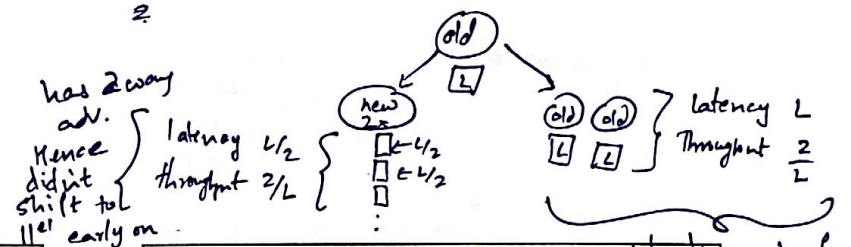
• Latency : time for a job.
• Throughput : Jobs in unit time.

has 2 way adv. Hence didn't shift to 111 early on



updating code for this freq is hard

ECE437: Introduction to Digital Computer Design

Mithuna Thottethodi
Chapter 2

Spring 2016

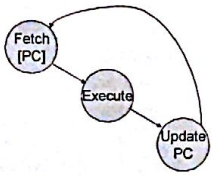
Instructions

- Instructions are the "words" of a computer
- ISA is its "vocabulary"
- With a few other things, this defines the interface of computers
  - But implementations vary
- We will study MIPS™ ISA
  - but the next ISA is not too hard after the first
- We won't write programs - EE362

car ↓ suv ↓ bus analogy.

Computer

- What does it do?
  - Fetch instruction from address in Program Counter (PC)
  - Execute instruction
  - Update PC to point to next instruction



Outline

- Instructions
  - Basics
  - Registers and ALU ops
  - Memory and load/store
  - Branches and jumps
  - And more . . .
- Read Chapter 2 and skim through Appendix B for MIPS ISA

Basics

- C statement
- f = (g + h) - (i + j)
- MIPS instructions*

```
add t0, g, h   // t0 = g + h
add t1, i, j   // t1 = i + j
sub f, t0, t1  // f  = t0 - t1
```

- Opcodes/mnemonic, operands, source/destination

In MIPS, first is destination as convention.

Basics

- Opcode: Specifies the kind of operation (mnemonic)
- Operands: input and output data (source/destination)
- Operands t0, t1 are temps
- One operation, two inputs, one output
- Multiple instructions for one C statement

→ • compilers became super good that it was used everywhere. BUT, they use simple instructions. Cannot use complex fn like FFT. => not used. They are clock and power/area kills. => removed

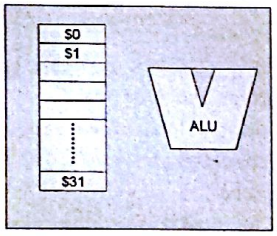
Why not have bigger instructions?

- Why not have "f = (g + h) - (i + j)" as one instruction?
- Church's thesis: A very primitive computer can compute anything that a fancy computer can compute
  - need only logical functions, read and write memory and data-dependent decisions
- Therefore, ISA selected for practical reasons: performance and cost, not computability
- Regularity tends to improve both
  - E.g., H/W to handle arbitrary number of operands
  - complex, slow and rarely usable and NOT NECESSARY

Registers and ALU ops

- Ok, I lied!
- Operands must be registers, not variables
  - add $8, $17, $18
  - add $9, $19, $20
  - sub $16, $8, $9
- MIPS has 32 registers $0-$31 (figure next slide)
- Registers $8, $9 are temps, $16 - f, $17 - g, $18 - h, $19 - i, $20 - j
- MIPS also allows one constant called "immediate"
  - later we will see immediate is 16 bits [-32768, 32767]

Registers and ALU



ALU ops

- Some ALU ops: i u
  - add, addi, addu, addiu (immediate, unsigned)
  - sub . . .
  - mul, div - weird!
  - and, andi
  - or, ori
  - sll, srl, . . .
- Why registers? fit in instructions, smaller is faster
- Are registers enough?

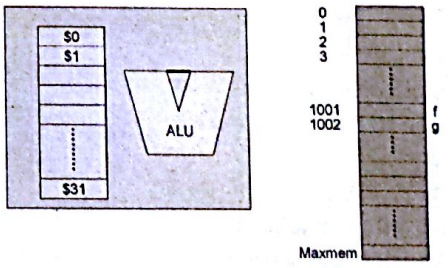
Memory and load/store

- But need more than 32 words of storage
- An array of locations M[addr], indexed by addr (figure next slide)
- Data movement (on words or integers)
  - load word for reg <-- memory
  - lw $17, 1002 // get input g
  - store word for reg --> memory
  - sw $16, 1001 // save output f
    - Note: destination LAST for stores!

load/store memory to/from registers

MIPS order reversed

Memory and load/store

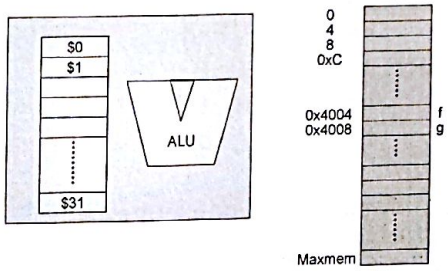


Byte vs. Word addresses

- I lied again!
  - we need to address bytes for character strings
  - words take 4 bytes
  - therefore, word addresses must be multiples of 4
- figure next slide

Memory and load/store

to look at addr in steps of 4.



Memory and load/store

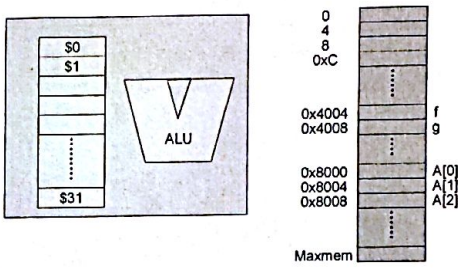
- Important for arrays imp
  - A[i] = A[i] + h (figure next slide)
  - # $8 -> temp, $18 -> h, $21 -> (i x 4)
  - Astart is 8000, A is an array of words NOT bytes

```
lw $8, Astart($21)      // or 8000($21)
add $8, $18, $8
sw $8, Astart($21)
```

addr = Val$_{21}$ + 0x8000

- MIPS has other load/store for bytes and halfwords

Memory and load/store



Endian

- Word storage
- Word = bytes (0x400,0x401,0x402,0x403)?
- Word = bytes (0x403,0x402,0x401,0x400)?
  - It depends…
- Big endian: MSB at address xxxxxx00
  - e.g., IBM, SPARC
- Little endian: MSB at address xxxxxx11
  - e.g., Intel x86
- Mode selectable
  - e.g., PowerPC, MIPS

↳ In case they must talk, a network endian is set. one sends raw & other converts.

Branches and Jumps

```
While ( i != j) {
   j= j + i;
   i= i + 1;
}
```

- HLL-> Assembly, $8 is i, $9 is j

```
Loop: beq $8, $9, Exit
      // note that the condition is reversed
      add $9, $9, $8
      addi $8, $8 , 1
      j   Loop
Exit:
```

Branches and Jumps

- Better yet:

```
        beq $8, $9, Exit        // not !=
Loop:   add $9, $9, $8
        addi $8, $8 , 1
        bne $8, $9, Loop
Exit:
```

- Let compilers worry about such optimizations

Branches and Jumps

- What does bne do really?
  - read $8; read $9, compare
  - set PC = PC + 4 or PC = Target
- To do compares other than "equal" or "not equal"
  - e.g., blt $8, $9, Target // assembler pseudo-instruction
- expands to

```
slt $1, $8, $9    // $1 = ($8 < $9) = ($8 - $9 < 0)
bne $1, $0, Target    // $0 is always 0
```

not implemented in hardware. converts to this

assembler

assembler reserves this.

You could instead not use assembler and do what you wish. That means no short cut like blt.

Branches and Jumps

- Other MIPS branches/jumps
  - beq $8, $9, imm
  - // if ($8 == $9) PC = PC + imm<<2 else PC += 4
- bne ...
- slt, sle, sgt, sge
  - with immediate, unsigned
- j addr // PC = addr
- jr $12 // PC = $12
- jal addr // $31 = PC+4; PC = addr; used for procedure calls

$0 is reserved for 0 value

$31 is reserved for return addr

jr $31 can return from a jump.

done by hardware itself. Not software convention like blt.

Assembly Exercise

```
for(i=0;i<n;i++) {
  blah blah
}
```

```
Init:   mov $8,0
Test:   slt $5,$8,$9
        beq $5,$0,Exit
Loop:   blahblah
        blahblah
Ind:    addi $8,$8,1
        b Test
Exit:
```


⟹ stack : software construct.

Say fn calls a fn.
$31 gets overwritten.
This is bad.
Hence store value in memory] the stack.

Assembly Exercise

- HLL-> assembly

```
if (a==b)
   a = a+b
else
   b = a+b
```


why branch better than jump?

Can use $4 & $5 to pass param to f^n.
$2 is return addr.

