

# **Initial Design of an Alpha Processor**

**March 12, 1998**

**(Revised 3/16/98)**

## **Topics**

- **Objective**
- **Instruction formats**
- **Instruction processing**
- **Principles of pipelining**
- **Inserting pipe registers**

# Objective

## Design Processor for Alpha Subset

- Interesting but not overwhelming quantity
- High level functional blocks

## Initial Design

- One instruction at a time
- Single cycle per instruction
  - Follows H&P 5.1--5.3

## Refined Design

- 5-stage pipeline
  - Similar to early RISC processors
  - Follows H&P 6.1--6.7
- **Goal: approach 1 cycle per instruction but with shorter cycle time**

# Alpha Arithmetic Instructions

**RR-type instructions (addq, subq, xor, bis, cmplt):**  $rc \leftarrow ra \text{ funct } rb$

Op	ra	rb	000	0	funct	rc
31-26	25-21	20-16	15-13	12	11-5	4-0

**RI-type instructions (addq, subq, xor, bis, cmplt):**  $rc \leftarrow ra \text{ funct } ib$

Op	ra	ib	1	funct	rc
31-26	25-21	20-13	12	11-5	4-0

## Encoding

- **ib is 8-bit unsigned literal**

Operation	Op field	funct field
addq	0x10	0x20
subq	0x10	0x29
bis	0x11	0x20
xor	0x11	0x40
cmoveq	0x11	0x24
cmplt	0x11	0x4D

# Alpha Load/Store Instructions

**Load:**  $Ra \leftarrow \text{Mem}[Rb + \text{offset}]$

**Store:**  $\text{Mem}[Rb + \text{offset}] \leftarrow Ra$

Op	ra	rb	offset
31-26	25-21	20-16	15-0

## Encoding

- **offset is 16-bit signed offset**

Operation	Op field
ldq	0x29
stq	0x2D

# Branch Instructions

**Cond. Branch:**  $PC \leftarrow \text{Cond}(Ra) ? PC + 4 + \text{disp} * 4 : PC + 4$

Op	ra	disp
31-26	25-21	20-0

## Encoding

- **disp is 21-bit signed displacement**

Operation	Op field	Cond
beq	0x39	Ra == 0
bne	0x3D	Ra != 0

**Branch [Subroutine] (br, bsr):**  $Ra \leftarrow PC + 4; PC \leftarrow PC + 4 + \text{disp} * 4$

Op	ra	disp
31-26	25-21	20-0

Operation	Op field
br	0x30
bsr	0x34

# Transfers of Control

**jmp, jsr, ret:** Ra  $\leftarrow$  PC+4; PC  $\leftarrow$  Rb

0x1A	ra	rb	Hint
31-26	25-21	20-16	15-0

## Encoding

- High order 2 bits of Hint encode jump type
- Remaining bits give information about predicted destination
- Hint does not affect functionality

## Jump Type

## Hint 15:14

jmp	00
jsr	01
ret	10

## call\_pal

0x00	Number
31-26	25-0

- Use as halt instruction

# Instruction Encoding

0x0:	40220403	addq	r1, r2, r3
0x4:	4487f805	xor	r4, 0x3f, r5
0x8:	a4c70abc	ldq	r6, 2748(r7)
0xc:	b5090123	stq	r8, 291(r9)
0x10:	e47ffffb	beq	r3, 0
0x14:	d35ffffa	bsr	r26, 0(r31)
0x18:	6bfa8001	ret	r31, (r26), 1
0x1c:	000abcde	call_pal	0xabcde

## Object Code

- Instructions encoded in 32-bit words
- Program behavior determined by bit encodings
- Disassembler simply converts these words to readable instructions

# Decoding Examples

0x0: 40220403 addq r1, r2, r3

4	0	2	2	0	4	0	3
0100	0000	0010	0010	0000	0100	0000	0011
10		01		02		20	
						03	

0x8: a4c70abc ldq r6, 2748(r7)

a	4	c	7	0	a	b	c
1010	0100	1100	0111	0000	1010	1011	1100
29		06		07		0abc	
						= 2748 <sub>10</sub>	

0x10: e47ffffb beq r3, 0

e	4	7	f	f	f	f	b
1110	0100	0111	1111	1111	1111	1111	1011
39		03		1ffffb			
				= -5 <sub>10</sub>			

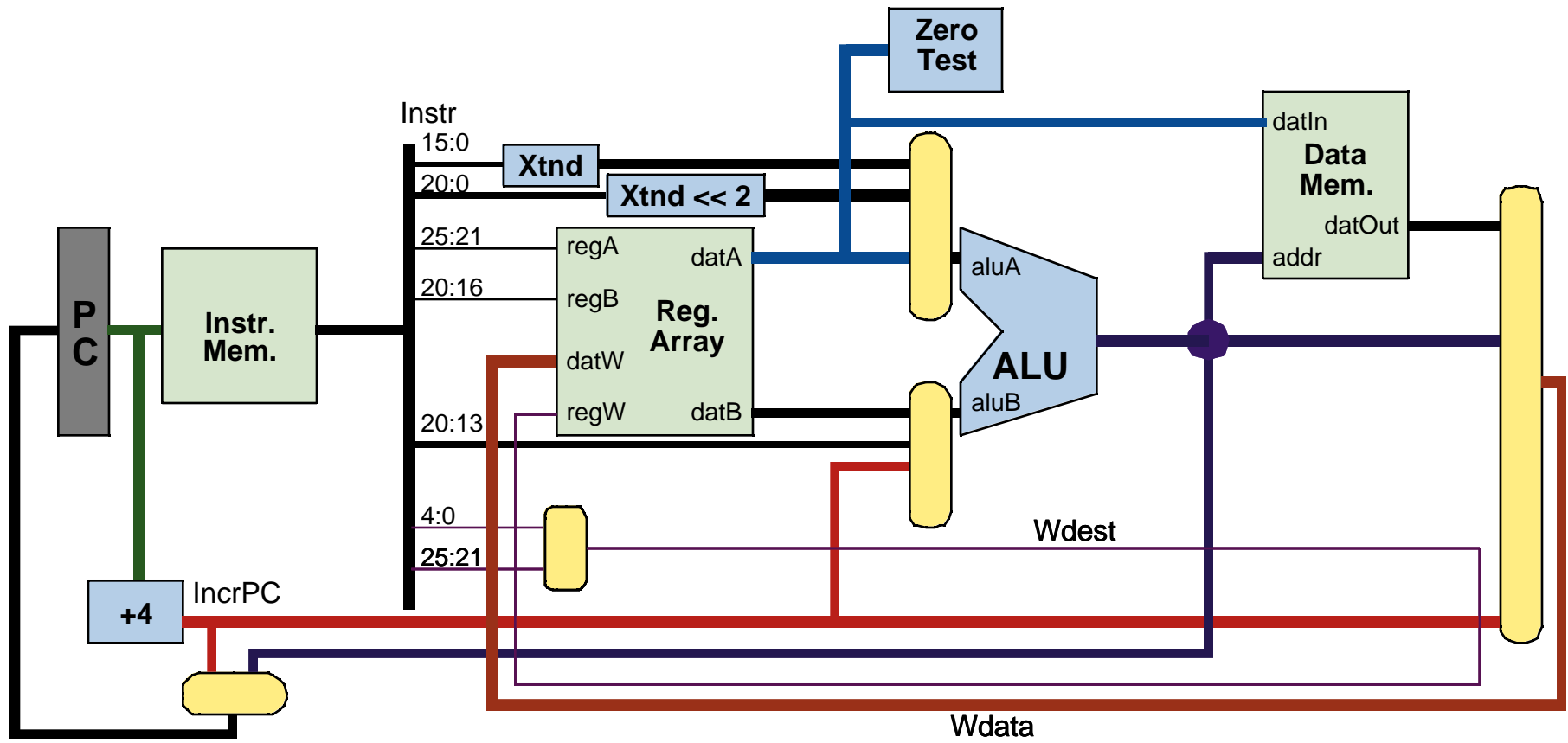
0x18: 6bfa8001 ret r31, (r26), 1

6	b	f	a	8	0	0	1
0110	1011	1111	1010	1000	0000	0000	0001
1a		1f		1a		2	
						= 31 <sub>10</sub> = 26 <sub>10</sub>	

Target =	16	# Current PC
+	4	# Increment
+	4 * -5	# Disp
=	0	



# Datapath



# Hardware Units

## Storage

- **Instruction Memory**
  - Fetch 32-bit instructions
- **Data Memory**
  - Load / store 64-bit data
- **Register Array**
  - Storage for 32 integer registers
  - Two read ports: can read two registers at once
  - Single write port

## Functional Units

- |                    |  |
|--------------------|--|
| • <b>+4</b>        | <b>PC incrementer</b>                      |
| • <b>Xtnd</b>      | <b>Sign extender</b>                       |
| • <b>ALU</b>       | <b>Arithmetic and logical instructions</b> |
| • <b>Zero Test</b> | <b>Detect whether operand == 0</b>         |

