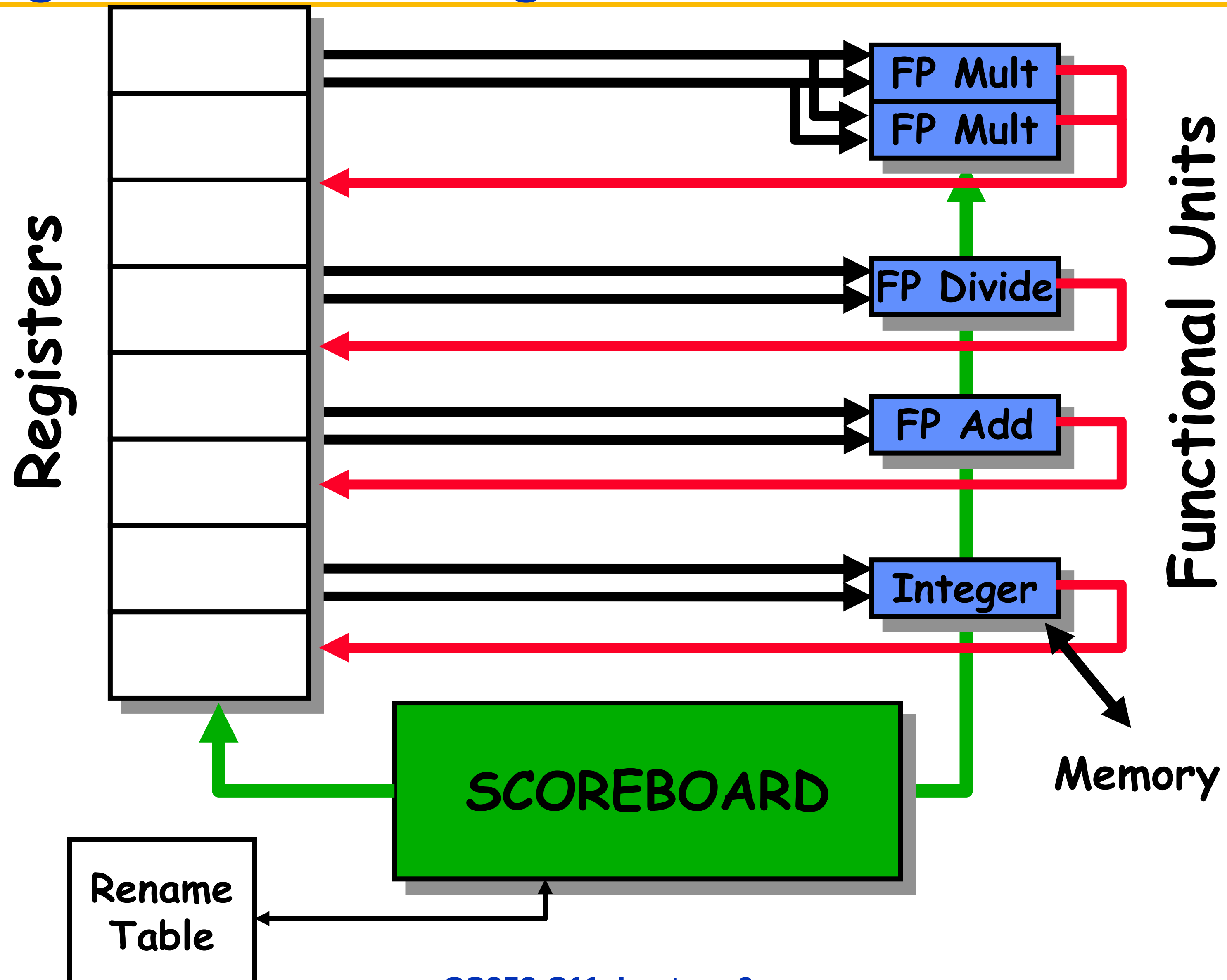
Why can Tomasulo overlap iterations of loops?

- **Register renaming**
  - Multiple iterations use different physical destinations for registers (dynamic loop unrolling).
- **Reservation stations**
  - Permit instruction issue to advance past integer control flow operations
- **Other idea: Tomasulo building dynamic "DataFlow" graph from instructions**
  - Fits in with readings for Wednesday

Explicit Register Renaming

- Tomasulo provides *Implicit Register Renaming*
  - User registers renamed to reservation station tags
- Explicit Register Renaming:
  - Use *physical* register file that is larger than number of registers specified by ISA
- Keep a translation table:
  - ISA register => physical register mapping
  - When register is written, replace table entry with new register from freelist.
  - Physical register becomes free when not being used by any instructions in progress.
- Pipeline can be exactly like "standard" DLX pipeline
  - IF, ID, EX, etc….
- Advantages:
  - Removes all WAR and WAW hazards
  - Like Tomasulo, good for allowing full out-of-order completion
  - Allows data to be fetched from a single register file
  - Makes speculative execution/precise interrupts easier:
    - » All that needs to be "undone" for precise break point is to undo the table mappings

Question: Can we use explicit register renaming with scoreboard?



Scoreboard Example

Instruction status:

Instruction		j	k	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2				
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Int1	No								
	Int2	No								
	Mult1	No								
	Add	No								
	Divide	No								

Register Rename and Result

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
	FU	P0	P2	P4	P6	P8	P10	P12		P30

- **Initialized Rename Table**

Renamed Scoreboard 1

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1			
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	Yes	Load	P32		R2				Yes
	Int2	No								
	Mult1	No								
	Add	No								
	Divide	No								

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
1	*FU*	P0	P2	P4	P32	P8	P10	P12		P30

- **Each instruction allocates free register**
- **Similar to single-assignment compiler transformation**

Renamed Scoreboard 2

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2		
LD	F2	45+	R3	2			
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	Yes	Load	P32		R2				Yes
	Int2	Yes	Load	P34		R3				Yes
	Mult1	No								
	Add	No								
	Divide	No								

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	*FU*	P0	P34	P4	P32	P8	P10	P12		P30

Renamed Scoreboard 3

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	
LD	F2	45+	R3	2	3		
MULTD	F0	F2	F4	3			
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	Yes	Load	P32		R2				Yes
	Int2	Yes	Load	P34		R3				Yes
	Mult1	Yes	Multd	P36	P34	P4	Int2		No	Yes
	Add	No								
	Divide	No								

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	*FU*	P36	P34	P4	P32	P8	P10	P12		P30

Renamed Scoreboard 4

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	
MULTD	F0	F2	F4	3			
SUBD	F8	F6	F2	4			
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Int1	No								
	Int2	Yes	Load	P34		R3				Yes
	Mult1	Yes	Multd	P36	P34	P4	Int2		No	Yes
	Add	Yes	Sub	P38	P32	P34		Int2	Yes	No
	Divide	No								

Register Rename and Result

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
4	*FU*	P36	P34	P4	P32	P38	P10	P12		P30

Renamed Scoreboard 5

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	5
MULTD	F0	F2	F4	3			
SUBD	F8	F6	F2	4			
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	No								
	Int2	No								
	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
	Add	Yes	Sub	P38	P32	P34			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	*FU*	P36	P34	P4	P32	P38	P40	P12		P30

Renamed Scoreboard 6

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	5
MULTD	F0	F2	F4	3	6		
SUBD	F8	F6	F2	4	6		
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	No								
	Int2	No								
10	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
2	Add	Yes	Sub	P38	P32	P34			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
6	*FU*	P36	P34	P4	P32	P38	P40	P12		P30

Renamed Scoreboard 7

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	5
MULTD	F0	F2	F4	3	6		
SUBD	F8	F6	F2	4	6		
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	No								
	Int2	No								
9	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
1	Add	Yes	Sub	P38	P32	P34			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	P36	P34	P4	P32	P38	P40	P12		P30

Renamed Scoreboard 8

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	5
MULTD	F0	F2	F4	3	6		
SUBD	F8	F6	F2	4	6	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	No								
	Int2	No								
8	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
0	Add	Yes	Sub	P38	P32	P34			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
8	*FU*	P36	P34	P4	P32	P38	P40	P12		P30

Renamed Scoreboard 9

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	5
MULTD	F0	F2	F4	3	6		
SUBD	F8	F6	F2	4	6	8	9
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	No								
	Int2	No								
7	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
	Add	No								
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
9	*FU*	P36	P34	P4	P32	P38	P40	P12		P30

Renamed Scoreboard 10

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	5
MULTD	F0	F2	F4	3	6		
SUBD	F8	F6	F2	4	6	8	9
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	10			

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	No								
	Int2	No								
6	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
	Add	Yes	Addd	P42	P38	P34			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

WAR Hazard gone!