(Always good to make space before writing. ☺)

Procedure Calls

- See section 2.7 for more details
  - save registers
  - set up parameters
  - call procedure
  - get results
  - restore registers

  Proc: save more registers
  do function
  set up results
  restore more registers
  return
- jal is the only special instruction: the rest is software convention
- jal addr // $31 = PC+4; PC = addr; used for procedure calls

Procedure Calls

- An important data structure is the stack
- Stack grows from larger to smaller addresses
- $29 is the stack pointer, it points just beyond valid data. Why?
- Push $2:
  - addi $29, $29, -4 (↑ 1 word in memory)
  - sw $2, 4($29)

  Pop $2:
  lw $2, 4($29)
  addi $29, $29, 4
- The order cannot be changed. Why? interrupts

If we switch to have sw first.
→ say sw finished. To add yet.
→ interrupt comes now.
↳ will overwrite memory ⇒ lose data

Hence this order

Procedure Example

- HLL code for swap

```
{ temp = v[k];          /* code for swap */
  v[k] = v[k+1];
  v[k+1] = temp;
  return (temp); }
```

- Corresponding assembly code

```
swap:  addi  $29, $29, -8   // save registers
       sw    $15, 4($29)
       sw    $16, 8($29)
```


Procedure Example

$4 is v
$5 is k

```
muli*  $2, $5, 4     // procedure body; $2 == k*4   (for memory word size is 4)
add    $2, $4, $2    // $2 == v + k*4 == address of v[k]
lw     $15, 0($2)      // $15 == v[k]
lw     $16, 4($2)      // $16 == v[k+1]
sw     $16, 0($2)      // v[k] == $16
sw     $15, 4($2)      // v[k+1] == $15
mov*   $2, $15         // return value
lw     $15, 4($29)     // restore registers
lw     $16, 8($29)
addi   $29, $29, 8
jr     $31             // return
```


Procedure Call Example

- Now, calling the procedure
  - HLL code

```
return_value = swap(arr_A, i);  /* call swap */
```

```
mov   $4, $18     // first parameter
mov   $5, $17     // second parameter
jal   swap
mov   $20, $2     // copy return value
```


Layers of Software

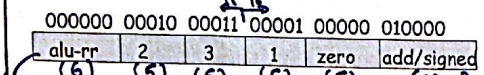
- Notation – program: input data -> output data
  - executable: input data -> output data
  - loader: executable file -> executable in memory
  - linker: object files -> executable file
  - assembler: assembly file -> object file
  - compiler: HLL file -> assembly file
  - editor: editor commands -> HLL file
- Only possible because programs can be manipulated as data
  - "stored program computer", "Von Neumann architecture".

↳ allows to create exe. by user.

MIPS Machine Language

- All 32-bit instructions
- Assembly: add $1, $2, $3
  - 14 ASCII (or Unicode) codes

  61 64 64 20 24 31 2C 20 24 32 2C 20 24 33 (hex)
- Machine code: 1 word

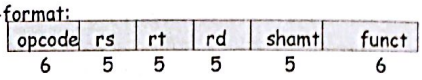


R

alu operation
register to register
dest
shift amt.
6 bits here is a signed add

Instruction Format

- R-format:

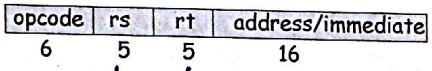


- Digression:
- How do you store the number 4,392,976?
- Same as add $1, $2, $3
- Stored program: instructions are represented as numbers
  - programs can be read/written in memory like numbers

Instruction Format

- Other R-format: addu, sub*, and,or, etc
- Assembly: lw $1, 100($2)
- Machine:

  100011 00010 00001 0000000001100100
  lw 2 1 100 (in binary)
- I-format:



limit to 2^{16}

I

source dest.

Instruction Format

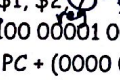
- I-format also used for ALU ops with immediates
  - addi $1, $2, 100
  - 001000 00010 00001 0000000001100100
- What about constants larger than 16 bits = [-32768, 32767]?
  - e.g., 0000 0000 0000 1100 0000 0000 0000 1111?

  lui $4, 12 // $4 <- 0x000C0000
  ori $4,$4,15 // $4 <- 0x000C000F
- All loads and stores use I-format

immediate allows only 16 bits. So, lui (load upper immediate) is done for first 16, then does ori

Instruction Format

- beq $1, $2, 7
- 000100 00001 00010 0000 0000 0000 0111
- PC = PC + (0000 0111 << 2) // word offset
- Finally, J-format:



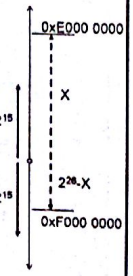
- addr is weird in MIPS:
  - 4 MSB of PC:addr:00 (4+26+2 = 32)

no. of words not bytes!!!
reg reg offset
concat
∵ word size is 4.

J

MIPS: "branch" vs "jump"

- Branch: 16 bit displacement
  - Current word +/- 2^{15}
  - Range is symmetric
    - $Range_{(forward)} = Range_{(backward)} = 2^{15}$
- Jump: 26 lower-bit concatenation in word address
  - Jump anywhere within a 2^{26} segment
  - Range may be asymmetric
    - $Range_{(forward)} + Range_{(backward)} = 2^{26}$



branch allows jump in 2^{15} # above/below current location.

Jump allows movement anywhere provided the top 4 bits are fixed by PC.

If PC = 0xA---

↳ then can jump in A region only

2

Summary: Instruction Formats

- R-format: opcode rs rt rd shamt funct

6	5	5	5	5	6

- I-format: opcode rs rt address/immediate

6	5	5	16

- J-format: opcode addr

6	26

- Instr decode
  - Theory: Inst bits -> identify instrs -> control signals
  - Practice: Instruction bits -> control signals

Memory (Hardware detour)

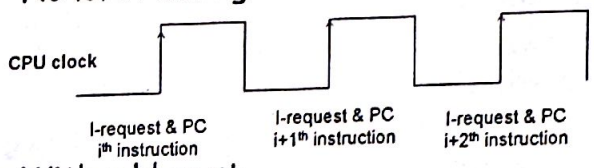
- Used for instruction fetch and data access ld/st
- Memory is big and slow
  - takes 300 CPU clocks today! (ch5)
- (1) Memory can allow only one of instruction fetch or ld/st data access at any given time → need to arbitrate between I-fetch and D-access
  - We will alleviate this in ch5 but will still need arbitration

Memory

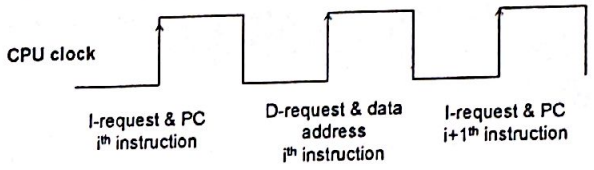
- While one could do both I fetch and D-access done one after the other in one (slow) CPU clock, our specific FPGA memory needs two clocks -one each for I-fetch and D-access
  - So our lab design is "mostly single-cycle CPU": one cycle for all instructions except ld and st, and two cycles for ld/st
  - Still you need arbitration so memory does either I-fetch or D-access at a time

Lab memory timing

- No ld/st among instructions



- With a ld or st



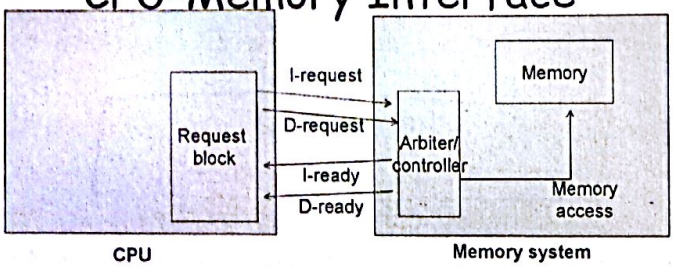
Memory arbitration

- How is this timing realized? Via request-ready handshake + arbitration
- The CPU has a hardware block called "request block" where the CPU generates I-request and D-request signals - non ld/st instructions generate only I-request and ld/st generate both I-request (for I-fetch) and D-request (for ld/st of data)
- The requests go to the "Arbiter block"

Memory arbitration

- The arbiter block chooses one of I-request or D-request if both are active (ld/st) OR I-request if D-request is not active (non-ld/st)
- If both active,
  - the D-request is for current instruction and I-request is for the next instruction
  - D-request gets priority to complete the current instruction before going to next
- The chosen request accesses memory

CPU-Memory Interface



- Request is in CPU and arbiter is in memory system
- Do not break this interface by going around it - else you will suffer later

Memory arbitration

- When access complete (I or D), the arbiter/controller asserts the corresponding ready (I-ready or D-ready)
  - This is how CPU knows access is complete - this is important in 3 slides
- VERY IMPORTANT: once a D-access is complete, de-assert the D-request
  - Else arbiter will keep choosing that request
  - Changes behavior for pipeline.

Memory arbitration - Detail 1

- For ld/st, the instruction is latched within memory system WHILE data access occurs (else instruction would vanish while data access occurs!)
  - During the middle clock of previous timing diagram
  - This latching is done for you in the code given to you

Memory arbitration- Detail 2

- Lab has two clocks - CPU clock and RAM clock
  - CPU clock is the main clock
  - RAM clock an internal detail to be ignored
  - RAM clock happens to be 2x CPU clock
    - Does not match reality
    - Done to make our specific FPGA work for mostly single-cycle CPU
    - Fixed in the next slide

Memory arbitration

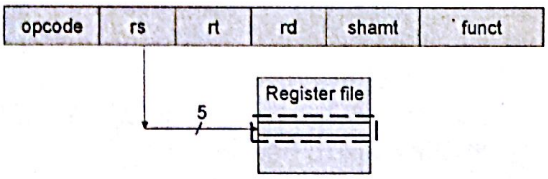
- In lab, memory is variable latency - so your design should work at different latencies (important in real world)
  - Mostly single-cycle is at the lowest latency
  - Longer latencies → multiple clocks per I-fetch or D-access
  - You MAY NOT assume memory will complete within one cycle → you MUST wait for I-ready or D-ready
  - I-ready, D-ready in one cycle at lowest latency & in more cycles at longer latencies

Why?