# RR-type instructions

RR-type instructions (addq, subq, xor, bis, cmplt):  $rc \leftarrow ra \text{ funct } rb$

Op	ra	rb	000	0	funct	rc
31-26	25-21	20-16	15-13	12	11-5	4-0

## IF: Instruction fetch

- $IR \leftarrow I\text{Memory}[PC]$
- $PC \leftarrow PC + 4$

## ID: Instruction decode/register fetch

- $A \leftarrow \text{Register}[IR[25:21]]$
- $B \leftarrow \text{Register}[IR[20:16]]$

## Ex: Execute

- $ALUOutput \leftarrow A \text{ op } B$

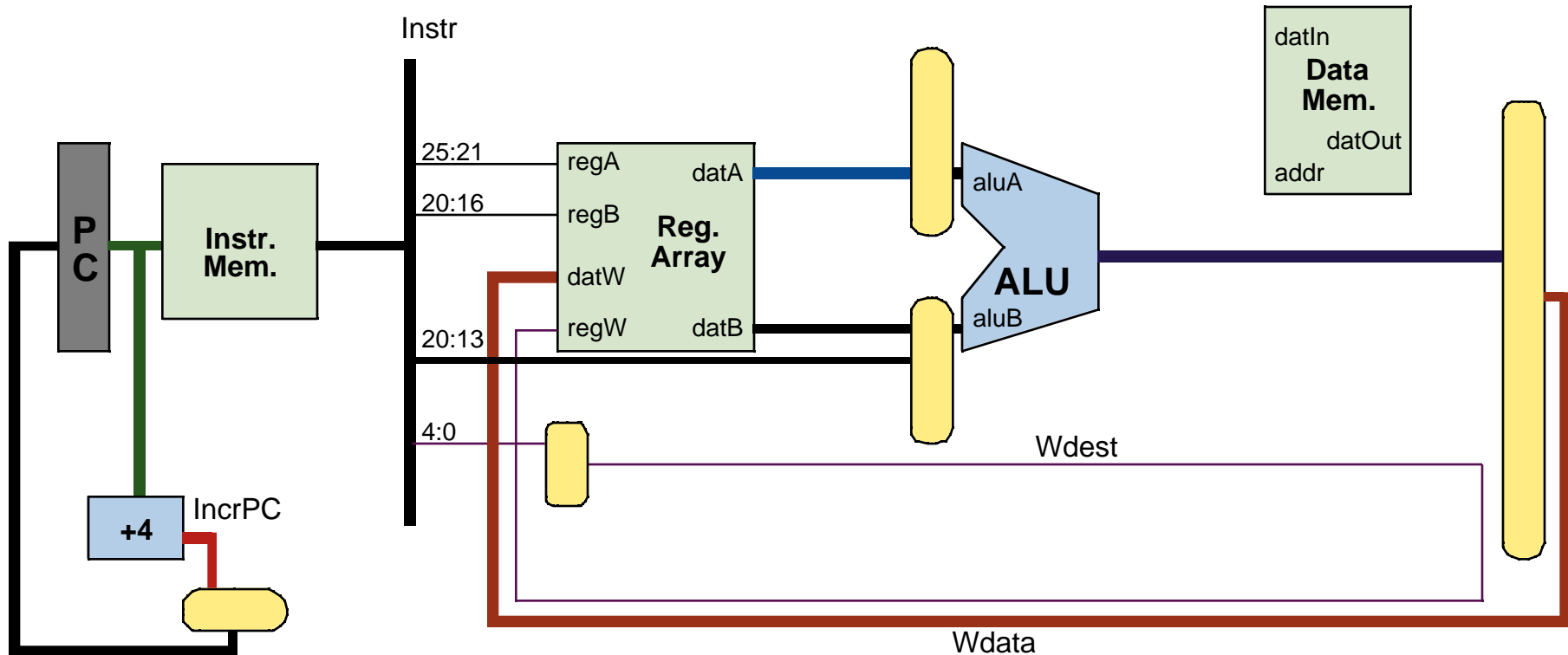
## MEM: Memory

- nop

## WB: Write back

- $\text{Register}[IR[4:0]] \leftarrow ALUOutput$

# Active Datapath for RR & RI



## ALU Operation

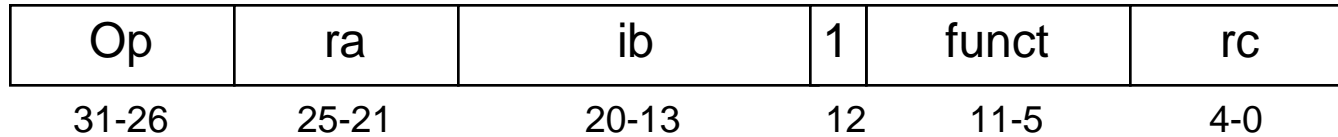
- Input B selected according to instruction type
  - datB for RR, IR[20:13] for RI
- ALU function set according to operation type
  - 12 –

## Write Back

- To Rc

# RI-type instructions

RI-type instructions (addq, subq, xor, bis, cmplt):  $rc \leftarrow ra \text{ funct } ib$



## IF: Instruction fetch

- $IR \leftarrow I\text{Memory}[PC]$
- $PC \leftarrow PC + 4$

## ID: Instruction decode/register fetch

- $A \leftarrow \text{Register}[IR[25:21]]$
- $B \leftarrow IR[20:13]$

## Ex: Execute

- $ALUOutput \leftarrow A \text{ op } B$

## MEM: Memory

- nop

## WB: Write back

- $\text{Register}[IR[4:0]] \leftarrow ALUOutput$

# Load instruction

Load:  $Ra \leftarrow Mem[Rb + offset]$

Op	ra	rb	offset
31-26	25-21	20-16	15-0

## IF: Instruction fetch

- $IR \leftarrow IMemory[PC]$
- $PC \leftarrow PC + 4$

## ID: Instruction decode/register fetch

- $B \leftarrow Register[IR[20:16]]$

## Ex: Execute

- $ALUOutput \leftarrow B + SignExtend(IR[15:0])$

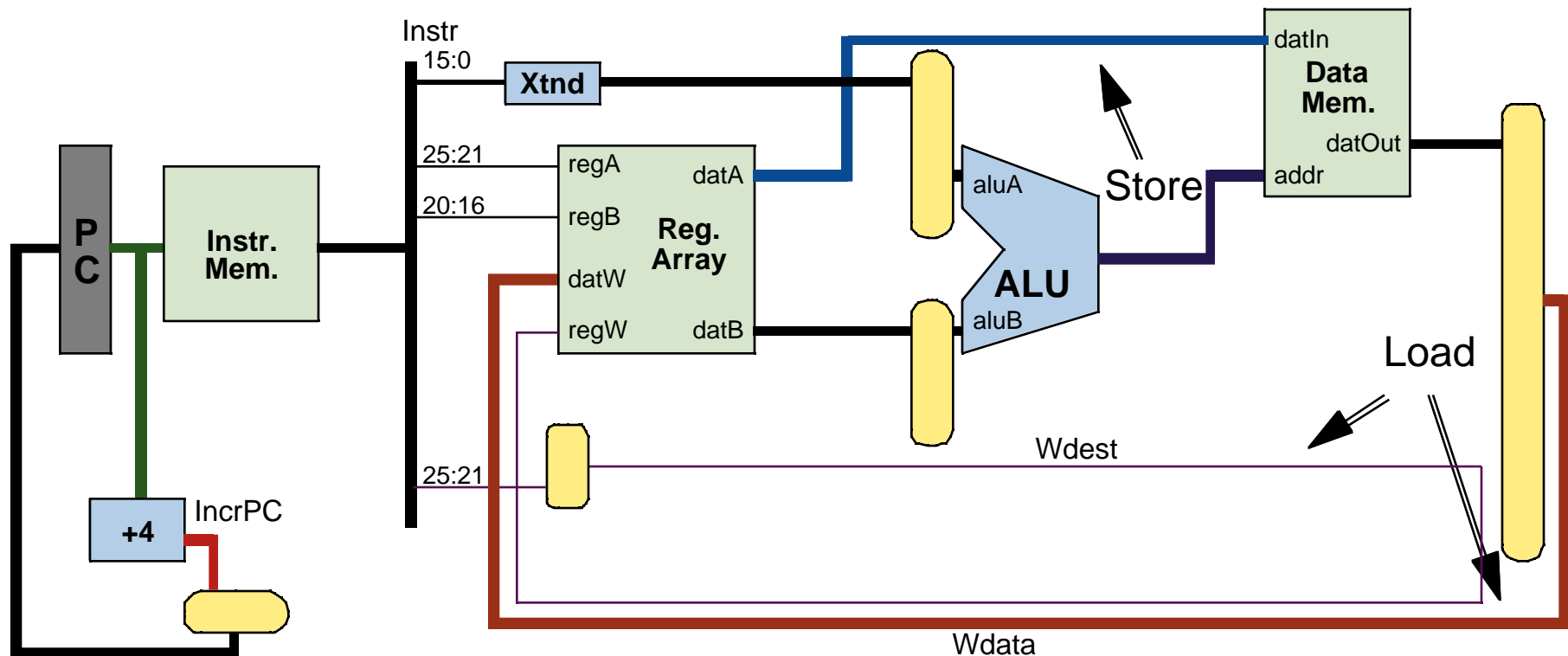
## MEM: Memory

- $Mem-Data \leftarrow DMemory[ALUOutput]$

## WB: Write back

- $Register[IR[25:21]] \leftarrow Mem-Data$

# Active Datapath for Load & Store



## ALU Operation

- Used to compute address
  - A input set to extended IR[15:0]
- ALU function set to add