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
10	*FU*	P36	P34	P4	P42	P38	P40	P12		P30

- Notice that P32 not listed in Rename Table
  - Still live. Must not be reallocated by accident

Renamed Scoreboard 11

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	5
MULTD	F0	F2	F4	3	6		
SUBD	F8	F6	F2	4	6	8	9
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	10	11		

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	No								
	Int2	No								
5	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
2	Add	Yes	Addd	P42	P38	P34			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
11	*FU*	P36	P34	P4	P42	P38	P40	P12		P30

Renamed Scoreboard 12

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	5
MULTD	F0	F2	F4	3	6		
SUBD	F8	F6	F2	4	6	8	9
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	10	11		

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	No								
	Int2	No								
4	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
1	Add	Yes	Addd	P42	P38	P34			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
12	FU	P36	P34	P4	P42	P38	P40	P12		P30

Renamed Scoreboard 13

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	5
MULTD	F0	F2	F4	3	6		
SUBD	F8	F6	F2	4	6	8	9
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	10	11	13	

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	No								
	Int2	No								
3	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
0	Add	Yes	Addd	P42	P38	P34			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
13	*FU*	P36	P34	P4	P42	P38	P40	P12		P30

Renamed Scoreboard 14

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	5
MULTD	F0	F2	F4	3	6		
SUBD	F8	F6	F2	4	6	8	9
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	10	11	13	14

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	No								
	Int2	No								
2	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
	Add	No								
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
14	FU	P36	P34	P4	P42	P38	P40	P12		P30

Renamed Scoreboard 15

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	5
MULTD	F0	F2	F4	3	6		
SUBD	F8	F6	F2	4	6	8	9
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	10	11	13	14

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	No								
	Int2	No								
1	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
	Add	No								
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
15	FU	P36	P34	P4	P42	P38	P40	P12		P30

Renamed Scoreboard 16

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	5
MULTD	F0	F2	F4	3	6	16	
SUBD	F8	F6	F2	4	6	8	9
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	10	11	13	14

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	No								
	Int2	No								
0	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
	Add	No								
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
16	*FU*	P36	P34	P4	P42	P38	P40	P12		P30

Renamed Scoreboard 17

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	5
MULTD	F0	F2	F4	3	6	16	17
SUBD	F8	F6	F2	4	6	8	9
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	10	11	13	14

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	No								
	Int2	No								
	Mult1	No								
	Add	No								
	Divide	Yes	Divd	P40	P36	P32	Mult1		Yes	Yes

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
17	*FU*	P36	P34	P4	P42	P38	P40	P12		P30

Renamed Scoreboard 18

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	5
MULTD	F0	F2	F4	3	6	16	17
SUBD	F8	F6	F2	4	6	8	9
DIVD	F10	F0	F6	5	18		
ADDD	F6	F8	F2	10	11	13	14

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	No								
	Int2	No								
	Mult1	No								
	Add	No								
40	Divide	Yes	Divd	P40	P36	P32	Mult1		Yes	Yes

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
18	*FU*	P36	P34	P4	P42	P38	P40	P12		P30

Explicit Renaming Support Includes:

- **Rapid access to a table of translations**
- **A physical register file that has more registers than specified by the ISA**
- **Ability to figure out which physical registers are free.**
  - **No free registers $\Rightarrow$ stall on issue**
- **Thus, register renaming doesn't require reservation stations. However:**
  - **Many modern architectures use explicit register renaming + Tomasulo-like reservation stations to control execution.**

Summary

- **Scoreboard: Track dependencies through reservations**
  - **Simple scheme for out-of-order execution**
  - **WAW and WAR hazards force stalls – cannot handle multiple instructions with same destination register**
- **Reservations stations: *renaming* to larger set of registers + buffering source operands**
  - **Prevents registers as bottleneck**
  - **Avoids WAR, WAW hazards of Scoreboard**
  - **Allows loop unrolling in HW**
- **Dynamic hardware schemes can unroll loops dynamically in hardware**
  - **Form of limited dataflow**
  - **Register renaming is essential**
- **Helps cache misses as well**

Summary #2

- **Lasting Contributions of Tomasulo Algorithm**
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- **360/91 descendants are Pentium II; PowerPC 604; MIPS R10000; HP-PA 8000; Alpha 21264**
- **Explicit Renaming: more physical registers than needed by ISA.**
  - Rename table: tracks current association between architectural registers and physical registers
  - Uses a translation table to perform compiler-like transformation on the fly
- **With Explicit Renaming:**
  - All registers concentrated in single register file
  - Can utilize bypass network that looks more like 5-stage pipeline
  - Introduces a register-allocation problem
    - » Need to handle branch misprediction and precise exceptions differently, but ultimately makes things simpler

Computer Architecture: Branch Prediction

Prof. Onur Mutlu
Carnegie Mellon University

A Note on This Lecture

- These slides are partly from 18-447 Spring 2013, Computer Architecture, Lecture 11: Branch Prediction

- Video of that lecture:
  - http://www.youtube.com/watch?v=XkerLktFtJg

Today's Agenda

- Branch prediction techniques

- Wrap up control dependence handling

Control Dependence Handling

Review: Branch Types

Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional	Unknown	2	Execution (register dependent)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Execution (register dependent)
Indirect	Always taken	Many	Execution (register dependent)

Different branch types can be handled differently

Review: How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:

- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:

  - Stall the pipeline until we know the next fetch address
  - Guess the next fetch address (branch prediction)
  - Employ delayed branching (branch delay slot)
  - Do something else (fine-grained multithreading)
  - Eliminate control-flow instructions (predicated execution)
  - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

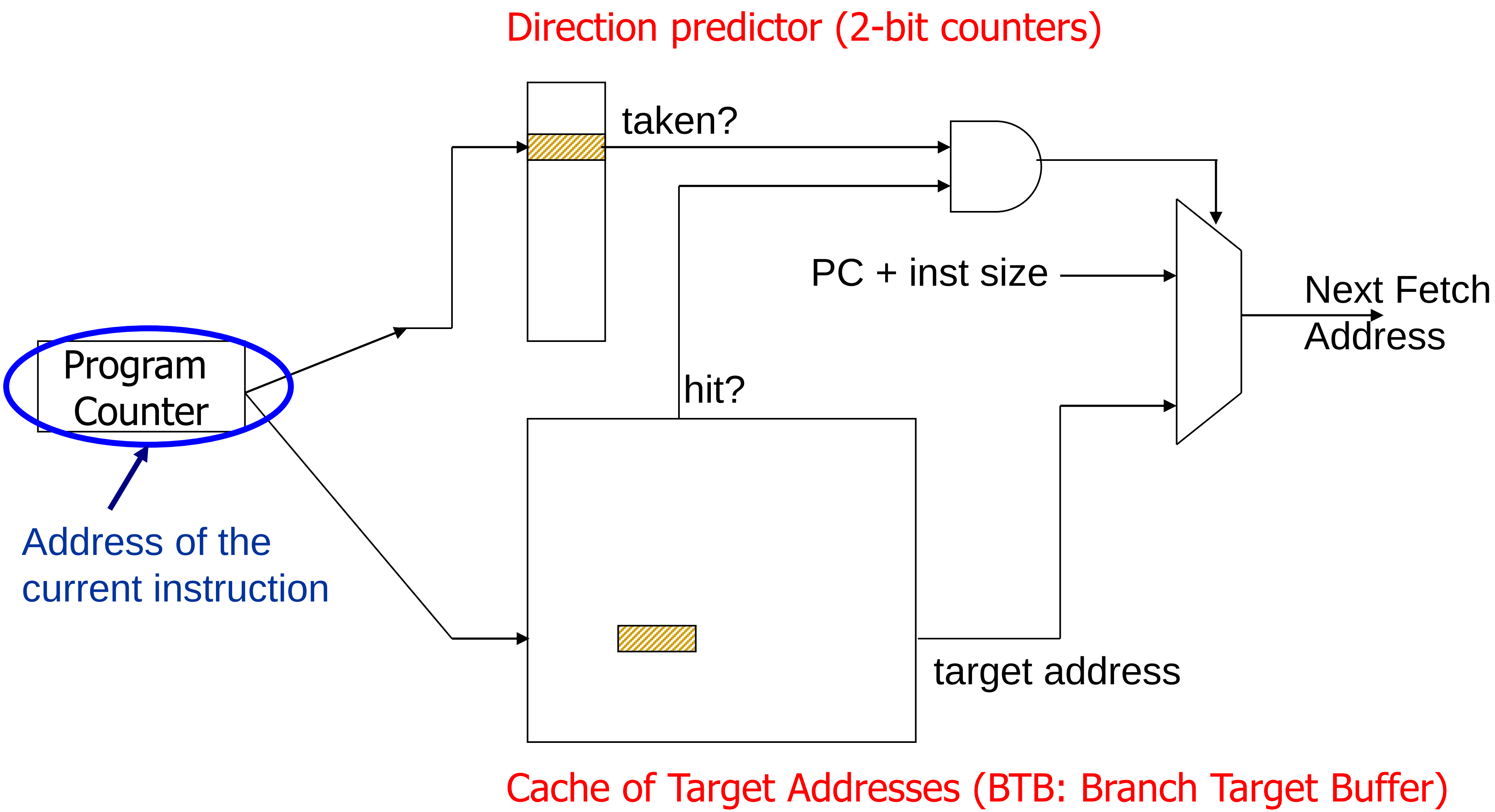
Review: Branch Prediction

- Idea: Predict the next fetch address (to be used in the next cycle)

- Requires three things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch
  - (Conditional) branch direction
  - Branch target address (if taken)

- Observation: Target address remains the same for a conditional direct branch across dynamic instances
  - Idea: Store the target address from previous instance and access it with the PC
  - Called Branch Target Buffer (BTB) or Branch Target Address Cache

Review: Fetch Stage with BTB



Always-taken CPI = [1 + (0.20*0.3) * 2] = 1.12 (70% of branches taken)

Simple Branch Direction Prediction Schemes

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)

More Sophisticated Direction Prediction

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid

Static Branch Prediction (I)

- Always not-taken
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40%
  - Compiler can layout code such that the likely path is the "not-taken" path

- Always taken
  - No direction prediction
  - Better accuracy: ~60-70%
    - Backward branches (i.e. loop branches) are usually taken
    - Backward branch: target address lower than branch PC

- Backward taken, forward not taken (BTFN)
  - Predict backward (loop) branches as taken, others not-taken

Static Branch Prediction (II)

- Profile-based
  - Idea: Compiler determines likely direction for each branch using profile run. Encodes that direction as a hint bit in the branch instruction format.