- Variable latency checks for timing independence of design - important skill
- Single memory, so we don't change from two memories (in single cycle) to one memory (in pipelining) - easier for you
- GENERAL- you should test and debug every block BOTH separately AND after merging with rest of design BEFORE going to the next block else you will have a mess that can't be debugged

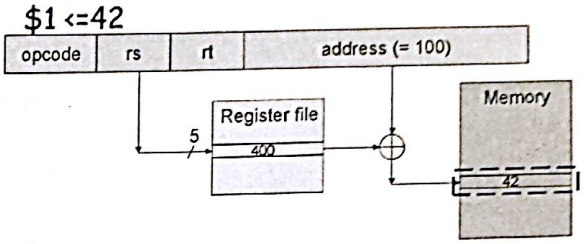
Addressing modes

- There are many ways of accessing data
- 1. Register addressing
  - add $1, $2, $3



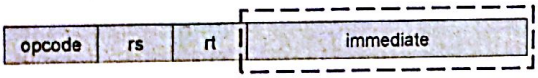
Addressing Modes

- 2. Base addressing (aka displacement)
  - lw $1, 100($2) // $2 = 400, M[500] = 42, $1 <=42



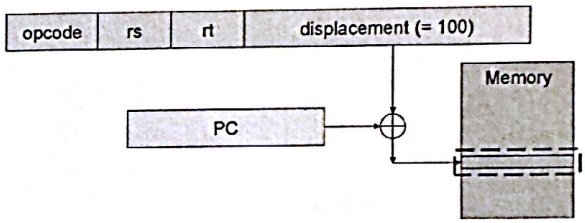
Addressing Modes

- 3. Immediate addressing
  - addi $1, $2, 100



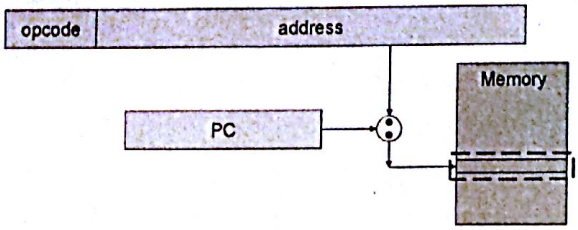
Addressing Modes

- 4. PC relative addressing
  - beq $1, $2, 100 // if ($1 == $2) PC = PC + 100



Addressing Modes

- Pseudodirect addressing
  - j Loop // instruction contains address*



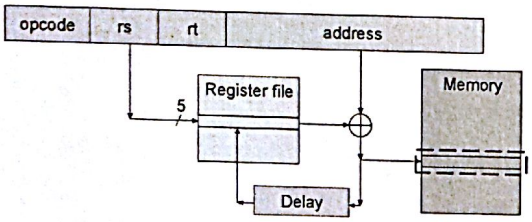
Addressing Modes

- Not found in MIPS
- 5. Indexed: add two registers - base + index
- 6. Indirect: M[M[addr]] - two memory references
- 7. Autoincrement/decrement: add data size
- 8. Autoupdate - found in IBM PowerPC, HP PA-RISC
  - like displacement but update register

Addressing Modes

- Autoupdate

lwupdate $1, 4[$2] // $1 == M[4+$2]; $2 == $2 + 4



Addressing Modes

```
for (i = 0, I < N, i +=1)
  sum += A[i];
```

- $7 - sum, $8 - address of a[i], $2 - tmp

```
inner:                    new inner:
  lw $2, 0($8)      }  →    lwupdate $2, 4($8)
  addi $8, $8, 4    }
  add $7, $7, $2            add $7, $7, $2
```

- any problems with new inner ?

Chosing ISA:
-> very application specific.
-> might have system with less memory. Might want to have more & powerful instructions.

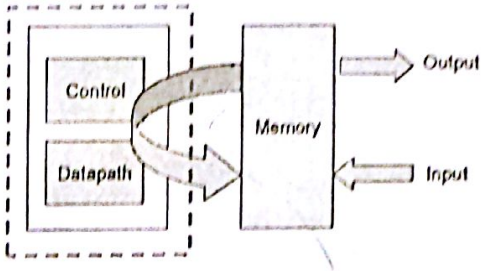
ECE437: Introduction to Digital Computer Design

Mithuna Thottethodi

Chapter 4

Spring 2016

Processor Implementation



Outline

- Datapath - single cycle
  - single instruction, 2's complement, unsigned
- Control

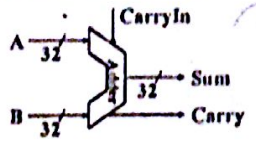
runs instruction. data flows here.

Datapath for Instructions

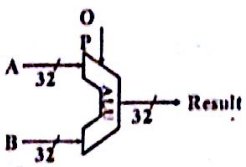
- Single cycle datapath
- Compose using well-understood pieces
  - Mux, flip-flops and gates
  - ALU
  - Register file

Comb. Logic Elements

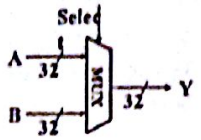
- Adder



- ALU

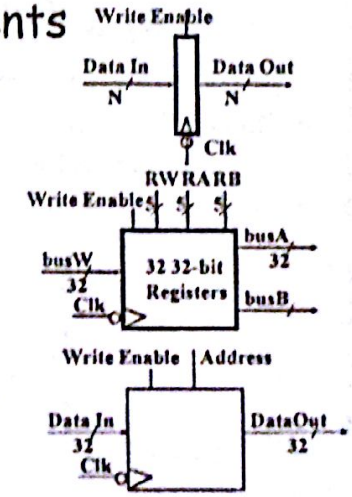


- Mux

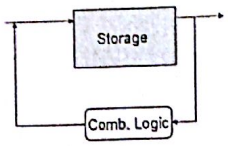


Storage Elements

- Register
  - for PC
- Register file
  - 32 registers
  - 2 read ports/buses
  - 1 write port/bus
- Memory
  - 1 input bus
  - 1 output bus
  - Not bidirectional



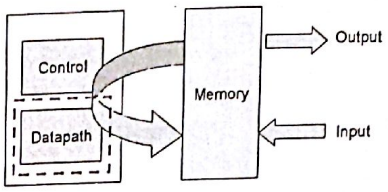
Computer as State Machine



- Storage elements
  - Memory, Register file, PC
- Combinational elements
  - ALUs, Adders, Muxes

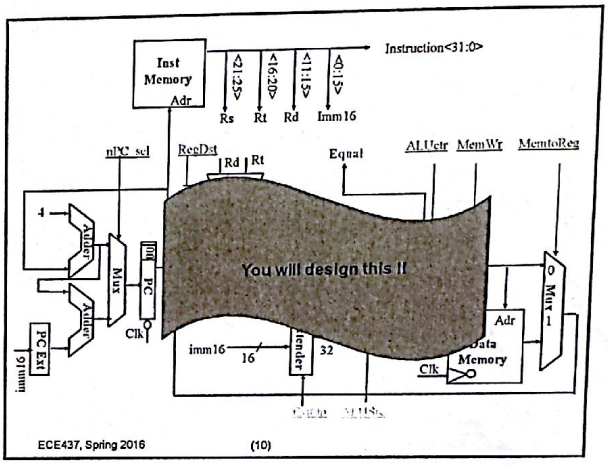
Processor Implementation

- This Lecture: Datapath



Datapath - 1 CPI

- Assumption: Get one whole instruction done in one long cycle
  - fetch, decode/read operands, execute, memory, writeback
  - useful way to represent steps and identify required datapath elements: RTL
- For single instruction
- Put it together



Outline

- Datapath - single cycle
  - single instruction, 2's complement, unsigned
- Control

Register Transfer Language

- RTL gives the meaning of the instructions
- All start by fetching the instruction

op | rs | rt | rd | shamt | funct = MEM[PC]
op | rs | rt | Imm16 = MEM[PC]

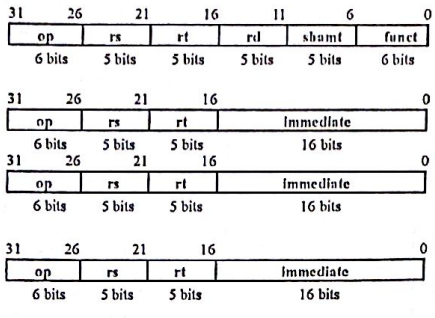
inst	Register Transfers	
ADDU	R[rd] <- R[rs] + R[rt];	PC <- PC + 4
SUBU	R[rd] <- R[rs] – R[rt];	PC <- PC + 4
ORi	R[rt] <- R[rs] OR zero_ext(Imm16);	PC <- PC + 4
LOAD	R[rt] <- MEM[R[rs] + sign_ext(Imm16)];	PC <- PC + 4
STORE	MEM[R[rs] + sign_ext(Imm16)] <- R[rt];	PC <- PC + 4
BEQ	if (R[rs] == R[rt]) then PC <- PC + sign_ext(Imm16)] || 00 else PC <- PC + 4	

pad with 1 or 0 according to Imm16[15]

pad end with 2 zeroes.