## Memory Operation

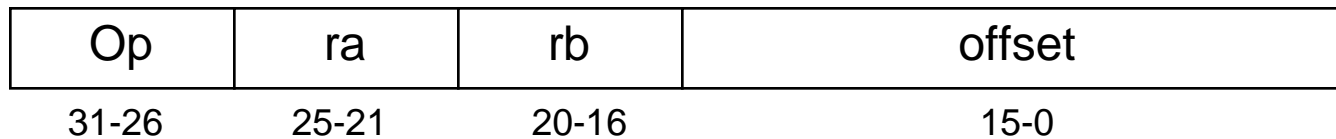
- Read for load, write for store

## Write Back

- To Ra for load
- None for store

# Store instruction

**Store:** Mem[Rb +offset] <-- Ra



## IF: Instruction fetch

- IR <-- IMemory[PC]
- PC <-- PC + 4

## ID: Instruction decode/register fetch

- A <-- Register[IR[25:21]]
- B <-- Register[IR[20:16]]

## Ex: Execute

- ALUOutput <-- B + SignExtend(IR[15:0])

## MEM: Memory

- DMemory[ALUOutput] <-- A

## WB: Write back

- nop



# Branch on equal

**beq:**  $PC \leftarrow Ra == 0 ? PC + 4 + disp * 4 : PC + 4$

0x39	ra	disp
31-26	25-21	20-0

## IF: Instruction fetch

- $IR \leftarrow IMemory[PC]$
- $incrPC \leftarrow PC + 4$

## ID: Instruction decode/register fetch

- $A \leftarrow Register[IR[25:21]]$

## Ex: Execute

- $Target \leftarrow incrPC + SignExtend(IR[20:0]) \ll 2$
- $Z \leftarrow (A == 0)$

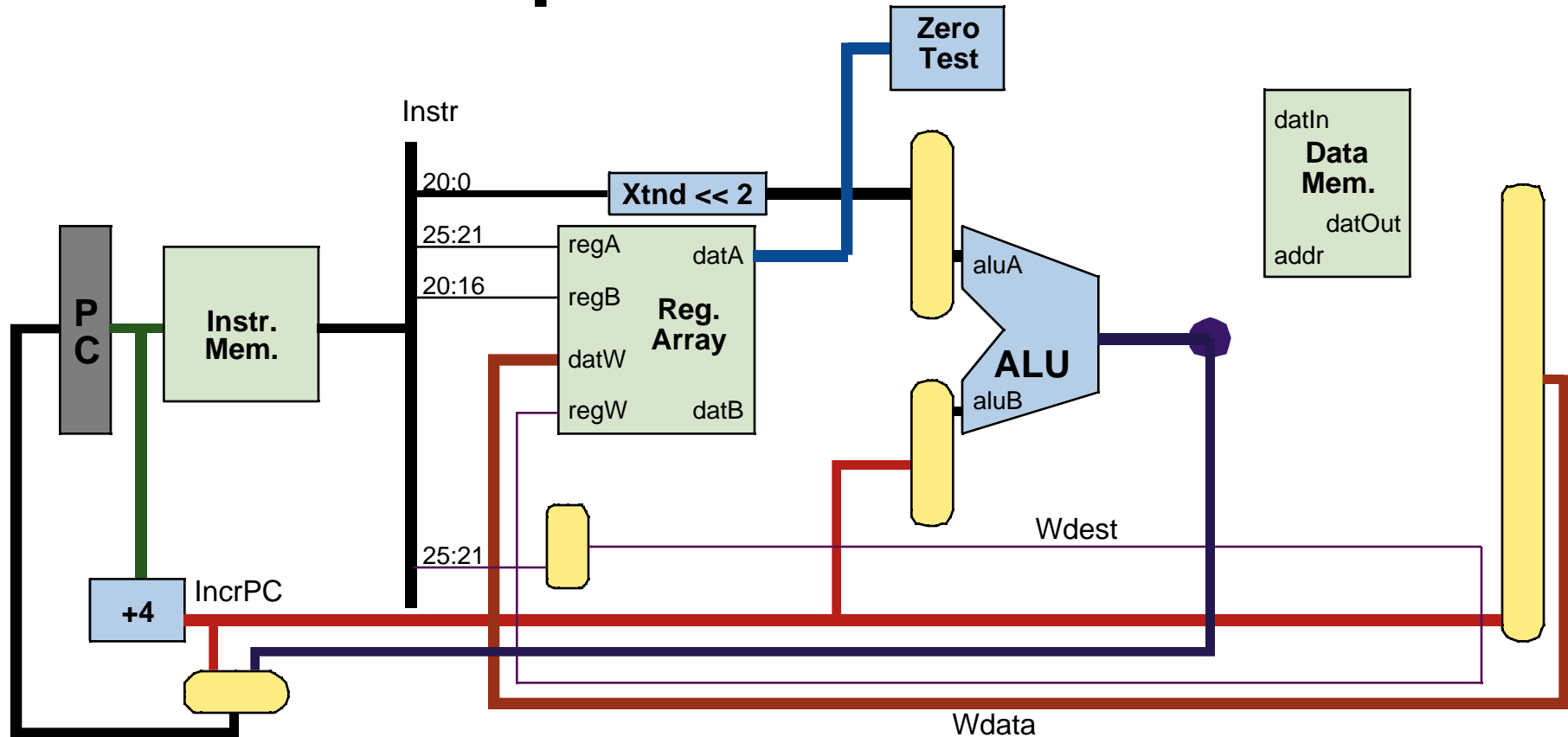
## MEM: Memory

- $PC \leftarrow Z ? Target : incrPC$

## WB: Write back

- nop

# Active Datapath for Branch and BSR



## ALU Computes target

- A = shifted, extended IR[20:0]
- B = IncrPC
- Function set to add

## Zero Test

- Determines branch condition

## PC Selection

- Target for taken branch
- IncrPC for not taken

## Write Back

- Only for bsr and br
- Incremented PC as data

# Branch to Subroutine

**Branch Subroutine (bsr):**  $Ra \leftarrow PC + 4$ ;  $PC \leftarrow PC + 4$

0x34	ra	disp
31-26	25-21	20-0

## IF: Instruction fetch

- $IR \leftarrow IMemory[PC]$
- $incrPC \leftarrow PC + 4$

## ID: Instruction decode/register fetch

- nop

## Ex: Execute

- $Target \leftarrow incrPC + SignExtend(IR[20:0]) \ll 2$

## MEM: Memory

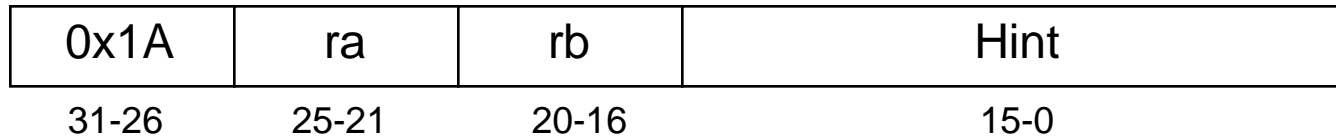
- $PC \leftarrow Target$

## WB: Write back

- $Register[IR[25:21]] \leftarrow oldPC$

# Jump

**jmp, jsr, ret:** Ra  $\leftarrow$  PC+4; PC  $\leftarrow$  Rb



## IF: Instruction fetch

- IR  $\leftarrow$  IMemory[PC]
- incrPC  $\leftarrow$  PC + 4

## ID: Instruction decode/register fetch

- B  $\leftarrow$  Register[IR[20:16]]