+ Per branch prediction (more accurate than schemes in previous slide) → accurate if profile is representative!

-- Requires hint bits in the branch instruction format

-- Accuracy depends on dynamic branch behavior:

TTTTTTTTTTNNNNNNNNNN → 50% accuracy
TNTNTNTNTNTNTNTNTNTN → 50% accuracy

-- Accuracy depends on the representativeness of profile input set

Static Branch Prediction (III)

- Program-based (or, program analysis based)
  - Idea: Use heuristics based on program analysis to determine statically-predicted direction
  - Opcode heuristic: Predict BLEZ as NT (negative integers used as error values in many programs)
  - Loop heuristic: Predict a branch guarding a loop execution as taken (i.e., execute the loop)
  - Pointer and FP comparisons: Predict not equal

+ Does not require profiling

-- Heuristics might be not representative or good

-- Requires compiler analysis and ISA support

- Ball and Larus, "Branch prediction for free," PLDI 1993.
  - 20% misprediction rate

Static Branch Prediction (III)

- Programmer-based
  - Idea: Programmer provides the statically-predicted direction
  - Via pragmas in the programming language that qualify a branch as likely-taken versus likely-not-taken

+ Does not require profiling or program analysis

+ Programmer may know some branches and their program better than other analysis techniques

-- Requires programming language, compiler, ISA support

-- Burdens the programmer?

Aside: Pragmas

- Idea: Keywords that enable a programmer to convey hints to lower levels of the transformation hierarchy

- if (likely(x)) { ... }
- if (unlikely(error)) { … }

- Many other hints and optimizations can be enabled with pragmas
  - E.g., whether a loop can be parallelized
  - **#pragma omp parallel**
  - **Description**
    - The omp parallel directive explicitly instructs the compiler to parallelize the chosen segment of code.

Static Branch Prediction

- All previous techniques can be combined
  - Profile based
  - Program based
  - Programmer based

- How would you do that?

- What are common disadvantages of all three techniques?
  - Cannot adapt to dynamic changes in branch behavior
    - This can be mitigated by a dynamic compiler, but not at a fine granularity (and a dynamic compiler has its overheads…)

Dynamic Branch Prediction

- Idea: Predict branches based on dynamic information (collected at run-time)

- Advantages

  \+ Prediction based on history of the execution of branches

    \+ It can adapt to dynamic changes in branch behavior

  \+ No need for static profiling: input set representativeness problem goes away

- Disadvantages

  -- More complex (requires additional hardware)

Last Time Predictor

- Last time predictor
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed

    TTTTTTTTTTNNNNNNNNNN → 90% accuracy

- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = (N-2)/N

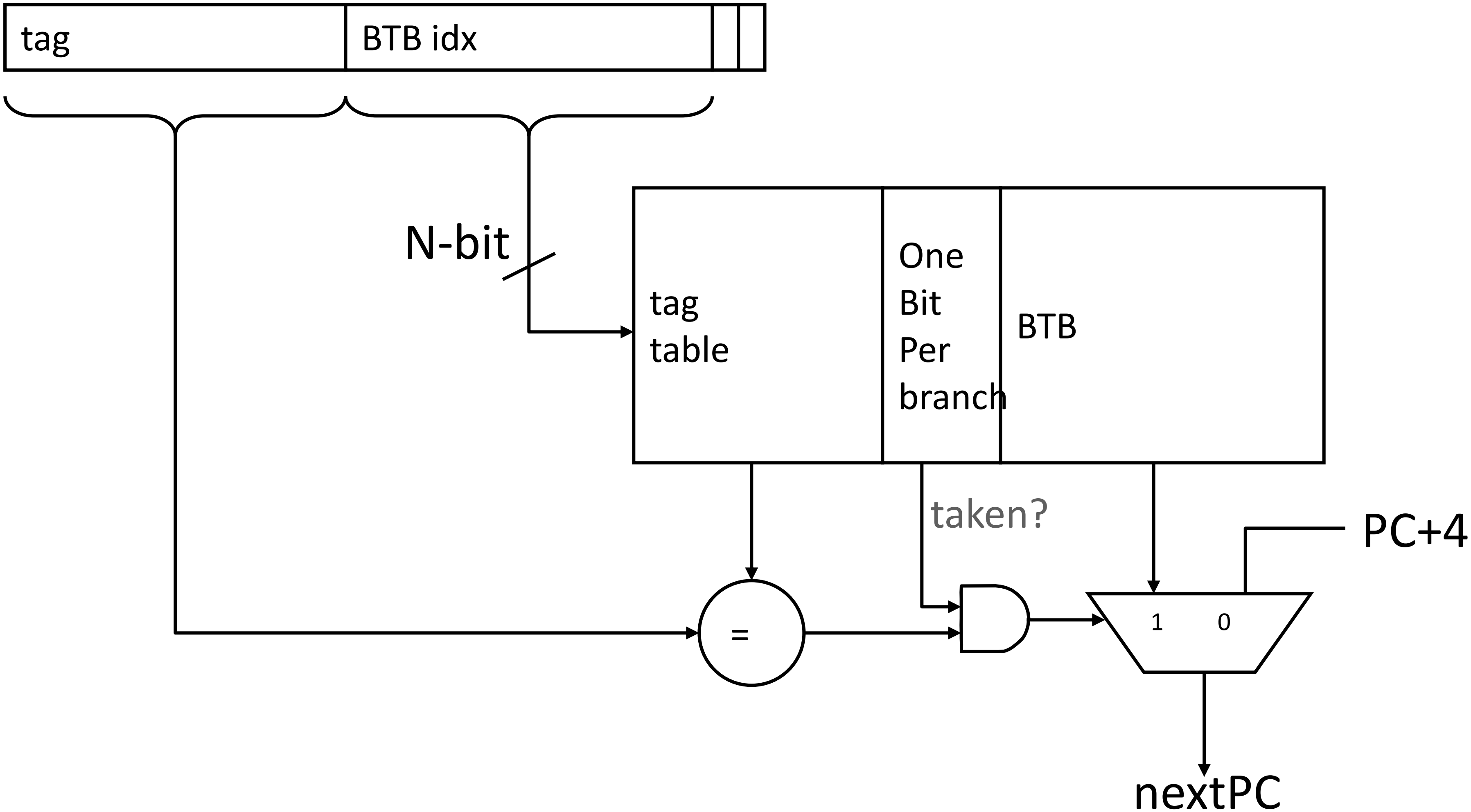
+ Loop branches for loops with large number of iterations

-- Loop branches for loops will small number of iterations

TNTNTNTNTNTNTNTNTNTN → 0% accuracy

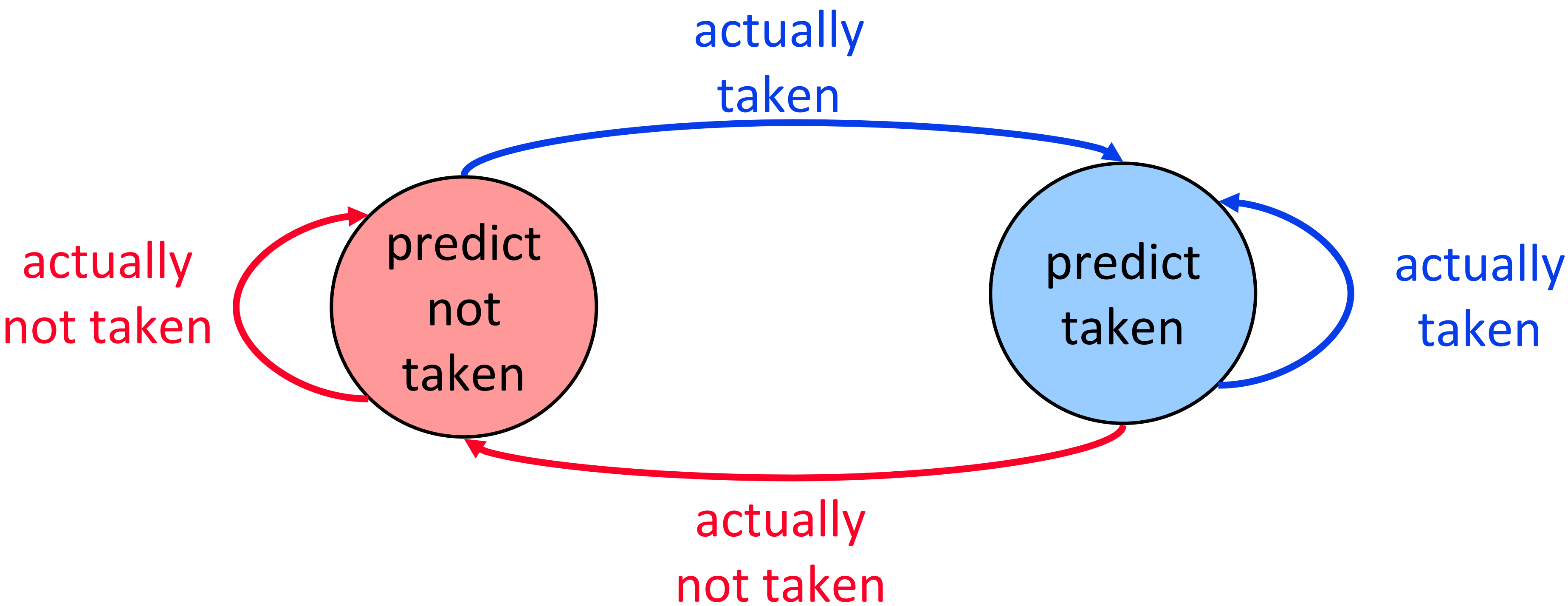
Last-time predictor CPI = [1 + (0.20*0.15) * 2] = 1.06 (Assuming 85% accuracy)