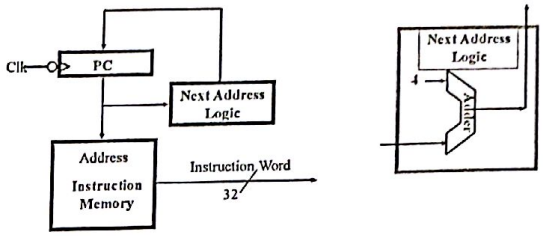
A Simple Implementation

- ADD and SUB
  - addU rd, rs, rt
  - subU rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE Word
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16



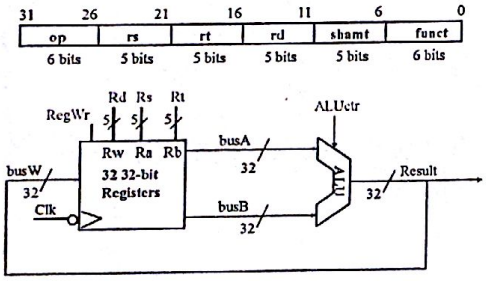
Fetch Instructions

- Fetch instruction, then update PC
- PC updated (at the end of) every cycle
  - What if no branches or jumps?



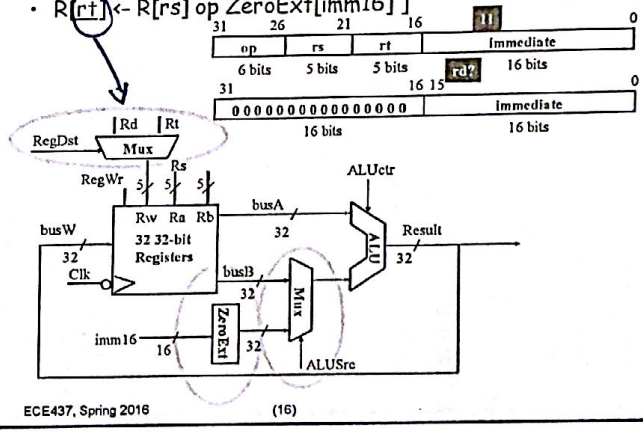
ALU Instructions

- R[rd] <- R[rs] op R[rt] Example: addU rd, rs, rt
  - Ra, Rb, and Rw come from instruction's rs, rt, and rd fields
  - ALUctr and RegWr: control logic after decoding the instruction



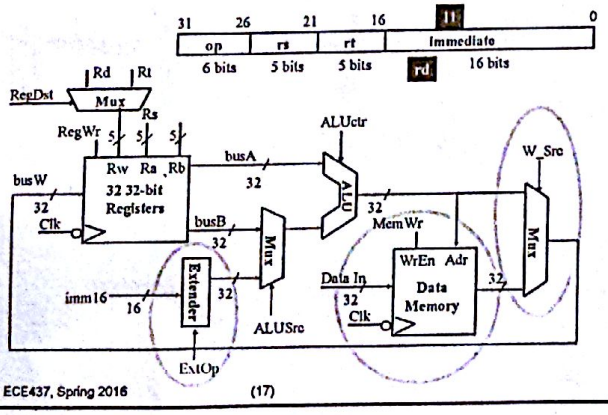
Logical operation with Immediate

- R[rt] <- R[rs] op ZeroExt[imm16]]



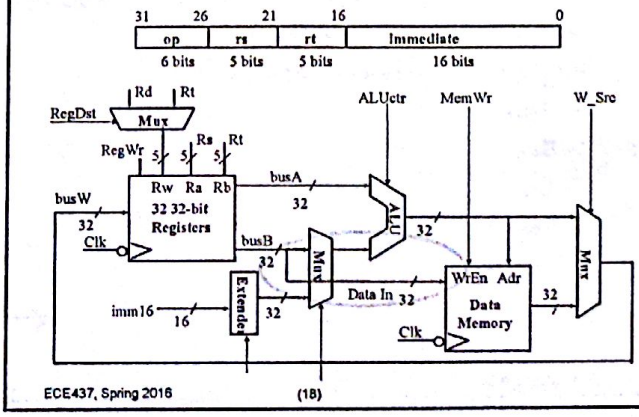
Load Instruction

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

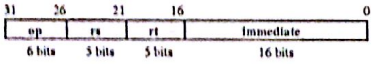


Store Instruction

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



Conditional Branch Instruction

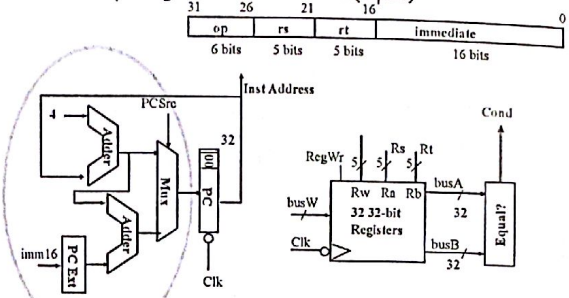


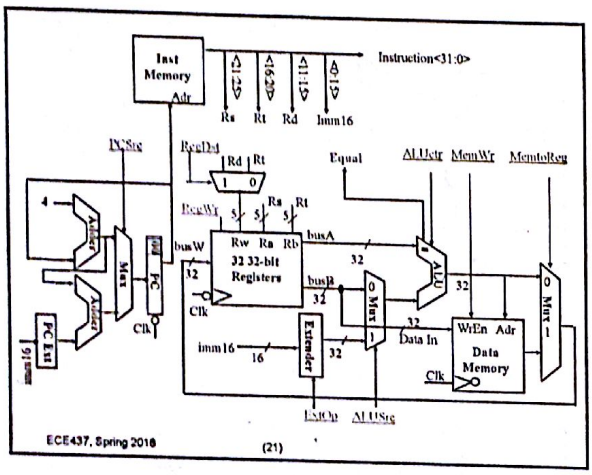
- beq rs, rt, imm16
  - IR = Mem[PC] // Fetch the instruction from memory
  - COND <- R[rs] == R[rt] // Calculate the branch condition
  - if (COND eq 0) // Calculate the next instruction's address
    - PC <- PC + 4 + (SignExt(imm16) x 4)
  - else
    - PC <- PC + 4

What is this?

Datapath for 'beq'

- beq rs, rt, imm16
  - Datapath generates condition (equal)





ORI

- ORI is not in the book
- ORI shows that some instructions need zero extension instead of sign extension
  - Logical vs. arithmetic ops
- What will change in the datapath if ORI is absent?