## Ex: Execute

- Target  $\leftarrow$  B

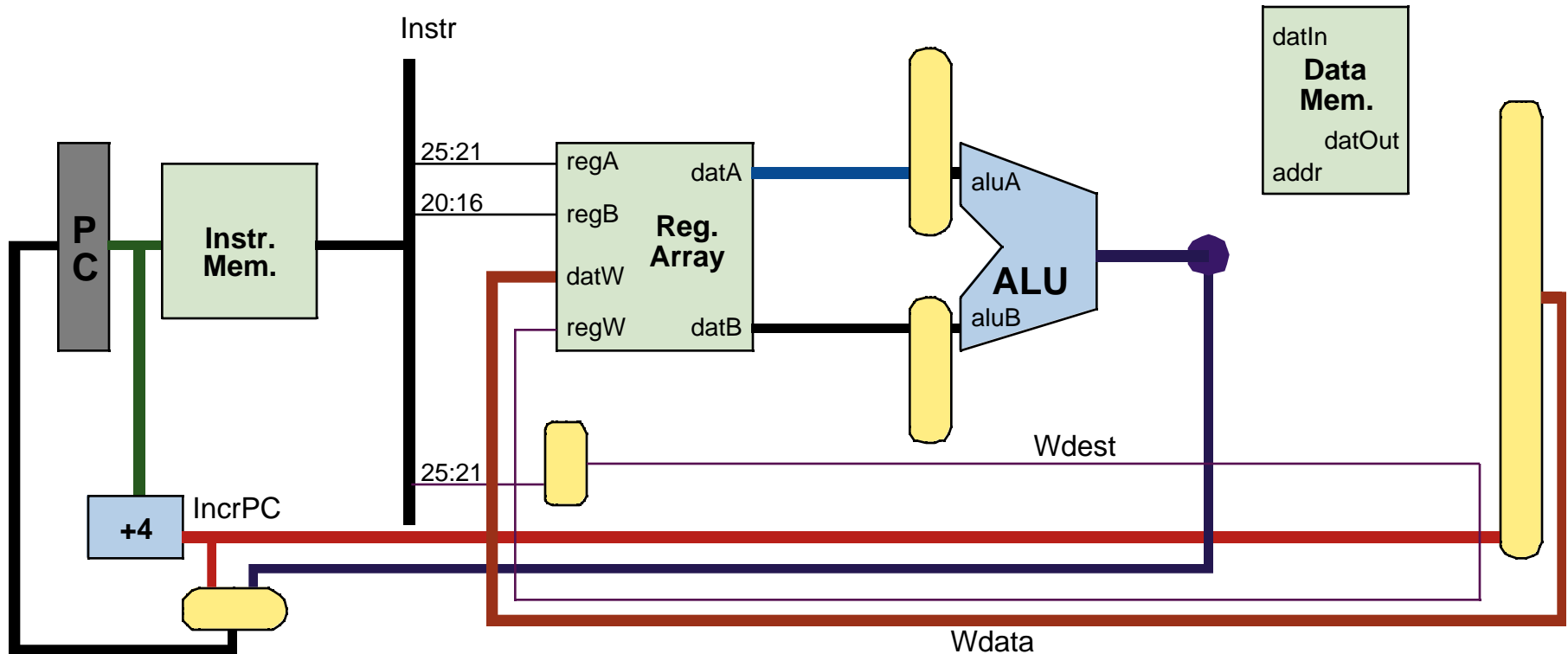
## MEM: Memory

- PC  $\leftarrow$  target

## WB: Write back

- IR[25:21]  $\leftarrow$  incrPC

# Active Datapath for Jumps



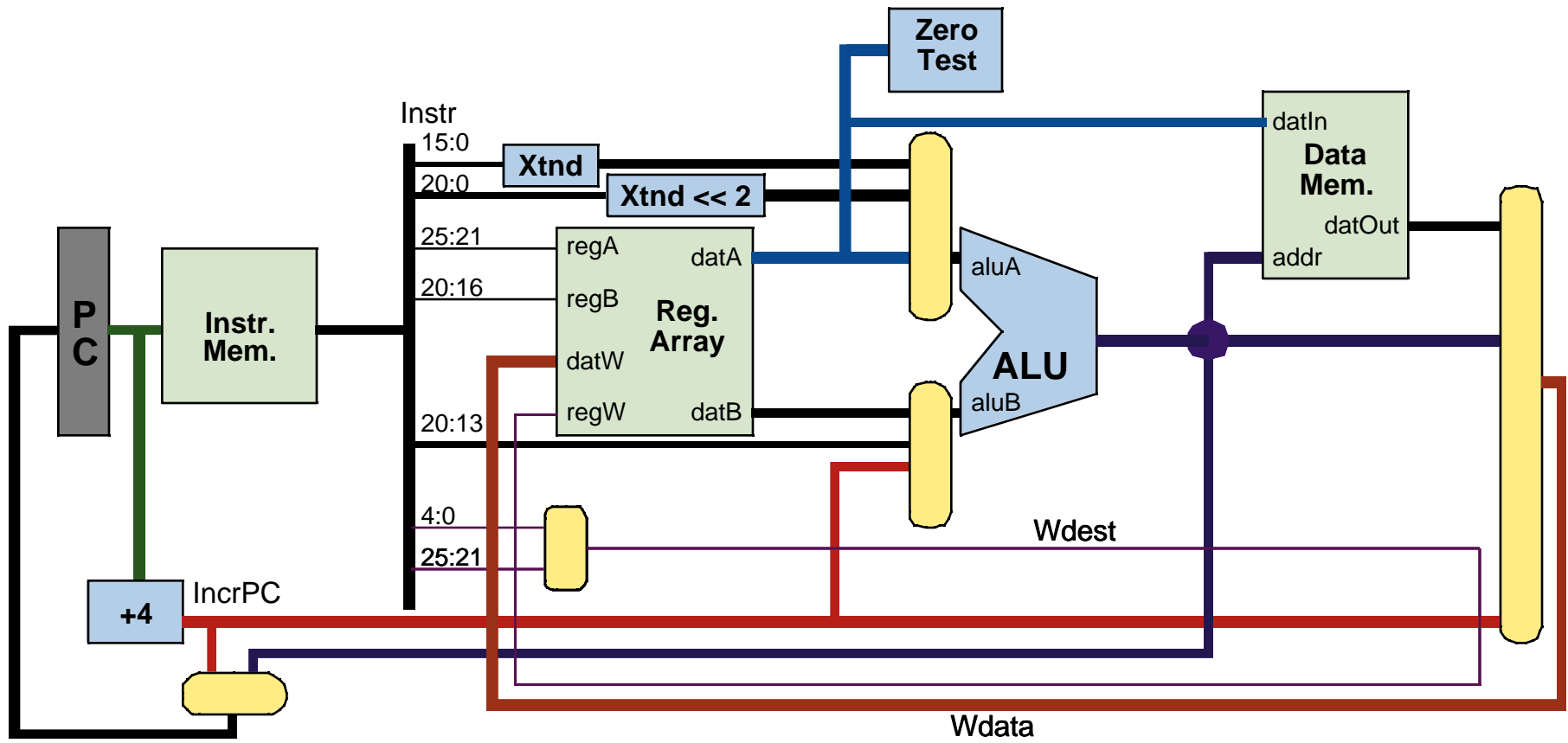
## ALU Operation

- Used to compute target
  - B input set to Rb