Implementing the Last-Time Predictor



The 1-bit BHT (Branch History Table) entry is updated with the correct outcome after each execution of a branch

State Machine for Last-Time Prediction



Improving the Last Time Predictor

- Problem: A last-time predictor changes its prediction from T→NT or NT→T too quickly
  - even though the branch may be mostly taken or mostly not taken
- Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
  - Use two bits to track the history of predictions for a branch instead of a single bit
  - Can have 2 states for T or NT instead of 1 state for each
- Smith, "A Study of Branch Prediction Strategies," ISCA 1981.

Two-Bit Counter Based Prediction

- Each branch associated with a two-bit counter
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome

- Accuracy for a loop with N iterations = (N-1)/N

  TNTNTNTNTNTNTNTNTNTN → 50% accuracy

  (assuming init to weakly taken)

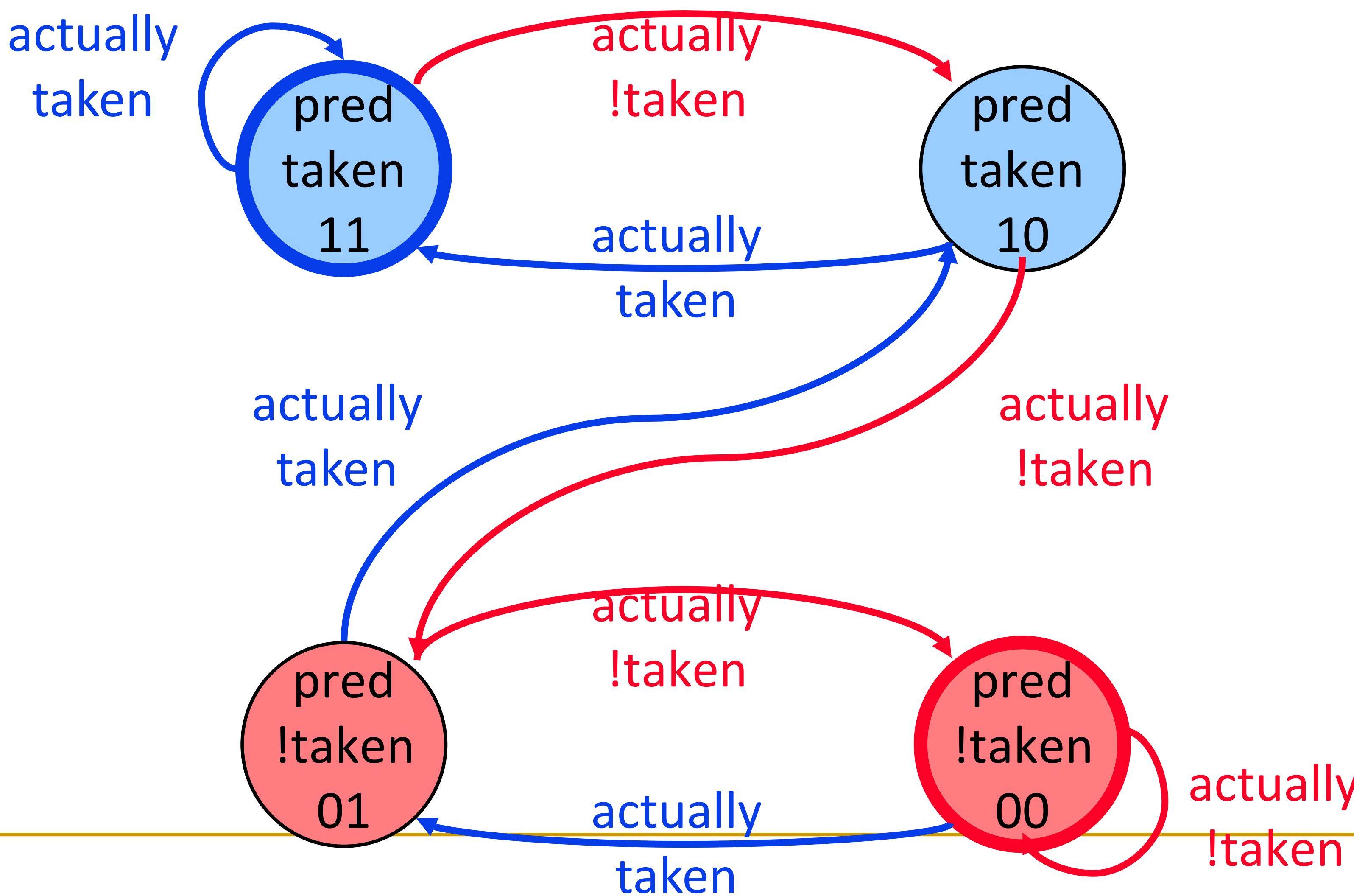
+ Better prediction accuracy

  2BC predictor CPI = [1 + (0.20*<u>0.10</u>) * 2] = 1.04 (90% accuracy)

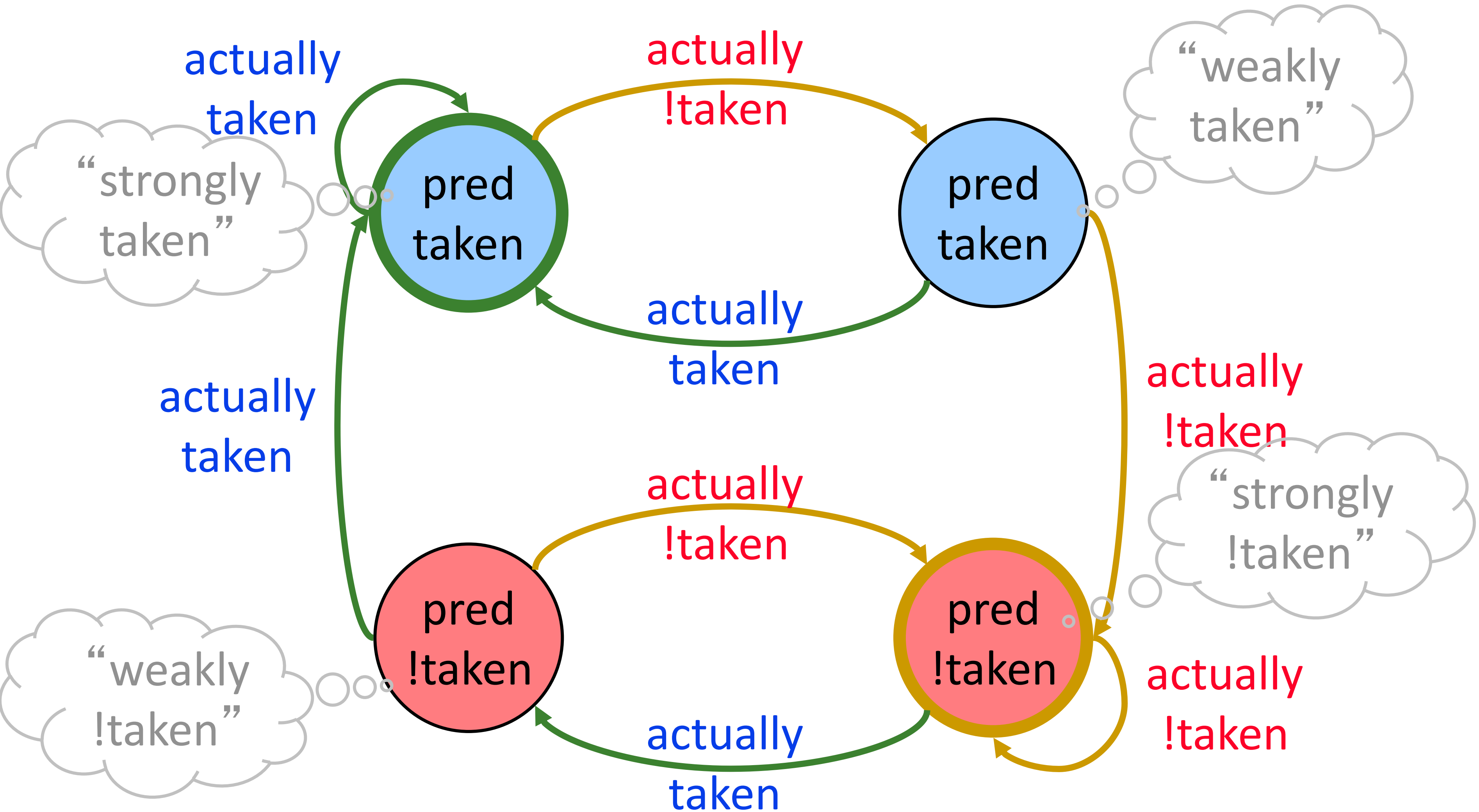
-- More hardware cost (but counter can be part of a BTB entry)

State Machine for 2-bit Saturating Counter

- Counter using saturating arithmetic
  - There is a symbol for maximum and minimum values



Hysteresis Using a 2-bit Counter



Change prediction after 2 consecutive mistakes

Is This Enough?