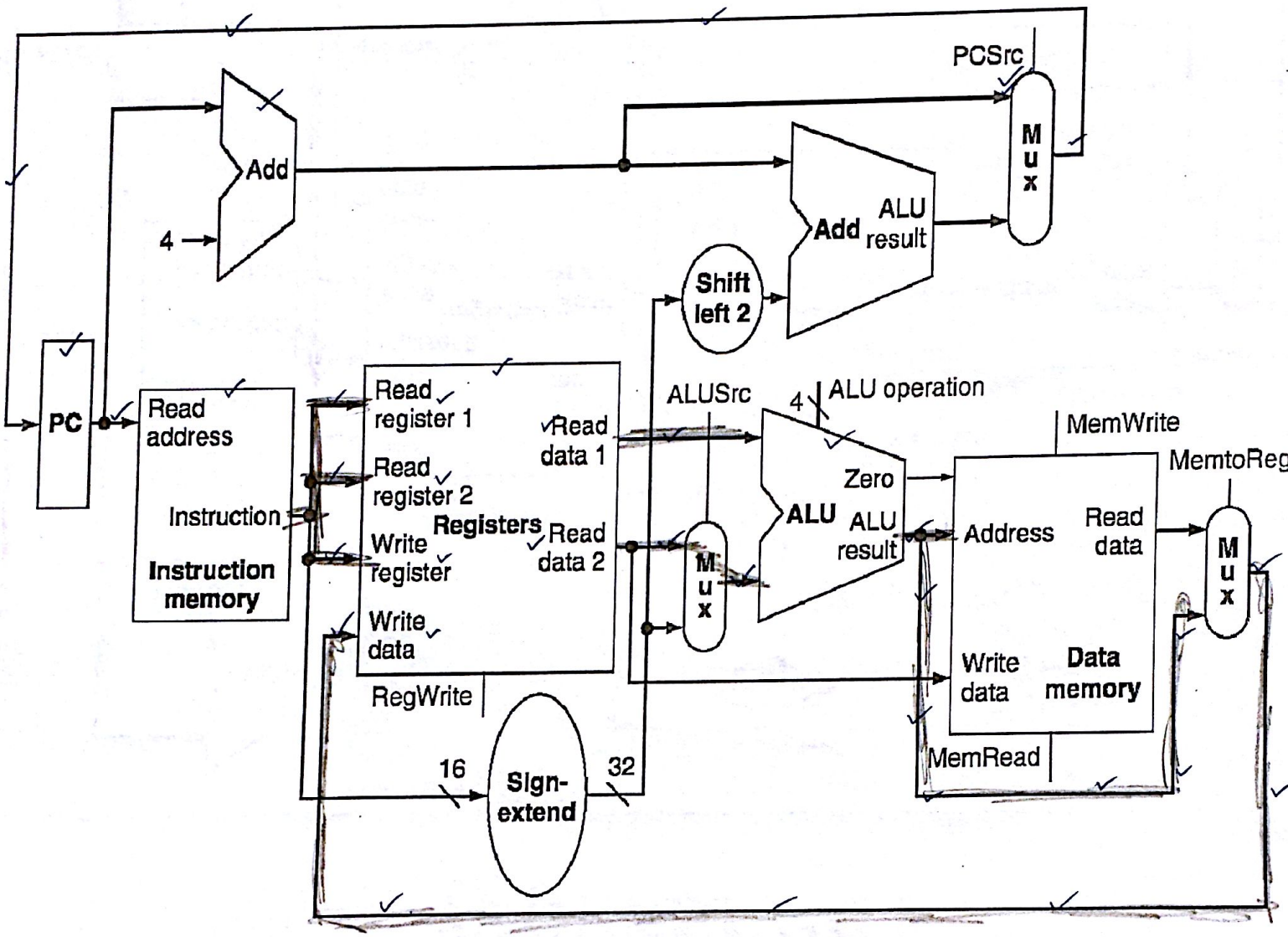
Exercise

- See worksheet (Fig 4.15)
- Highlight active datapath for
  - Add
  - Beq
  - Sw
  - Lw

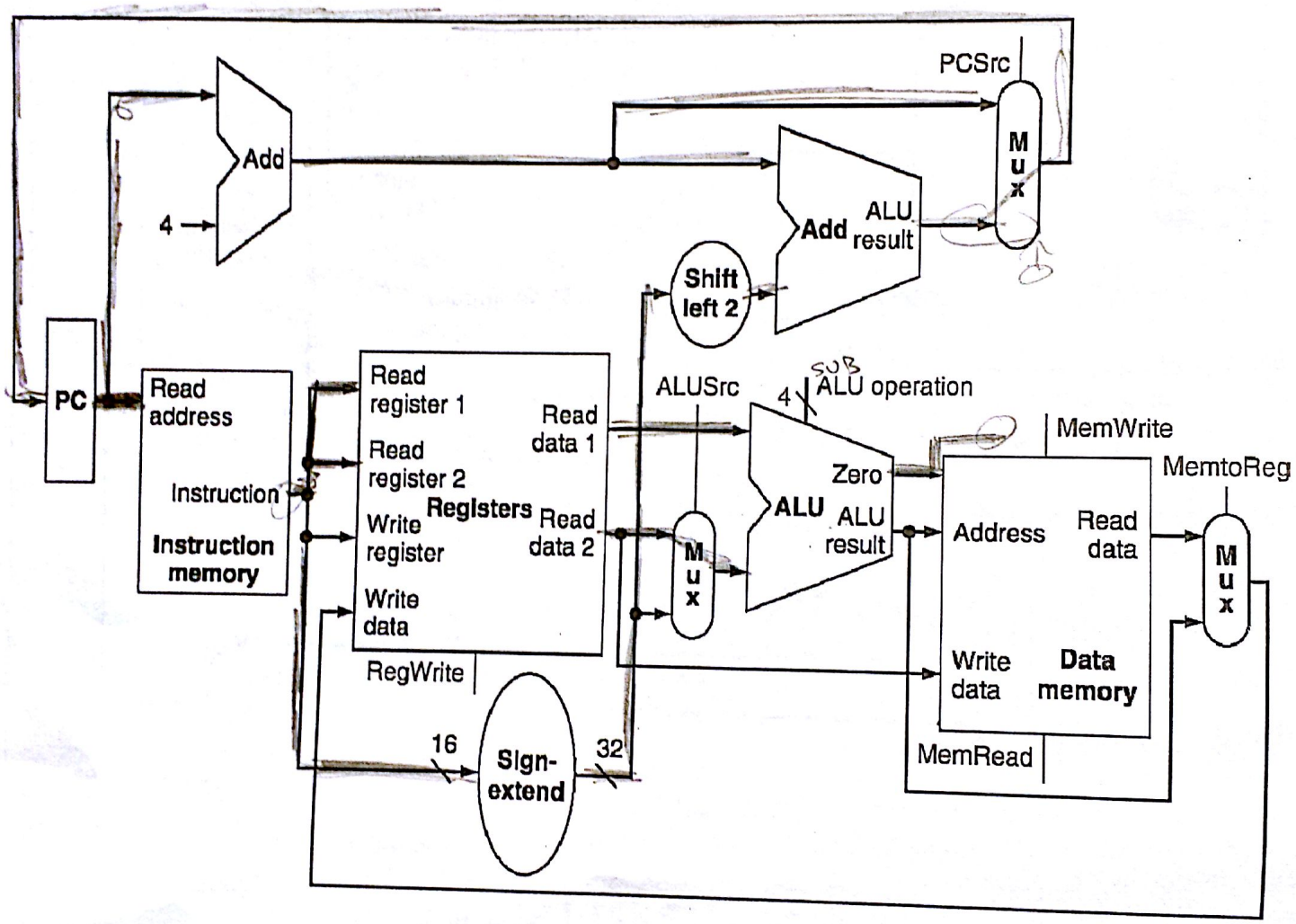
jal
jr
etc
not done in this circuit diag.
Must modify accordingly.

Add

Worksheet

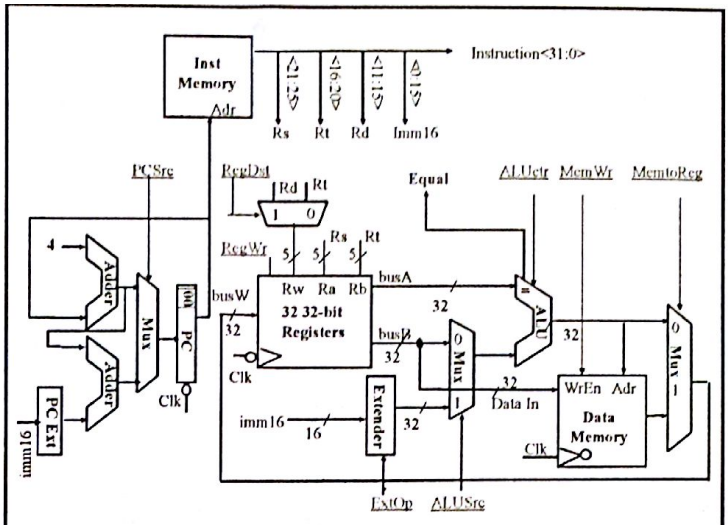


Worksheet



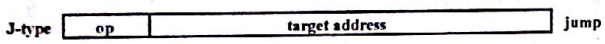
Summary

- For a given instruction
  - Describe operation in RTL
  - Use ALUs, Registers, Memory, adders to achieve reqd. functionality
- To add instructions
  - Rinse and repeat
  - Reuse components by using muxes
- Controls : later
  - Selection controls for muxes
  - ALU controls for ALU ops
  - Register address controls
  - Write enables for registers/memory



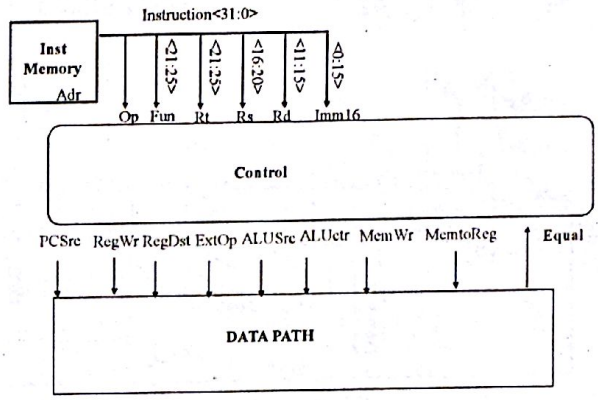
Exercise

- Add jump instruction to single cycle datapath
  - j Addr
  - RTL
    - PC <- (PC+4)[31:28] // Addr // 00