- ALU function set to select B

## Write Back

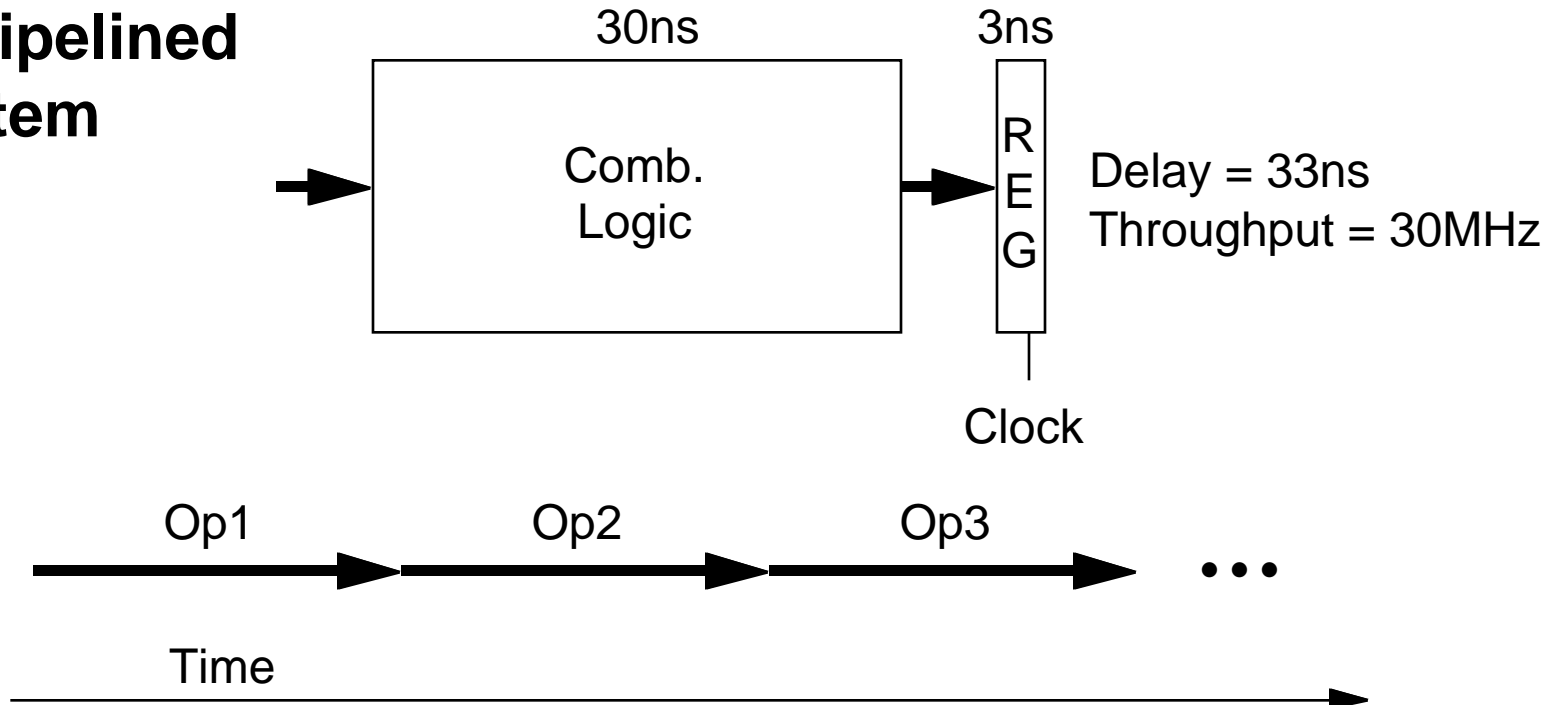
- To Ra
- IncrPC as data

# Complete Datapath



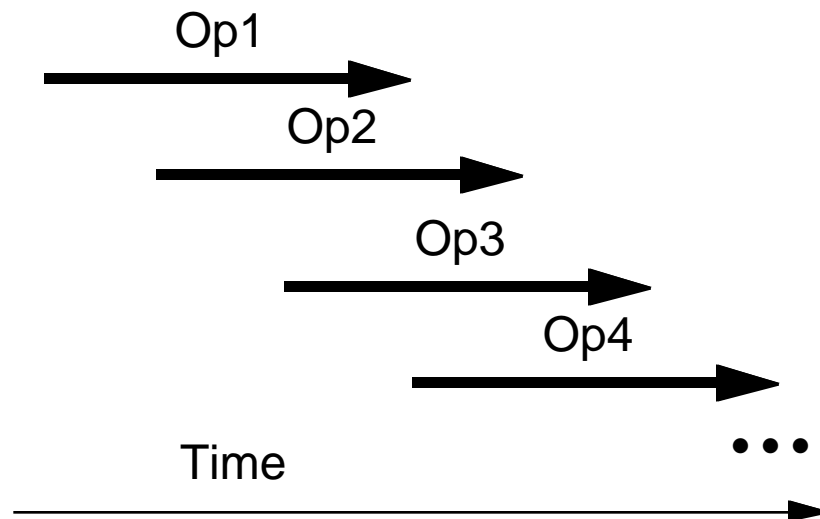
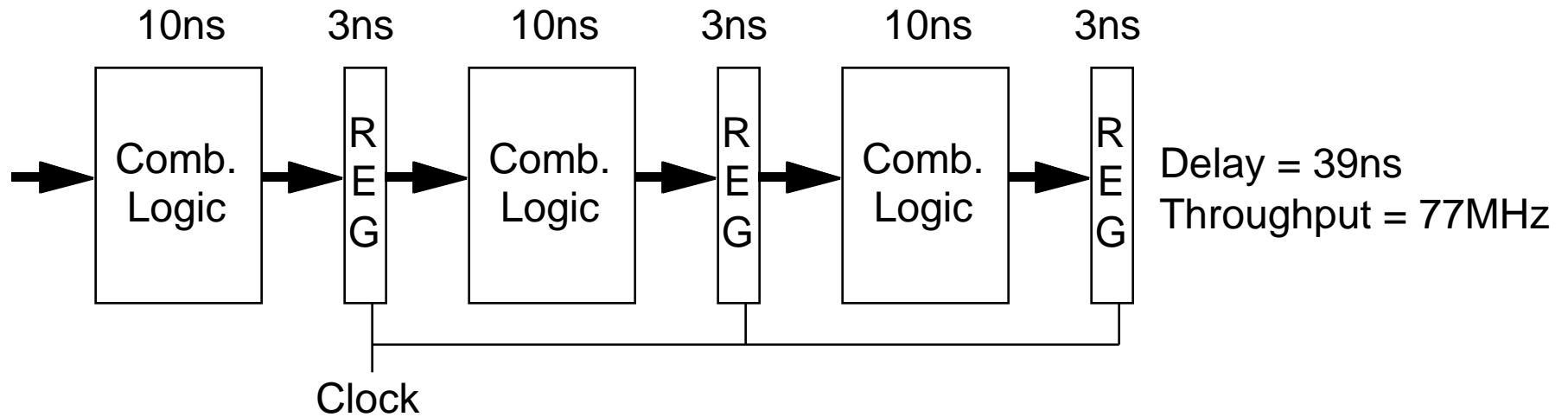
# Pipelining Basics

## Unpipelined System



- One operation must complete before next can begin
- Operations spaced 33ns apart