- ~85-90% accuracy for many programs with 2-bit counter based prediction (also called bimodal prediction)
- Is this good enough?
- How big is the branch problem?

Rethinking the The Branch Problem

- Control flow instructions (branches) are frequent
  - 15-25% of all instructions

- Problem: Next fetch address after a control-flow instruction is not determined after N cycles in a pipelined processor
  - N cycles: (minimum) branch resolution latency
  - Stalling on a branch wastes instruction processing bandwidth (i.e. reduces IPC)
    - N x IW instruction slots are wasted (IW: issue width)

- How do we keep the pipeline full after a branch?
- Problem: Need to determine the **next fetch address** when the branch is fetched (to avoid a pipeline bubble)

Importance of The Branch Problem

- Assume a 5-wide *superscalar* pipeline with 20-cycle branch resolution latency

- How long does it take to fetch 500 instructions?
  - Assume no fetch breaks and 1 out of 5 instructions is a branch
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work
  - 99% accuracy
    - 100 (correct path) + 20 (wrong path) = 120 cycles
    - 20% extra instructions fetched
  - 98% accuracy
    - 100 (correct path) + 20 * 2 (wrong path) = 140 cycles
    - 40% extra instructions fetched
  - 95% accuracy
    - 100 (correct path) + 20 * 5 (wrong path) = 200 cycles
    - 100% extra instructions fetched

Can We Do Better?

- Last-time and 2BC predictors exploit "last-time" predictability

- Realization 1: A branch's outcome can be correlated with other branches' outcomes
  - Global branch correlation

- Realization 2: A branch's outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch "last-time" it was executed)
  - Local branch correlation

Global Branch Correlation (I)

- Recently executed branch outcomes in the execution path is correlated with the outcome of the next branch

```
if (cond1)
...
if (cond1 AND cond2)
```

- If first branch not taken, second also not taken

```
branch Y: if (cond1) a = 2;
...
branch X: if (a == 0)
```

- If first branch taken, second definitely not taken

Global Branch Correlation (II)

```
branch Y: if (cond1)
...
branch Z: if (cond2)
...
branch X: if (cond1 AND cond2)
```

- If Y and Z both taken, then X also taken
- If Y or Z not taken, then X also not taken

Global Branch Correlation (III)

- Eqntott, SPEC 1992

```
if (aa==2)          ;; B1
      aa=0;
if (bb==2)          ;; B2
      bb=0;
if (aa!=bb) {       ;; B3
      ....
}
```

If B1 is not taken (i.e. aa==0@B3) and B2 is not taken (i.e. bb=0@B3) then B3 is certainly taken

Capturing Global Branch Correlation

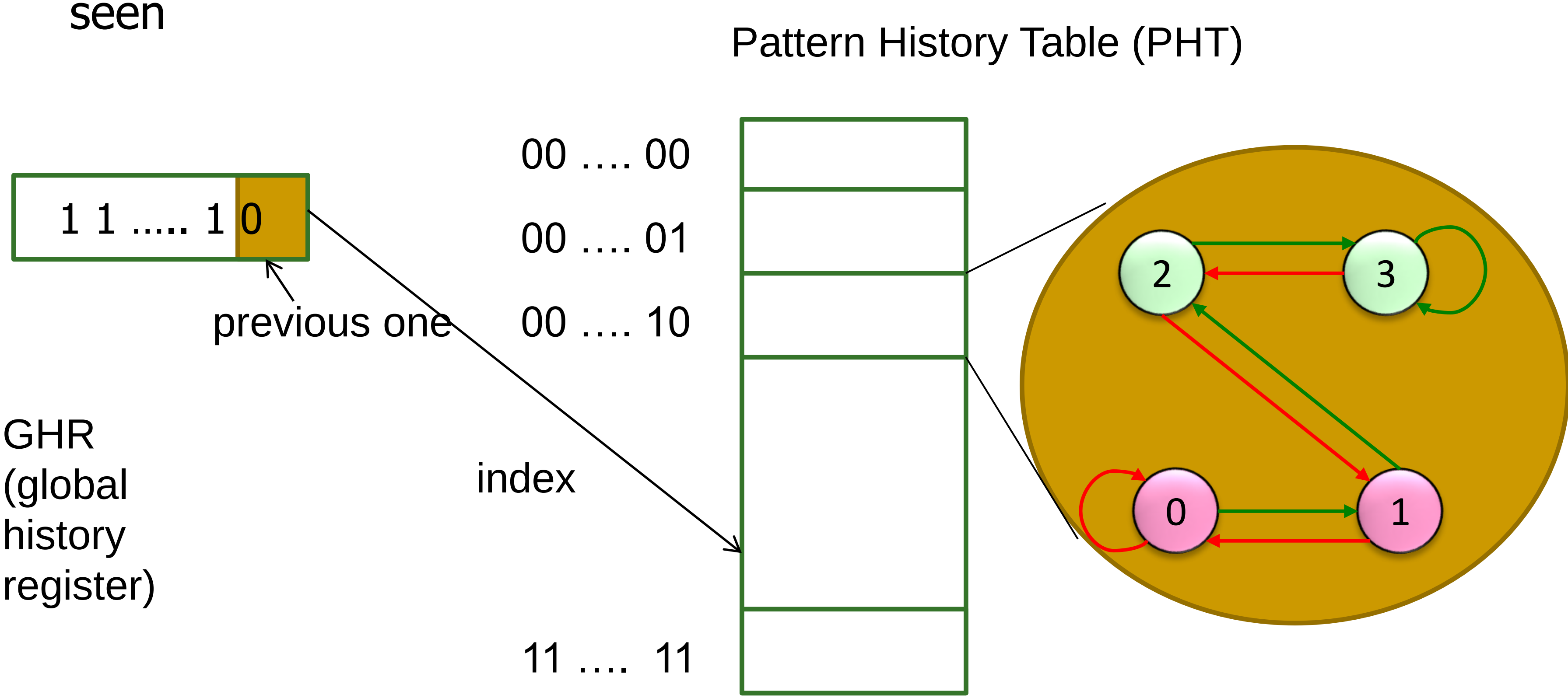
- Idea: Associate branch outcomes with "global T/NT history" of all branches
- Make a prediction based on the outcome of the branch the last time the same global branch history was encountered

- Implementation:
  - Keep track of the "global T/NT history" of all branches in a register → Global History Register (GHR)
  - Use GHR to index into a table of that recorded the outcome that was seen for that GHR value in the recent past → Pattern History Table (table of 2-bit counters)

- Global history/branch predictor
- Uses two levels of history (GHR + history at that GHR)

Two Level Global Branch Prediction

- First level: Global branch history register (N bits)
  - The direction of last N branches
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen



Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

How Does the Global Predictor Work?

```
for (i=0; i<100; i++)
    for (j=0; j<3; j++)
```

After the initial startup time, the conditional branches have the following behavior, assuming GR is shifted to the left:

test	value	GR	result
j<3	j=1	1101	taken
j<3	j=2	1011	taken
j<3	j=3	0111	not taken
i<100		1110	usually taken

- McFarling, "Combining Branch Predictors," DEC WRL TR 1993.

Intel Pentium Pro Branch Predictor

- 4-bit global history register
- Multiple pattern history tables (of 2 bit counters)
  - Which pattern history table to use is determined by lower order bits of the branch address

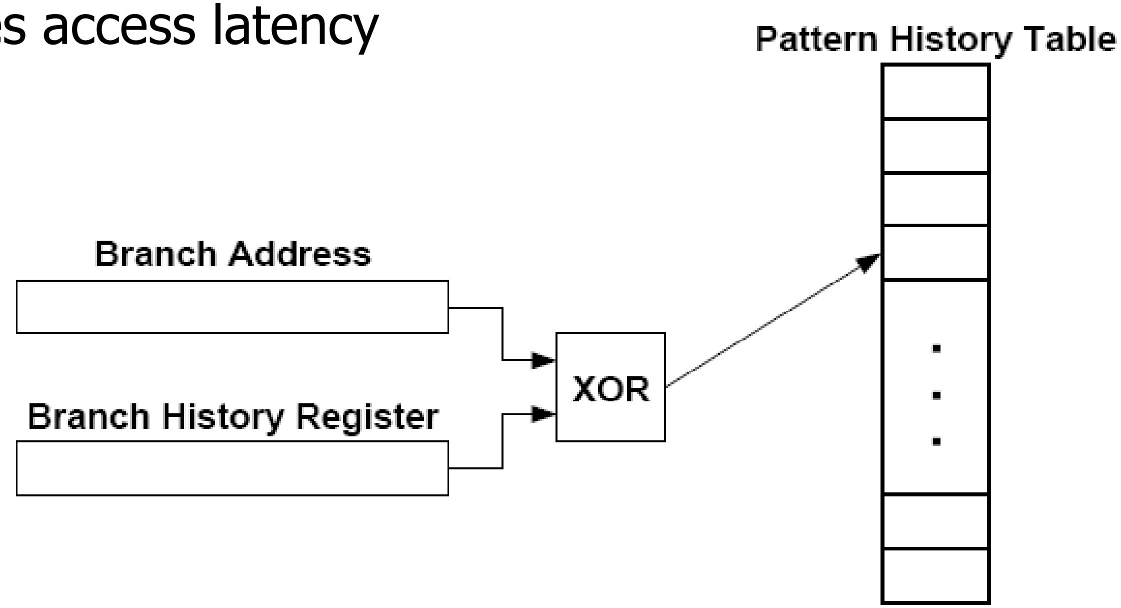
Improving Global Predictor Accuracy

- Idea: Add more context information to the global predictor to take into account which branch is being predicted
  - Gshare predictor: GHR hashed with the Branch PC