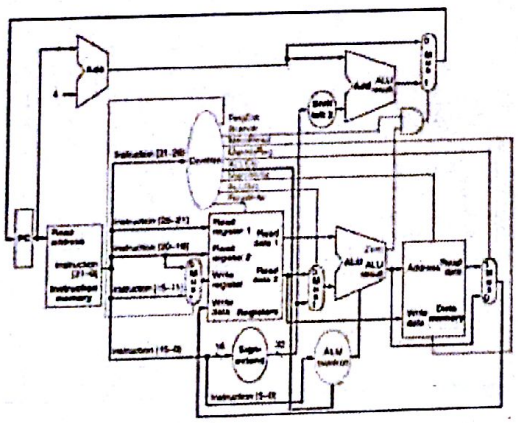


- See worksheet #1

Control for Datapath

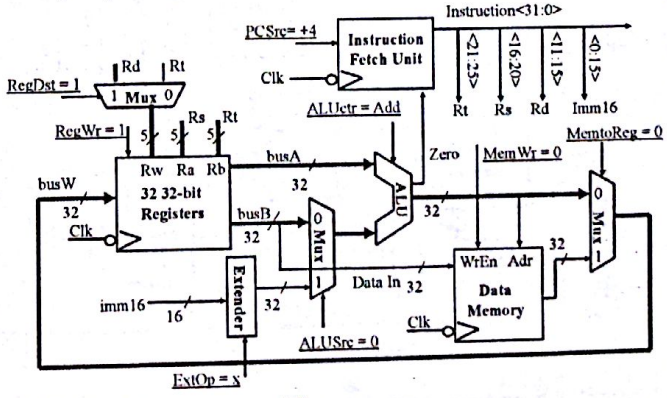


Textbook version



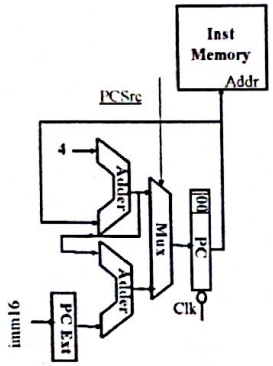
Controls for Add Operation

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



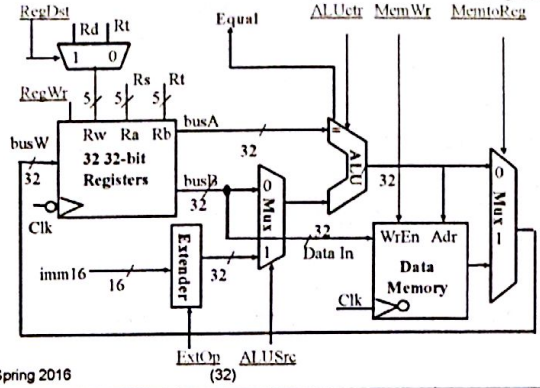
Meaning of Control Signals

- rs, rt, rd and imm16 hardwired in datapath
- PCSrc: 0 => PC <- PC + 4; 1 => PC <- PC + 4 + SignExt(Im16) || 00



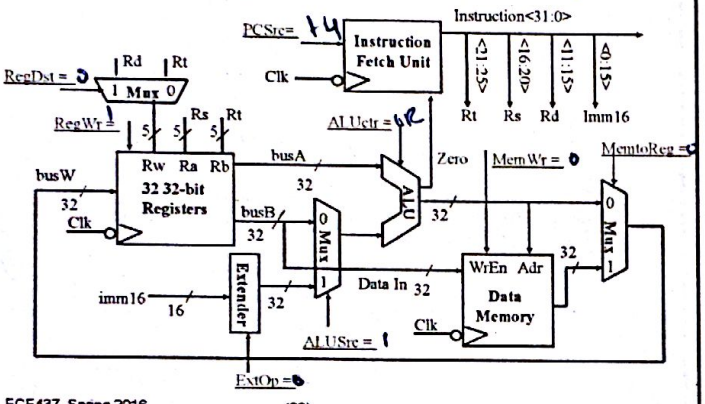
Meaning of Control Signals

- ExtOp: "zero", "sign"
- ALUsrc: 0 => regB; 1 => immed
- ALUctr: "add", "sub", "or"
- MemWr: write memory
- MemtoReg: 1 => Mem
- RegDst: 0 => "rt"; 1 => "rd"
- RegWr: write dest register



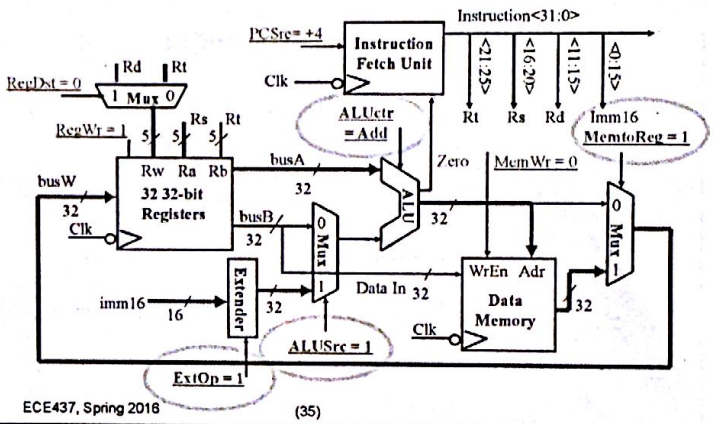
ORI Controls: Worksheet

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



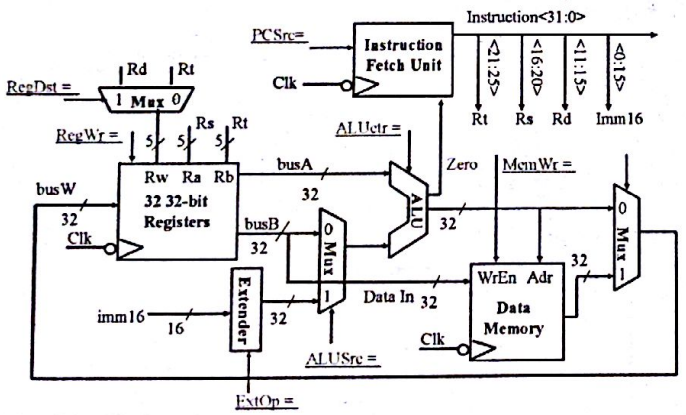
LW Controls

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



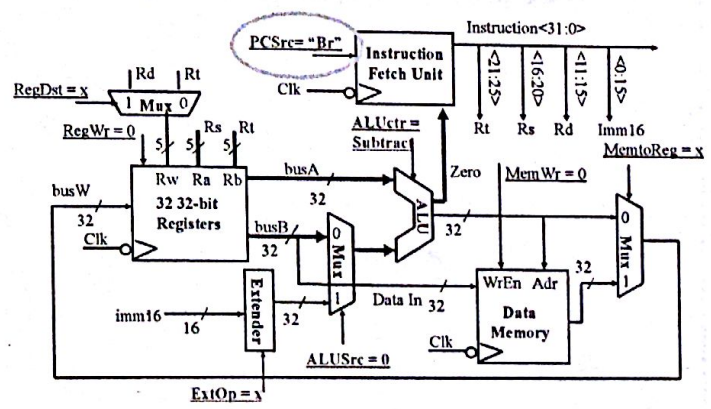
SW Controls: Worksheet

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



BEQ Controls

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



Control Logic

- Logic must generate appropriate signals for all instructions
- Summary slide (previous)
  - A way of representing the truth table
- First:
  - Equations in terms of opcodes
- Next
  - Equations in terms of instruction bits

Controls: Logic equations

- PCSrc <= if (OP == BEQ) then ZERO else 0
- ALUsrc <=if (OP == "R-type") then "regB"
  elseif (OP == BEQ) then regB, else "imm"
- ALUctr <= if (OP == "R-type") then funct
  elseif (OP == ORi) then "OR"
  elseif (OP == BEQ) then "sub"
  else "add"
- ExtOp <= _____
- MemWr <= op == store
- MemtoReg <= _____
- RegWr: <= _____
- RegDst: <= _____

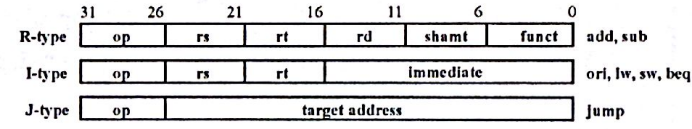
Controls: Logic equations

- PCSrc <= if (OP == BEQ) then EQUAL else 0
- ALUsrc <= if (OP == "R-type") then "regB"
  elseif (OP == BEQ) then regB, else "imm"
- ALUctr <= if (OP == "R-type") then funct
  elseif (OP == ORi) then "OR"
  elseif (OP == BEQ) then "sub"
  else "add"
- ExtOp <= if (OP == ORi) then "zero" else "sign"
- MemWr <= (OP == Store)
- MemtoReg <= (OP == Load)
- RegWr: <= if ((OP == Store) || (OP == BEQ)) then 0 else 1
- RegDst: <= if ((OP == Load) || (OP == ORi)) then 0 else 1

Truth Table summary

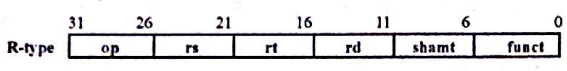
See Appendix A

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



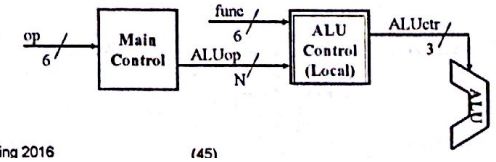
Local vs Global Control

- One more layer of abstraction
  - ALUctr <= if (OP == "R-type") then funct
    elseif (OP == ORi) then "OR"
    elseif (OP == BEQ) then "sub"
    else "add"