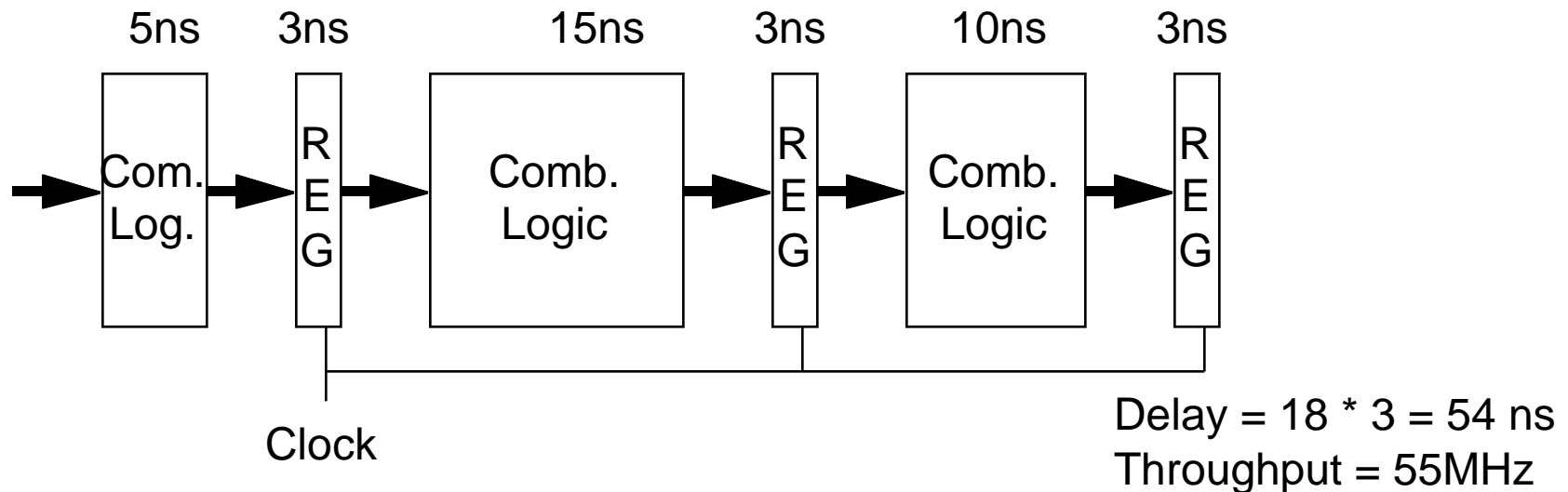
# 3 Stage Pipelining



- **Space operations 13ns apart**
- **3 operations occur simultaneously**

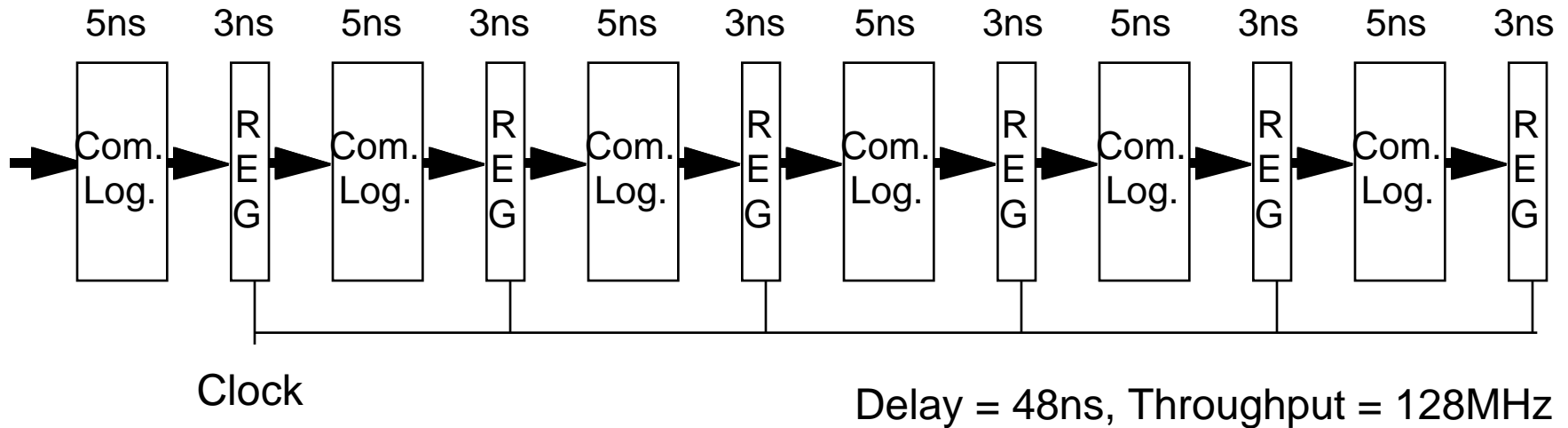


# Limitation: Nonuniform Pipelining



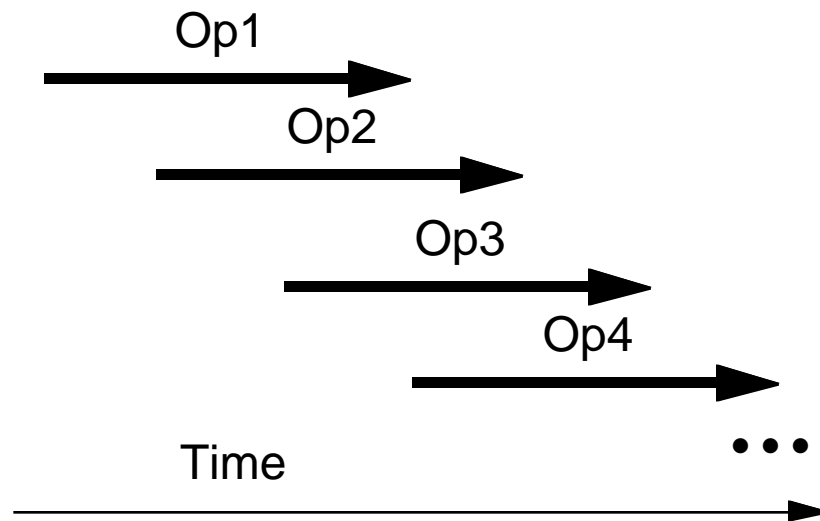
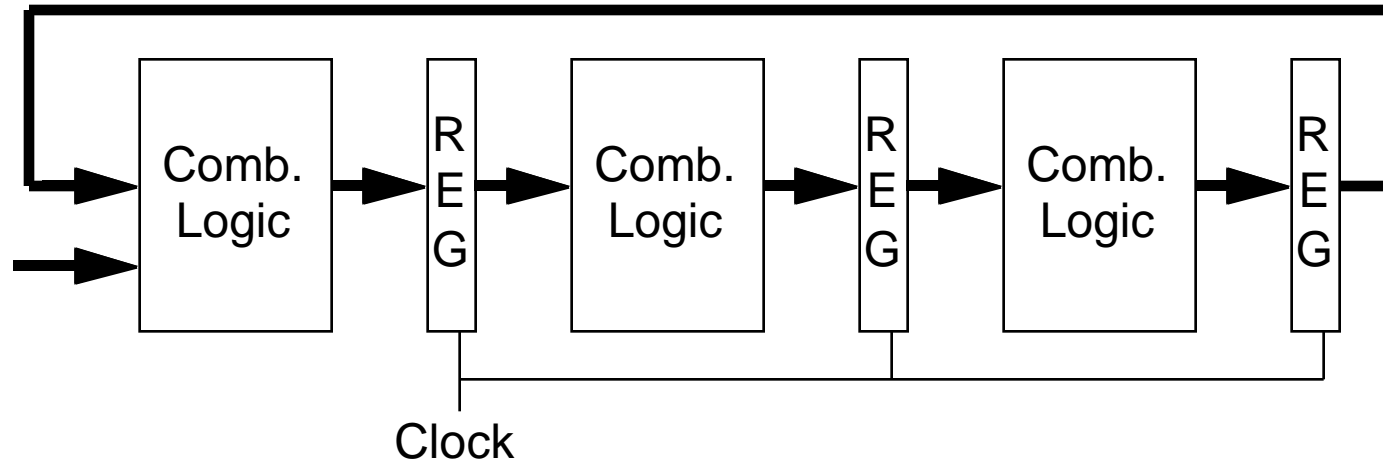
- **Throughput limited by slowest stage**
  - Delay determined by clock period \* number of stages
- **Must attempt to balance stages**

# Limitation: Deep Pipelines



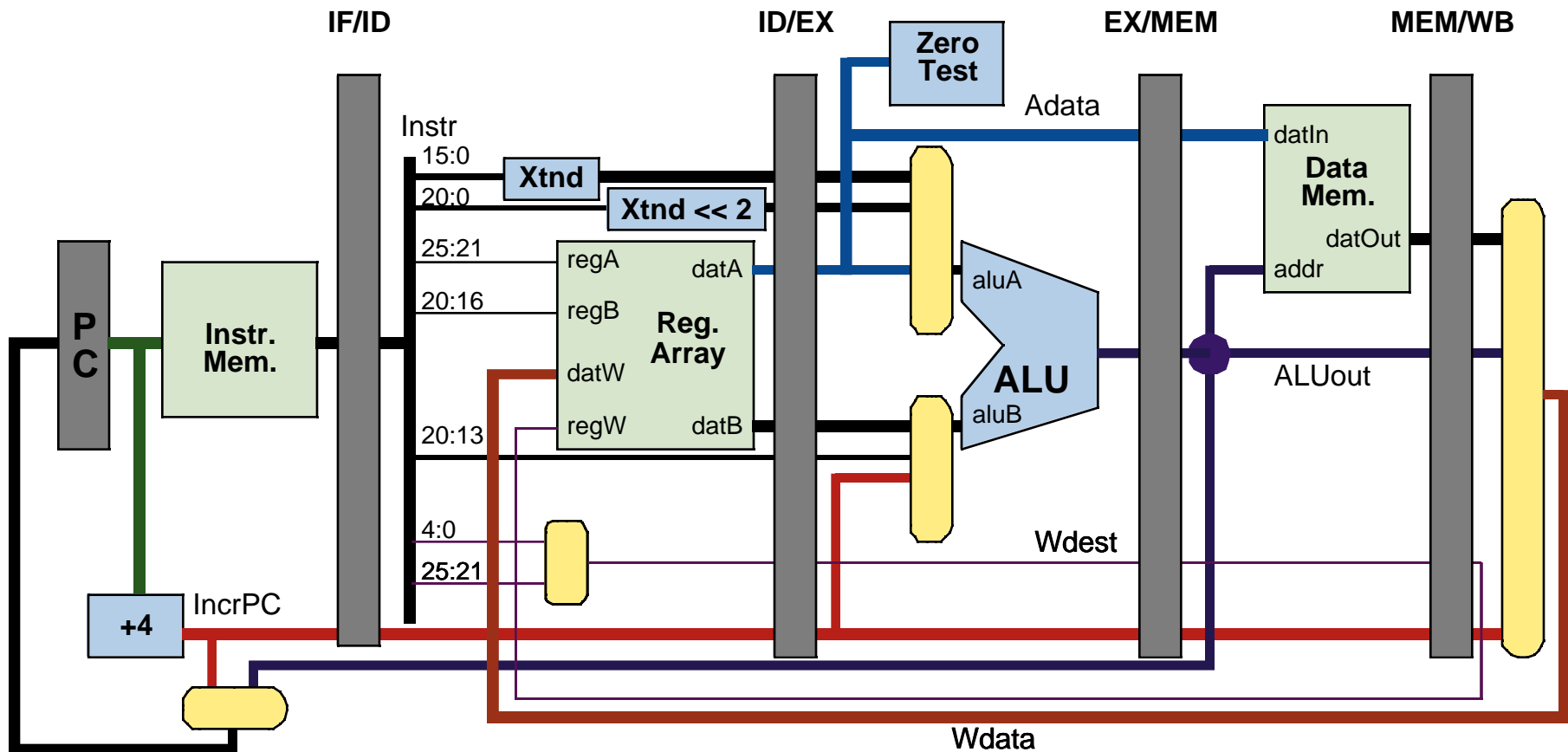
- **Diminishing returns as add more pipeline stages**
- **Register delays become limiting factor**
  - Increased latency
  - Small throughput gains