  + More context information

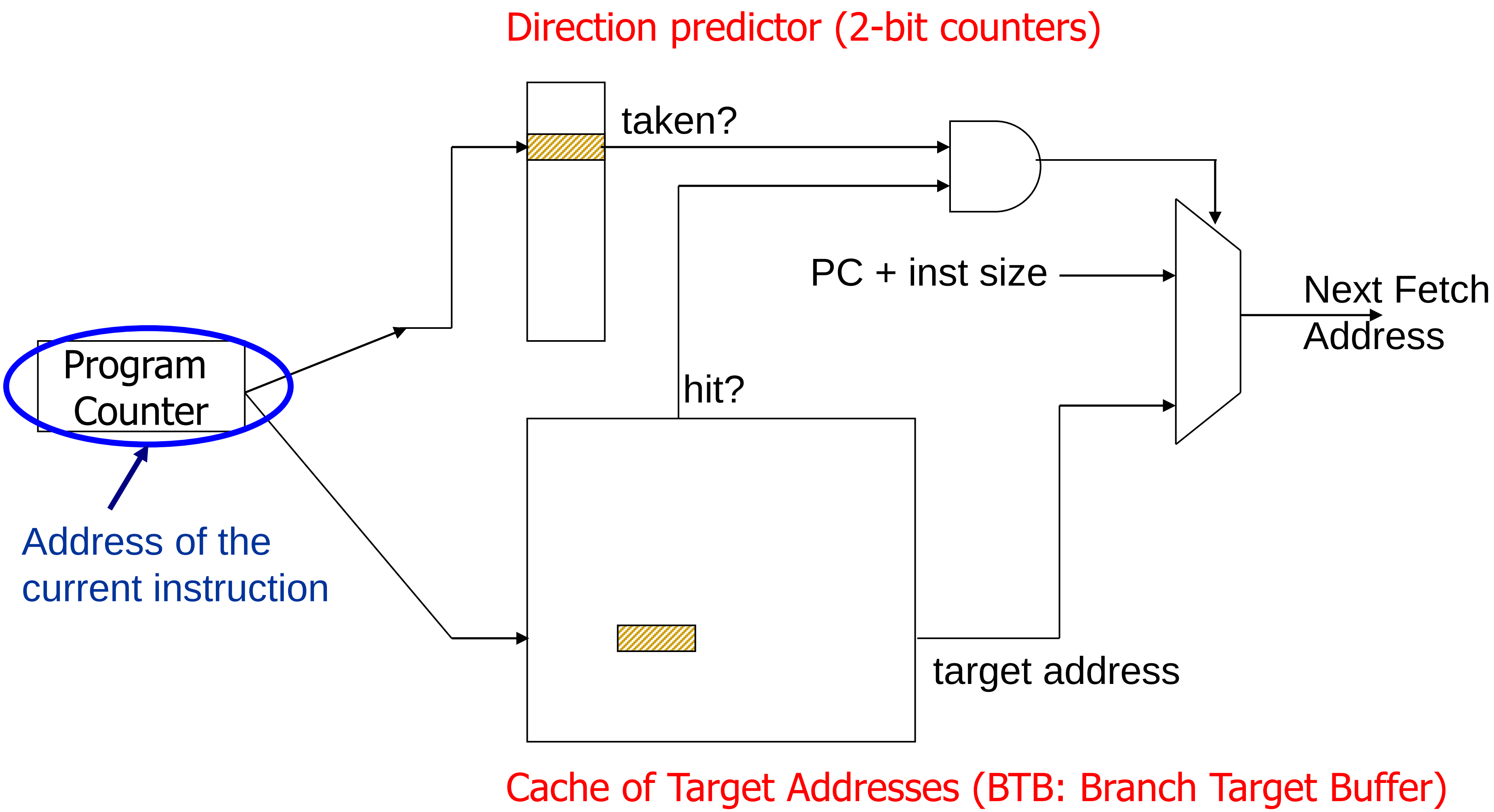
  + Better utilization of PHT

  -- Increases access latency

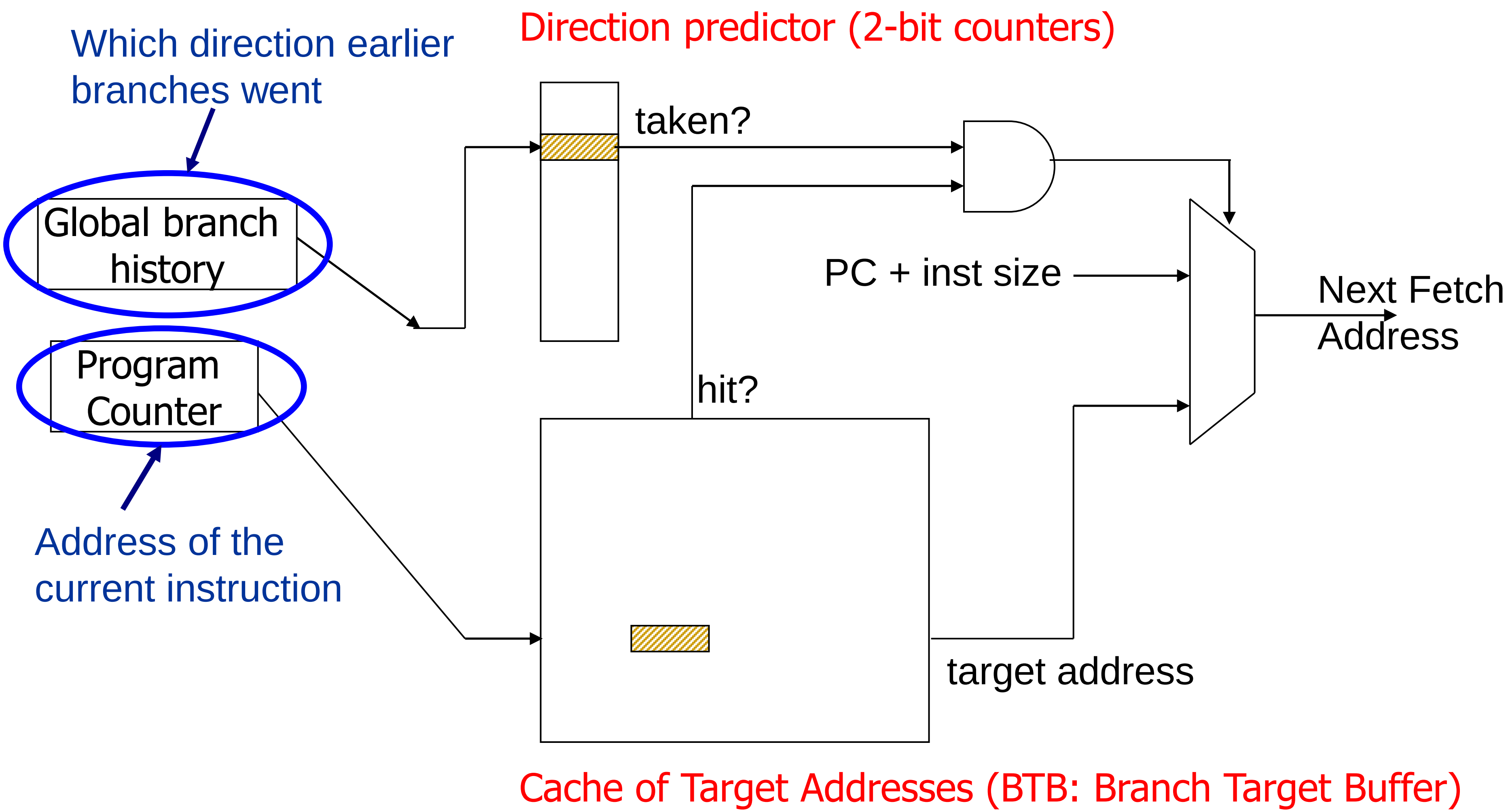


- McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

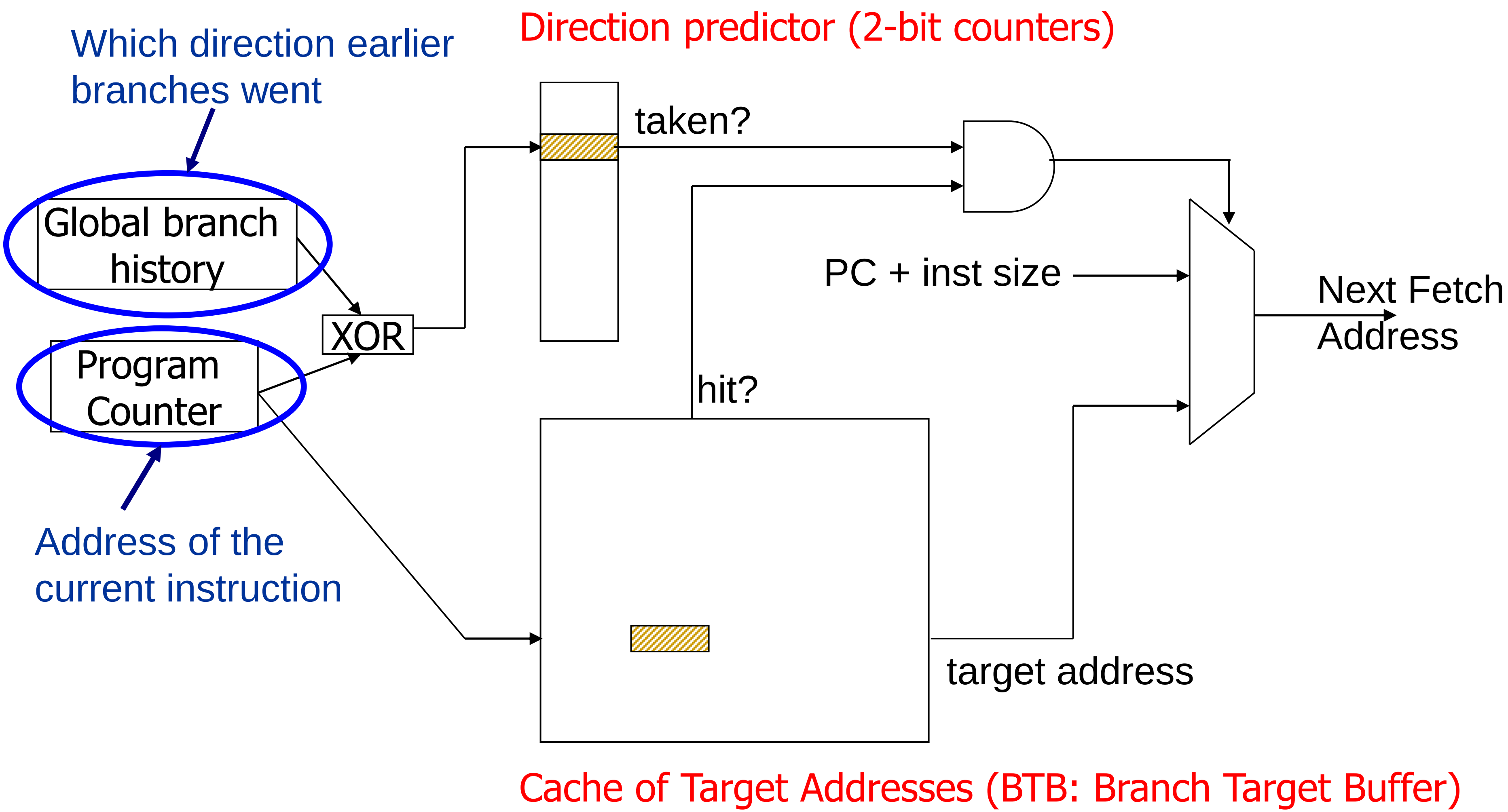
One-Level Branch Predictor



Two-Level Global History Predictor



Two-Level Gshare Predictor



Can We Do Better?

- Last-time and 2BC predictors exploit “last-time” predictability

- Realization 1: A branch’s outcome can be correlated with other branches’ outcomes
  - Global branch correlation

- Realization 2: A branch’s outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch “last-time” it was executed)
  - Local branch correlation

Local Branch Correlation

for (i=1; i<=4; i++) { }