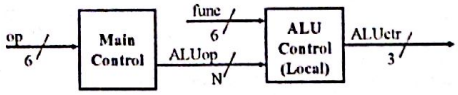


Global Control: Truth Table

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



Encoding

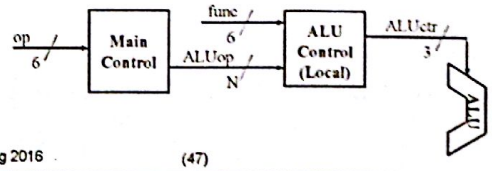


- In this exercise, ALUop has to be 2 bits wide to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, and (4) Subtract
- To implement the full MIPS ISA, ALUop has to be 3 bits to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, (4) Subtract, and (5) And (Example: andi)

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

Global Control: Truth Table

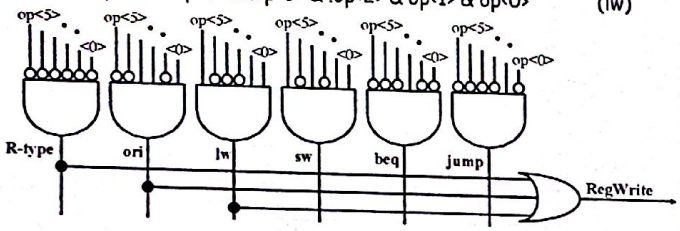
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	X	X	X
ALUSrc	0	1	1	1	0	X
MemtoReg	0	0	1	X	X	X
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	X	0	1	1	X	X
ALUop<2:0>	"R-type" (100)	Or (010)	Add (000)	Add (000)	Subtract (001)	XXX



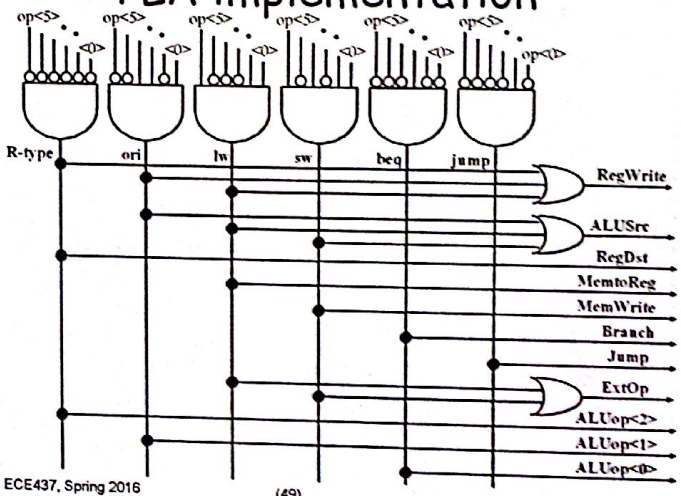
Truth Table for RegWrite

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

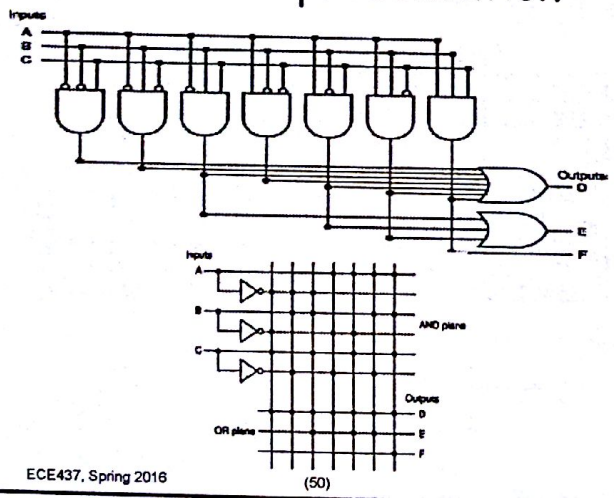
- RegWrite = R-type + ori + lw
  = !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)
  + !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)
  + op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



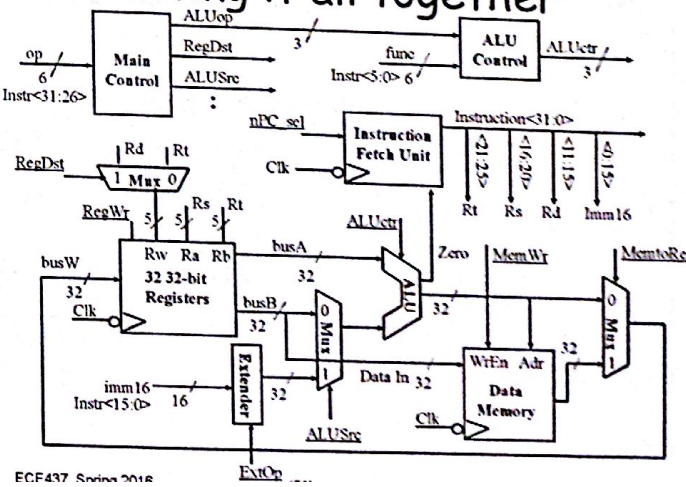
PLA implementation



PLA Representation

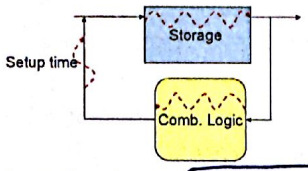


Putting it all together



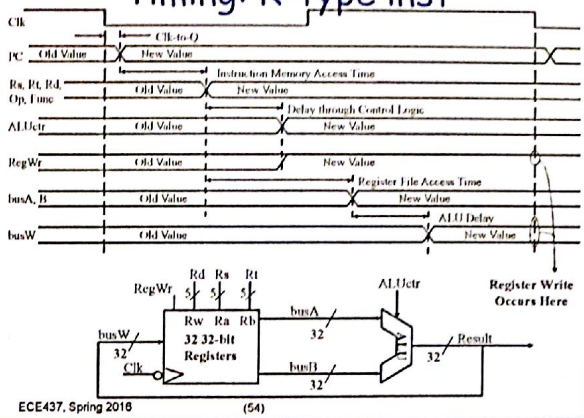
We lose something
150 to 200 × slower

Cycletime



- What should the clock period be?
  - Enough to compute the next state values
    - Propagation clk-to-Q (new state)
    - Comb. Logic delay
    - Setup requirements

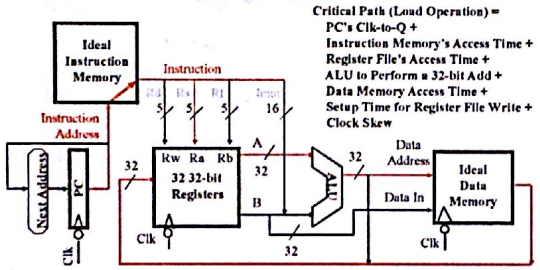
Timing: R-type inst



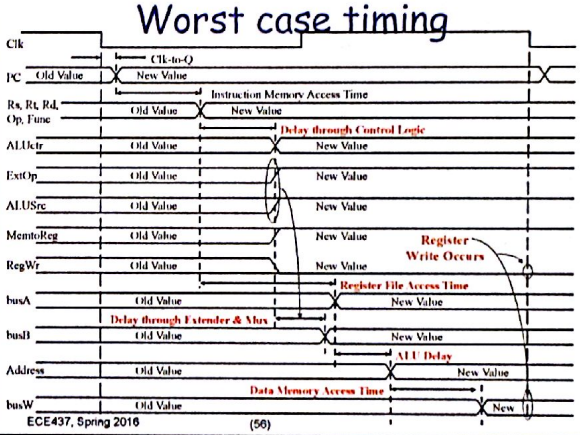
has most delay components. ⇐ determines clock.

"lw" Instruction

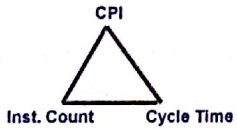
- Longer critical path
  - lower bound on cycletime



Worst case timing



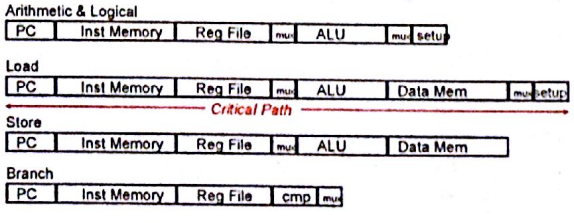
Single-cycle Datapath



- Performance Implications
  - Minimize all three
  - Insts/prog fixed -- f(interface,compiler)
  - CPI = 1 : As good as it gets (*)
  - Clock cycle time : high, "lw" critical path

throughput metric →

What's wrong with our CPI=1 processor?