# Limitation: Sequential Dependencies



- Op4 gets result from Op1 !
- *Pipeline Hazard*

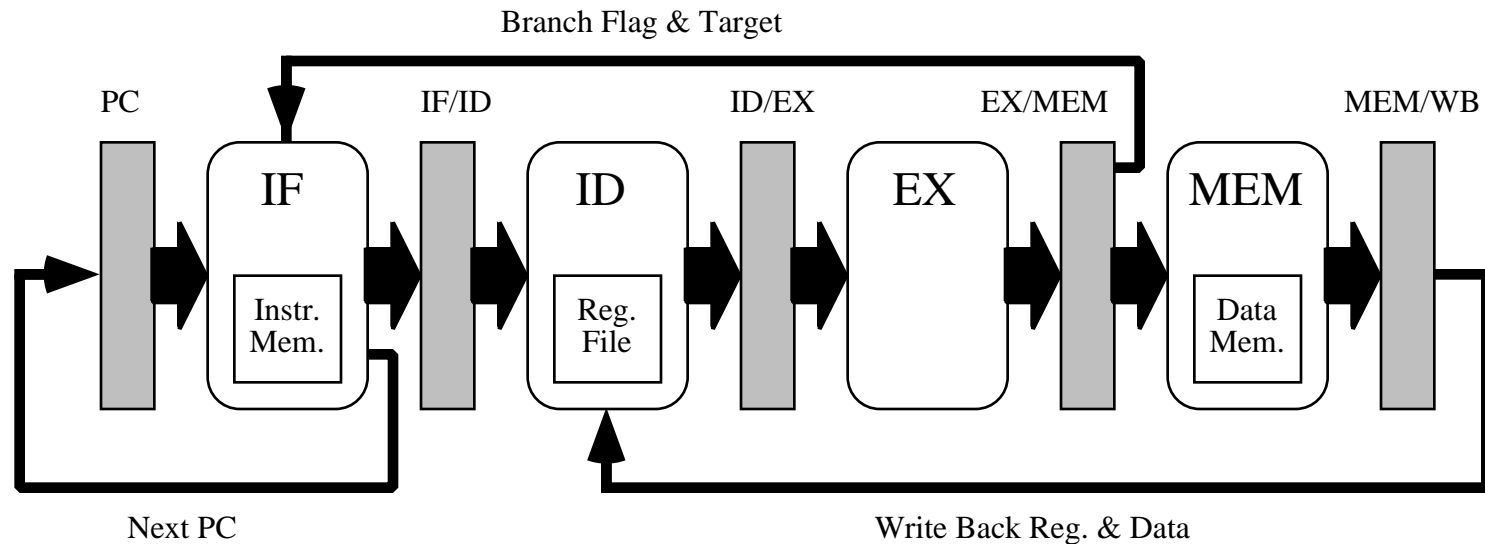
# Pipelined datapath



## Pipe Registers

- Inserted between stages
- Labeled by preceding & following stage

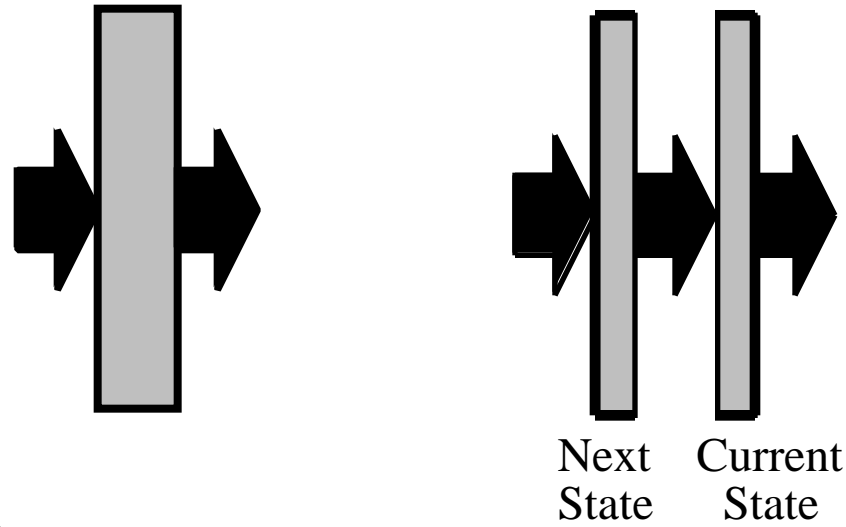
# Pipeline Structure



## Notes

- Each stage consists of operate logic connecting pipe registers
- WB logic merged into ID
- Additional paths required for forwarding

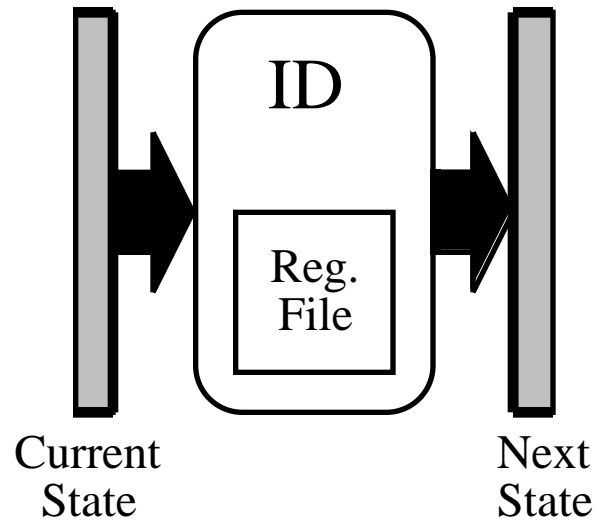
# Pipe Register



## Operation

- **Current State stays constant while Next State being updated**
- **Update involves transferring Next State to Current**

# Pipeline Stage



## Operation

- **Computes next state based on current**
  - From/to one or more pipe registers
- **May have embedded memory elements**
  - Low level timing signals control their operation during clock cycle
  - Writes based on current pipe register state
  - Reads supply values for Next state

# Alpha Simulator

## Features

- **Based on Alpha subset**
  - Code generated by `dis`
  - Hexadecimal instruction code
- **Executable available soon**
  - `AFS347/sim/solve_tk`

## Demo Programs

- `AFS347/sim/solve_tk/demos`

## Program Display

File	../demos/demo01.0	Load
0x0	43E07402	addq r31, 0x3, r2 # \$2 = 3
0x4	43E09403	addq r31, 0x4, r3 # \$3 = 4
0x8	47FF041F	W bis r31, r31, r31
0xc	47FF041F	M bis r31, r31, r31
0x10	40430404	E addq r2, r3, r4 # \$4 = 7
0x14	47FF041F	D bis r31, r31, r31
0x18	00000000	F call_pal halt
0x1c	47FF041F	bis r31, r31, r31

Hex-coded instruction

Pipe Stage

Treated as comment

– 32 –

The screenshot shows the Alpha Simulator interface with several components labeled:

- Run Controls:** Buttons for Quit, Go, Stop, and Reset.
- Speed Control:** A slider for Simulator Speed (10\*log Hz).
- Mode Selection:** Radio buttons for Wedged, Stall, and Forward.
- Pipeline Registers:** A table showing the state of pipeline registers for MEM, EX, ID, and IF stages.
- Current State:** A label pointing to the MEM Stage row in the pipeline registers table.
- Pipe Register:** A label pointing to the MEM Stage row in the pipeline registers table.
- Next State:** A label pointing to the EX Stage row in the pipeline registers table.
- Register Values:** A label pointing to the Register File table at the bottom.

MEM In	Exc	SPC	ALUdata	Rdata	Mem	Wbst
AOK: 0x8	0000000000000000	0000000000000000	-	31		
MEM Out	AOK: 0xc	0000000000000000	0000000000000000	-	31	
MEM In	Exc: 0xc	Adata	ALUdata	Mem	Wbst	Src
Exc: SPC	0000000000000000	0000000000000000	-	4	2	N
EX Out	AOK: 0x10	0000000000000003	0000000000000004			
EX In	Exc: 0x10	Adata	Bdata			
Exc: SPC	0000000000000000	0000000000000000				
ID Out	AOK: 0x14	0000000000000000	0000000000000000			
ID In	Exc: 0x14	47ff041f				
Exc: SPC	instr					
IF Out	AOK: 0x18	00000000				
IF In	000000000000001b					
PC Out	000000000000001c					



# Simulator ALU Example

## IF

- Fetch instruction

## ID

- Fetch operands

## EX

- Compute ALU result

## MEM

- Nothing

## WB

- Store result in Rc

```
0x0: 43e07402  addq  r31, 0x3, r2  # $2 = 3
0x4: 43e09403  addq  r31, 0x4, r3  # $3 = 4
0x8: 47ff041f  bis   r31, r31, r31  demo01.O
0xc: 47ff041f  bis   r31, r31, r31
0x10:40430404  addq  r2, r3, r4    # $4 = 7
0x14:47ff041f  bis   r31, r31, r31
0x18:00000000  call_pal          halt
```

```
# Demonstration of R-R instruction
.set noreorder
    mov  3, $2    demo01.s
    mov  4, $3
    nop
    nop
    addq $2, $3, $4
    nop
    call_pal 0x0
.set reorder
```

Tells assembler not to rearrange instructions

# Simulator Store/Load Examples

## IF

- Fetch instruction

## ID

- Get addr reg
- Store: Get data

## EX

- Compute EA

## MEM

- Load: Read
- Store: Write

## WB

- Load: Update reg.