If the loop test is done at the end of the body, the corresponding branch will execute the pattern $(1110)^n$, where 1 and 0 represent taken and not taken respectively, and n is the number of times the loop is executed. Clearly, if we knew the direction this branch had gone on the previous three executions, then we could always be able to predict the next branch direction.

- McFarling, "Combining Branch Predictors," DEC WRL TR 1993.

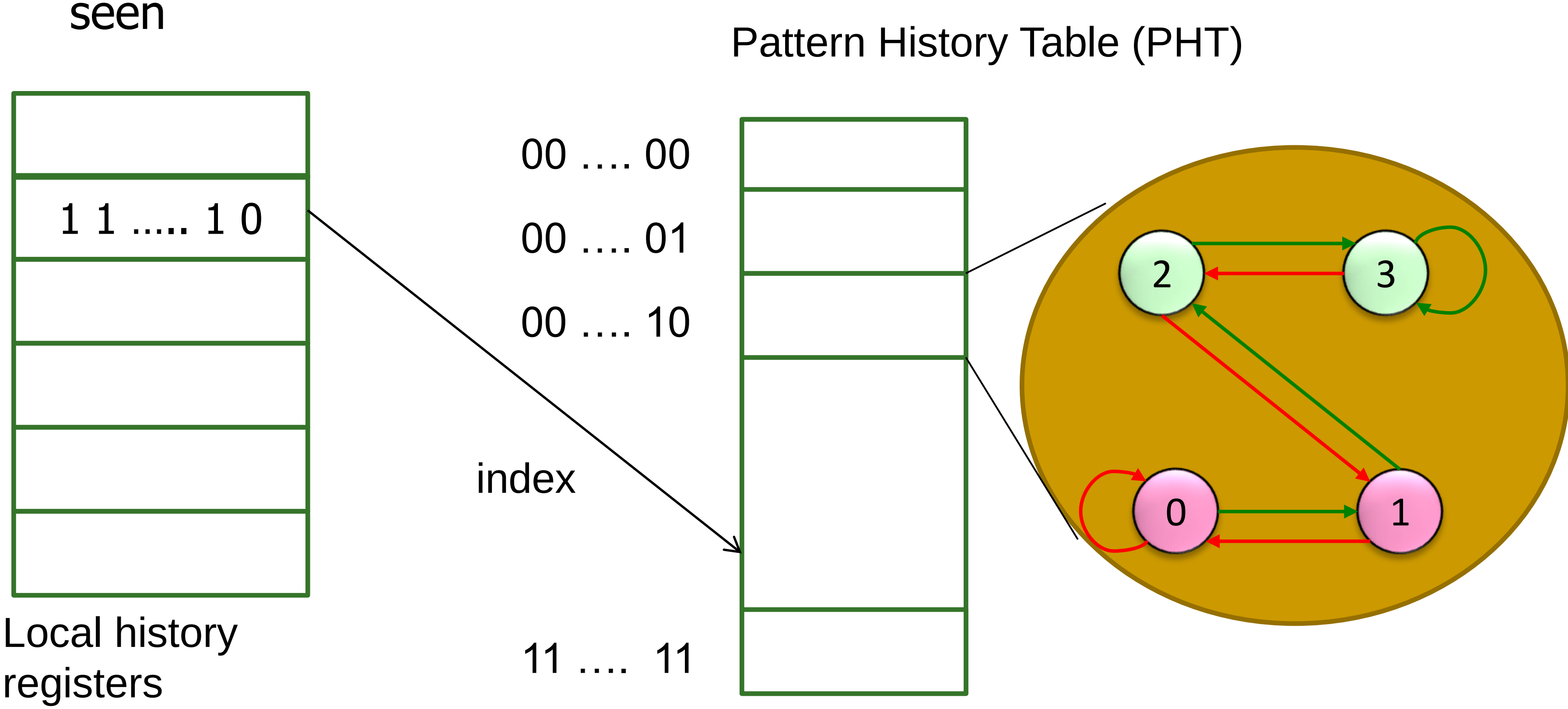
Capturing Local Branch Correlation

- Idea: Have a per-branch history register
  - Associate the predicted outcome of a branch with "T/NT history" of the same branch
- Make a prediction is based on the outcome of the branch the last time the same local branch history was encountered

- Called the local history/branch predictor
- Uses two levels of history (Per-branch history register + history at that history register value)