### demo02.O

```
0x0: 43e17402  addq  r31, 0xb, r2 # $2 = 0xB
0x4: 43e19403  addq  r31, 0xc, r3 # $3 = 0xC
0x8: 43fff404  addq  r31, 0xff, r4 # $4 = 0xFF
0xc: 47ff041f  bis   r31, r31, r31
0x10:47ff041f  bis   r31, r31, r31
0x14:b4820005  stq   r4, 5(r2)    # M[0x10] = 0xFF
0x18:47ff041f  bis   r31, r31, r31
0x1c:47ff041f  bis   r31, r31, r31
0x20:a4a30004  ldq   r5, 4(r3)    # $5 = 0xFF
0x24:47ff041f  bis   r31, r31, r31
0x28:00000000  call_pal          halt
```

# Simulator Branch Examples

## IF

- Fetch instruction

## ID

- Fetch operands

## EX

- test if operand 0
- Compute target

## MEM

- Taken: Update PC to target

## WB

- Nothing

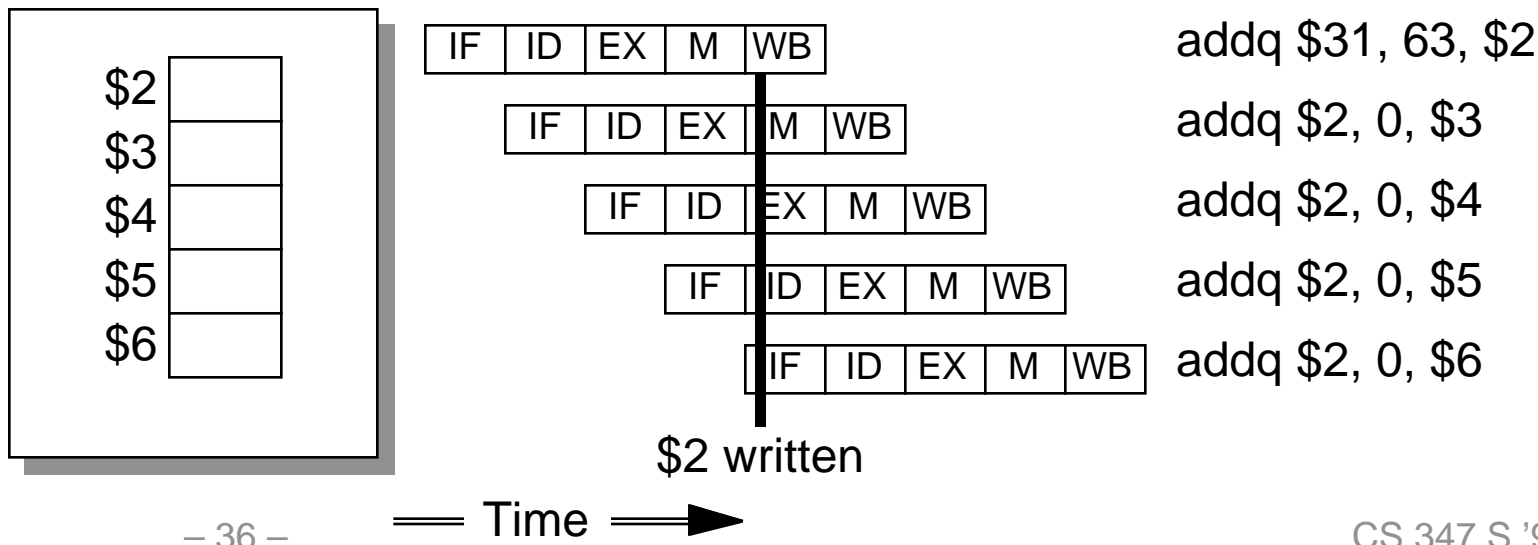
### demo3.0

```
0x0: 43e07402 addq r31, 0x3, r2 # $2 = 3
0x4: 47ff041f bis r31, r31, r31
0x8: 47ff041f bis r31, r31, r31
0xc: e4400008 beq r2, 0x30 # Don't take
0x10: 47ff041f bis r31, r31, r31
0x14: 47ff041f bis r31, r31, r31
0x18: 47ff041f bis r31, r31, r31
0x1c: f4400004 bne r2, 0x30 # Take
0x20: 47ff041f bis r31, r31, r31
0x24: 47ff041f bis r31, r31, r31
0x28: 47ff041f bis r31, r31, r31
0x2c: 40420402 addq r2, r2, r2 # Skip
0x30: 405f0404 addq r2, r31, r4 #Targ: $4 = 3
0x34: 47ff041f bis r31, r31, r31
```

# Data Hazards in Alpha Pipeline

## Problem

- Registers read in ID, and written in WB
- **Must resolve conflict between instructions competing for register array**
  - Generally do write back in first half of cycle, read in second
- **But what about intervening instructions?**
- **E.g., suppose initially \$2 is zero:**



# Simulator Data Hazard Example

## Operation

- Read in ID
- Write in WB
- Write-before-read register file

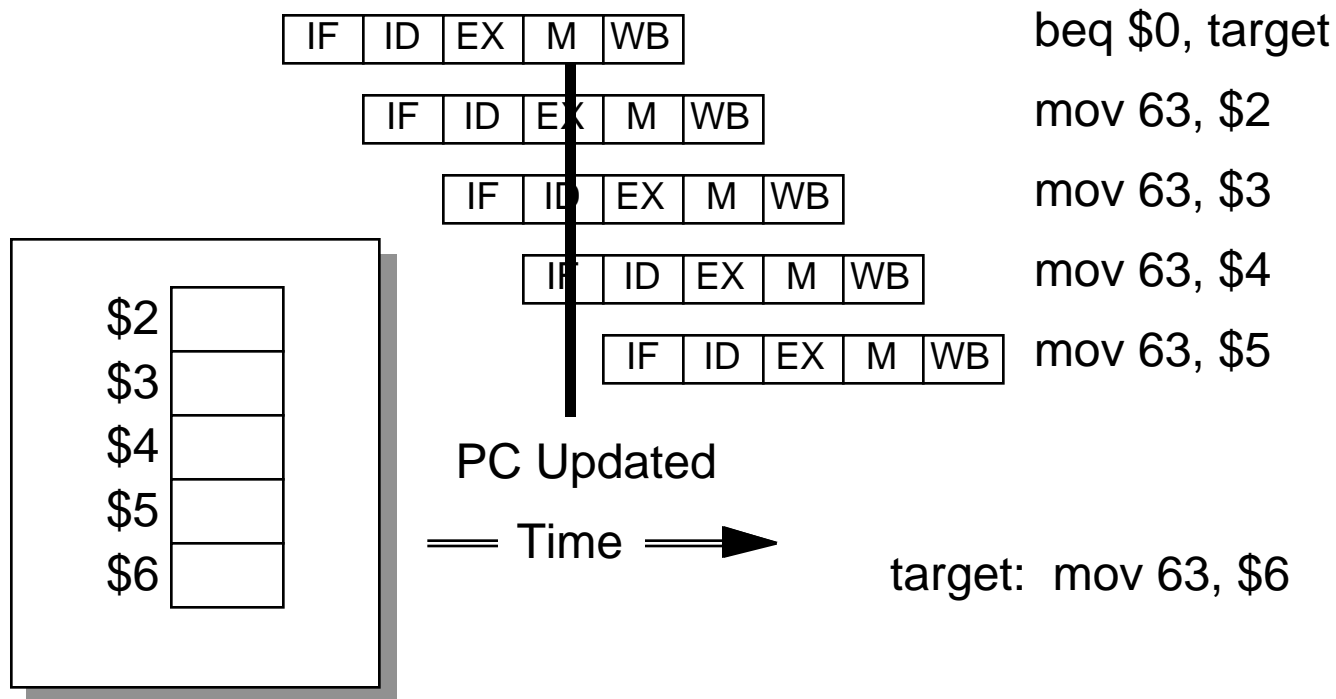
### demo04.O

```
0x0: 43e7f402  addq r31, 0x3f r2 # $2 = 0x3F
0x4: 40401403  addq r2, 0, r3    # $3 = 0x3F?
0x8: 40401404  addq r2, 0, r4    # $4 = 0x3F?
0xc: 40401405  addq r2, 0, r5    # $5 = 0x3F?
0x10:40401406  addq r2, 0, r6    # $6 = 0x3F?
0x14:47ff041f  bis  r31, r31, r31
0x18:00000000  call_pal          halt
```

# Control Hazards in Alpha Pipeline

## Problem

- Instruction fetched in IF, branch condition set in MEM
- When does branch take effect?
- E.g.: assume initially that all registers = 0



# Branch Example

## Branch Code (demo08.O)

```
0x0: e7e00005  beq   r31, 0x18      # Take
0x4: 43e7f401  addq  r31, 0x3f, r1  # (Skip) $1 = 0x3F
0x8: 43e7f402  addq  r31, 0x3f, r2  # (Skip) $2 = 0x3F
0xc: 43e7f403  addq  r31, 0x3f, r3  # (Skip) $3 = 0x3F
0x10: 43e7f404  addq  r31, 0x3f, r4  # (Skip) $4 = 0x3F
0x14: 47ff041f  bis   r31, r31, r31
0x18: 43e7f405  addq  r31, 0x3f, r5  # (Target) $5 = 0x3F
0x1c: 47ff041f  bis   r31, r31, r31
0x20: 00000000  call_pal                halt
```

# Conclusions

## **RISC Design Simplifies Implementation**

- Small number of instruction formats
- Simple instruction processing

## **RISC Leads Naturally to Pipelined Implementation**

- Partition activities into stages
- Each stage simple computation

## **We're not done yet!**

- Need to deal with data & control hazards