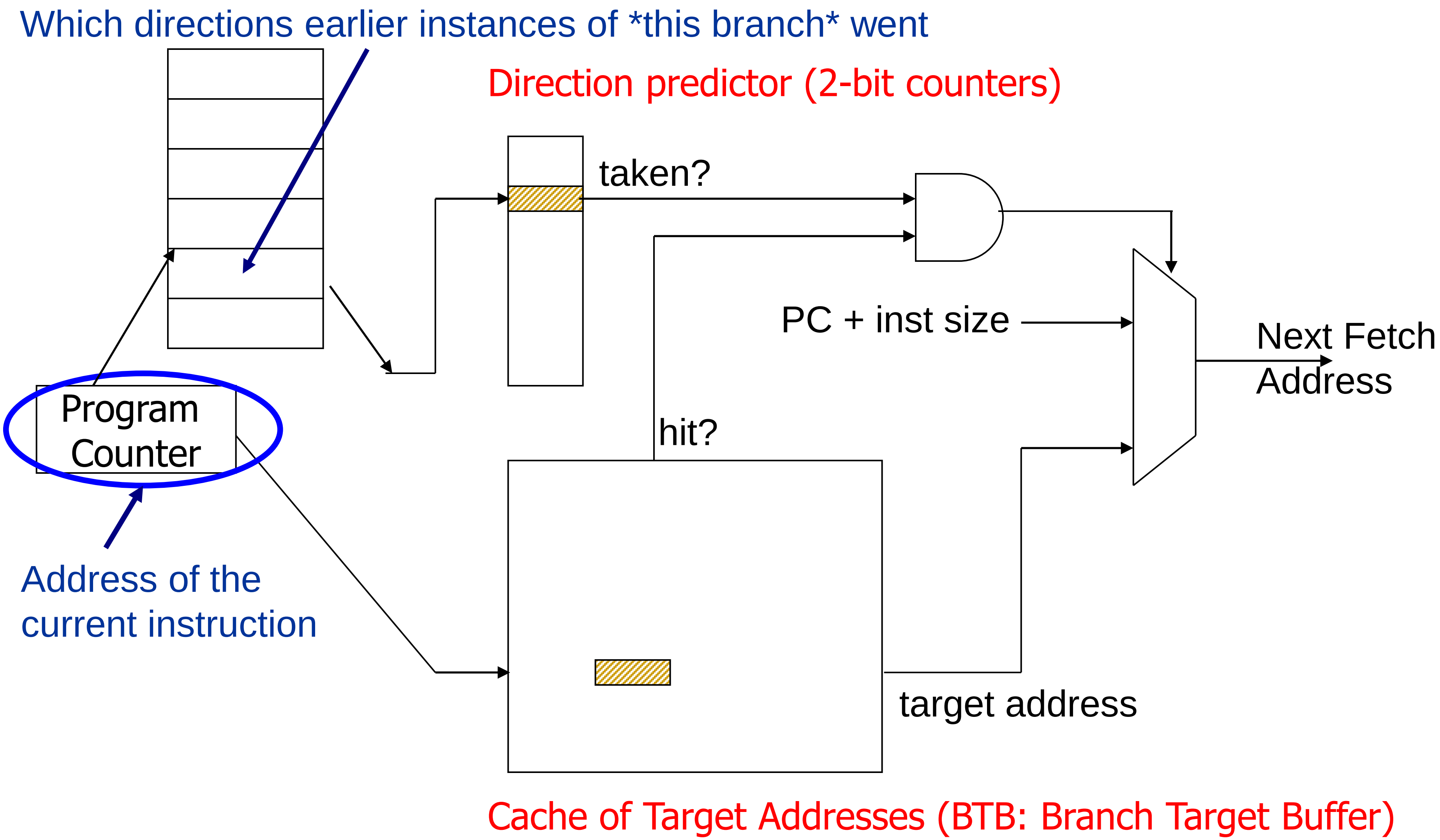
Two Level Local Branch Prediction

- First level: A set of local history registers (N bits each)
  - Select the history register based on the PC of the branch
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen



Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

Two-Level Local History Predictor



Hybrid Branch Predictors

- Idea: Use more than one type of predictor (i.e., multiple algorithms) and select the "best" prediction
  - E.g., hybrid of 2-bit counters and global predictor

- Advantages:

  + Better accuracy: different predictors are better for different branches

  + Reduced warmup time (faster-warmup predictor used until the slower-warmup predictor warms up)

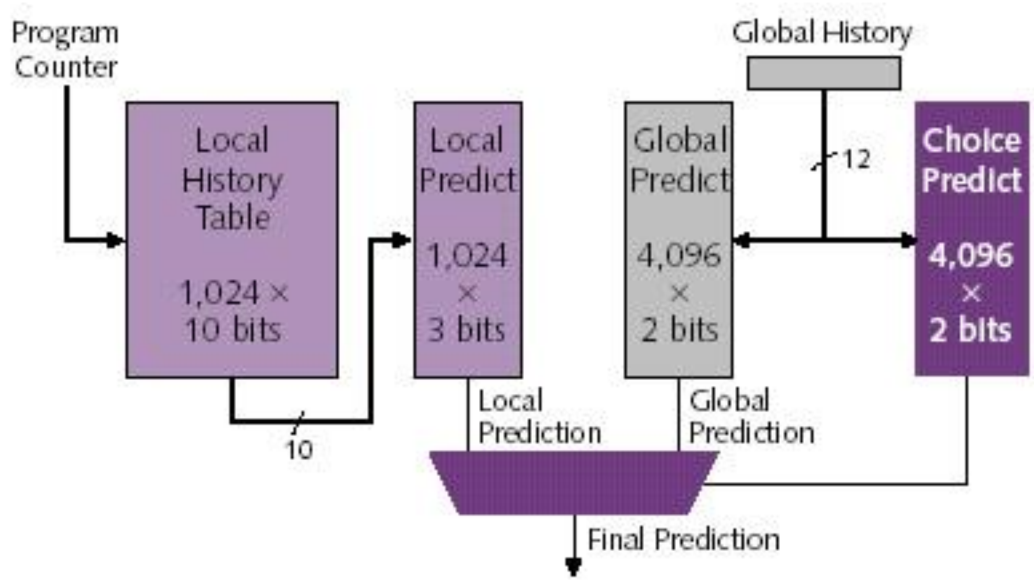
- Disadvantages:

  -- Need "meta-predictor" or "selector"

  -- Longer access latency

  - McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

Alpha 21264 Tournament Predictor



- Minimum branch penalty: 7 cycles
- Typical branch penalty: 11+ cycles
- 48K bits of target addresses stored in I-cache
- Predictor tables are reset on a context switch

- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

Branch Prediction Accuracy (Example)

- Bimodal: table of 2bc indexed by branch address

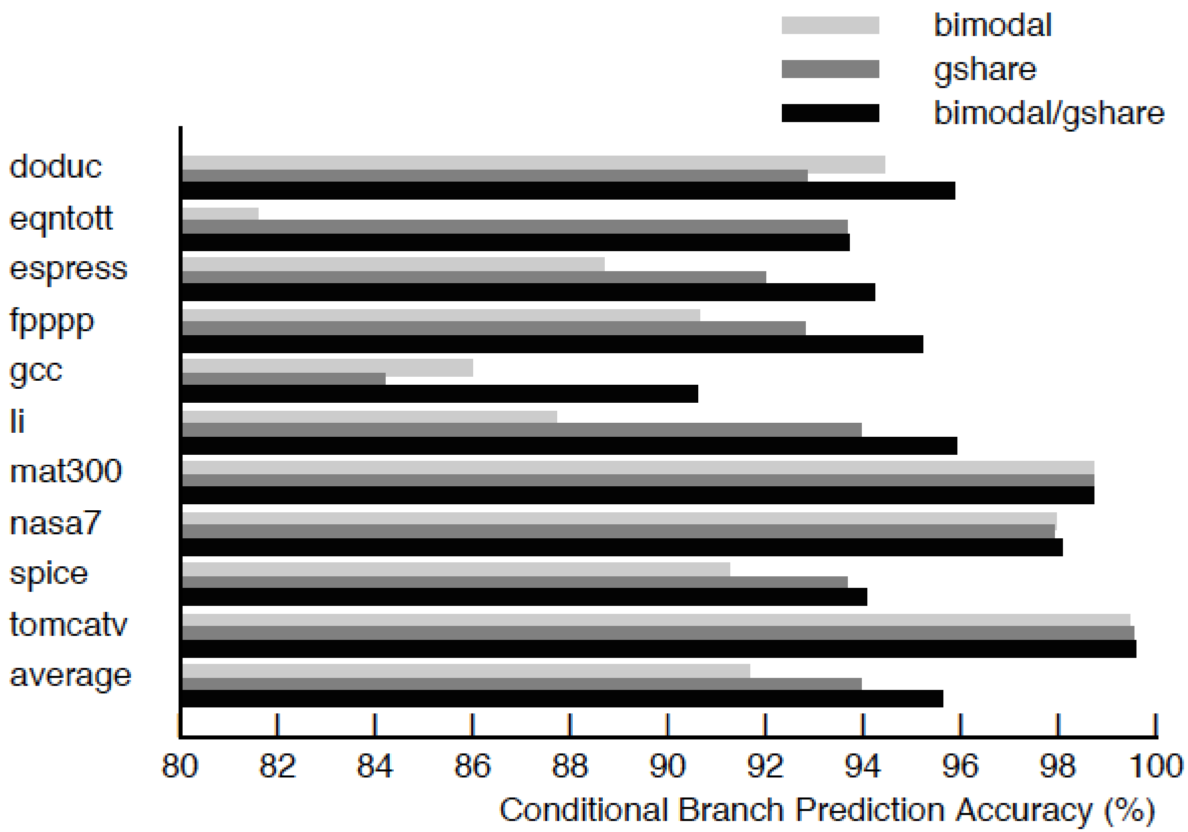


Figure 13: Combined Predictor Performance by Benchmark

Biased Branches

- Observation: Many branches are biased in one direction (e.g., 99% taken)

- Problem: These branches *pollute* the branch prediction structures → make the prediction of other branches difficult by causing "interference" in branch prediction tables and history registers

- Solution: Detect such biased branches, and predict them with a simpler predictor

- Chang et al., "Branch classification: a new mechanism for improving branch predictor performance," MICRO 1994.