- Long Cycle Time
- All instructions take as much time as the slowest
- Real memory is not so nice as our idealized memory
  - cannot always get the job done in one (short) cycle

ECE437: Introduction to Digital Computer Design

Mithuna Thottethodi

Back to Chapter 4

(Done with "Make it correct". Now: "Make it fast")

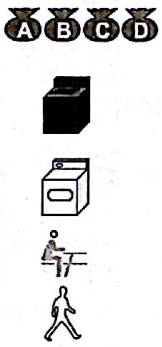
Spring 2016

Outline

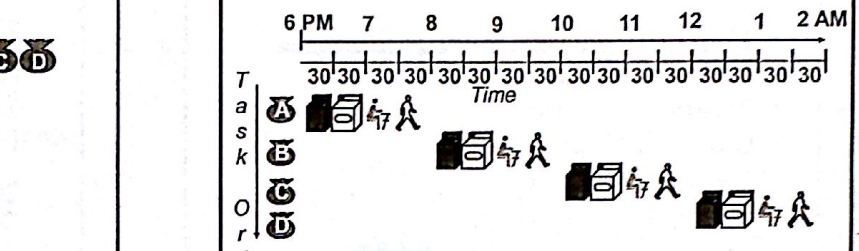
- Pipelining
  - What? Basic concepts
    - Overlapping execution
    - Latency vs. throughput
  - Why? Performance implications
    - Speedup
    - CPI, cycletime
  - How? Implementation challenges

Pipelining

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 30 minutes
- "Folder" takes 30 minutes
- "Stasher" takes 30 minutes to put clothes into drawers

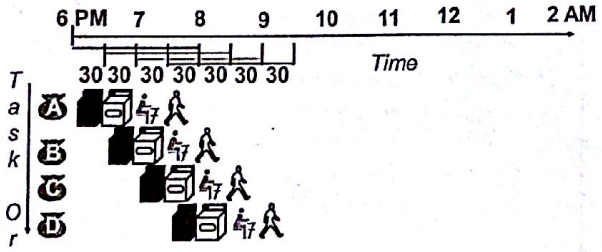


Sequential Laundry



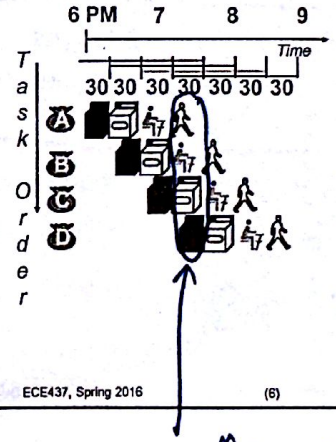
- Sequential laundry takes 8 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry



- Pipelined laundry takes 3.5 hours for 4 loads!
- Not esoteric computer architecture concept
- Natural, Intuitive

Pipelining lessons



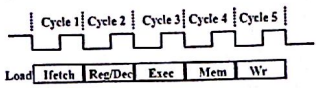
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependences

max efficiency point at $n \to \infty$

improvement factor $= \frac{2N}{2+\frac{(N-1)}{2}} = \frac{2N}{1.5+N/2} = \frac{4N}{3+N}$ loads/operations/instructions.

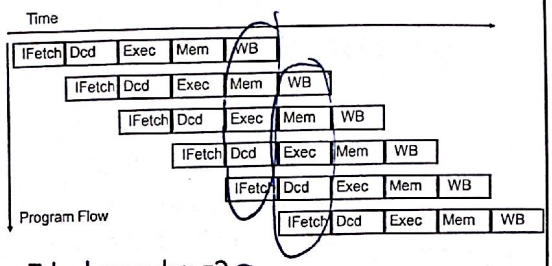
≥ 4.

The 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

Pipelined Execution Representation



- Ideal speedup =? 5

old time = CPI × n × t

new time = CPI × (n-1) × $\frac{t}{P}$ + CPI × 1 × $\frac{t}{P}$

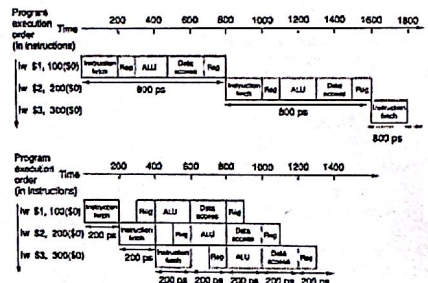
Speedup]$_{t \to 0}$ = P.

when pipeline stages are uniform

Ideal speedup

- All instructions are executed in P pipeline stages in a multicycle path (i.e. CPI = P)
- Cycletime = t ns (say)
- Instr. Count = n
- Old time = P x t x n
- New time = n x t + (P-1) x t
- Speedup = P/(1 + (P-1)/n) ≈ $\frac{\text{old time}}{\text{new time}}$
- P is some constant, n is large => Speedup = P

Non-uniform pipeline stages



Ideal speedup is number of stages in the pipeline. Do we achieve this?

can say depends on biggest ~~clock~~ block that defines clk

after some cycles, we'll have 5 instructions same time. but overlay will be 200 ns. old instr is 800 ns.

factor = 800/200 = 4.

If ideal, factor = 5 ⇒ overlay = 160 ns. [not happens].

~~Speedup~~

Non-uniform stages

Maximum Speedup ≤ Number of stages

Speedup ≤ $\frac{\text{Time for unpipelined operation}}{\text{Time for longest stage}}$

Exercise

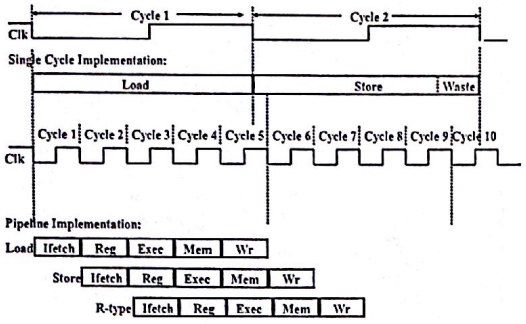
- A single cycle processor implementation can be pipelined in two ways
- Pipeline A uses a 5-stage pipeline
  - the 5 stages account for 15%, 10%, 20%, 20%, 35% of the delays respectively
- Pipeline B uses a 3-stage pipeline
  - the stages are balanced
- If instructions are all independent, which pipeline implementation is the better option

clock speed goes from 100% to 35%.

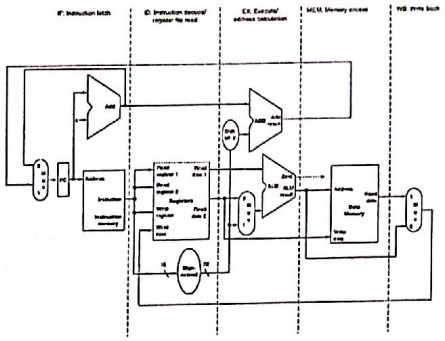
here clock speed goes from 100% to 33.3%.

⇒ B better.

Single Cycle vs. Pipeline



Hardware for pipelining?

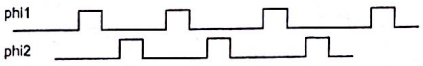


Hardware requirements

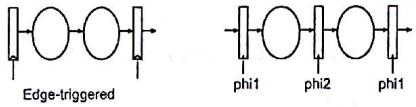
- Similar to single cycle pipeline
  - Latches** to separate stages
    - Real machines use latches and multiphase clocks
    - Latch => flipflops in this course
- Control
  - Good news
    - Minor changes from single-cycle version

• Not level sensitive.
• term used in ind.

More on Pipeline "latches"



- Remember construction of flip flop
  - Two latches, clock and inverted clock.
- 2-phase non-overlapping clocks
- Pipeline stage is two (level sensitive) latches

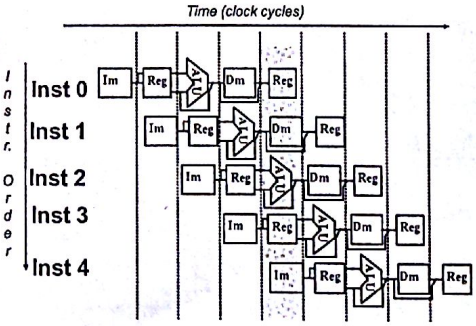


use phases to pick one of them, then other. Gets max performance. Done in industry.

Why pipeline?

- Suppose we execute 100 instructions
- Single Cycle Machine
  - 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
- Ideal pipelined machine
  - 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns

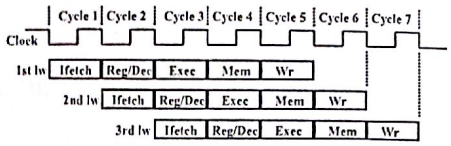
Because the resources are there!



Pipelined Processor Design

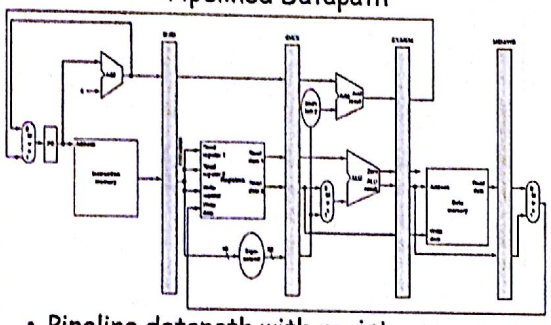
- Designing a pipelined processor
  - Associate resources with states
  - Resources not necessarily atomic
    - Register reads and writes can happen in the same cycle
    - Writes in first half of cycle and reads in the second half
  - Assert appropriate controls in each stage
    - Make sure all necessary information is carried through inter-pipestage flip-flops

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

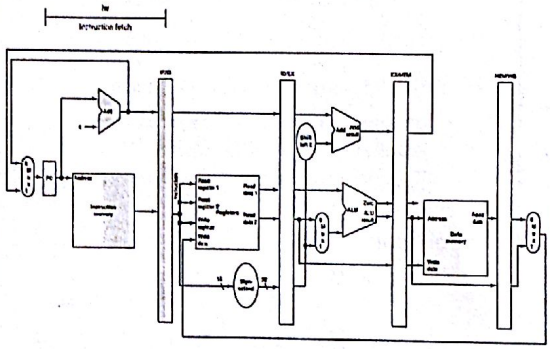
Pipelined Datapath



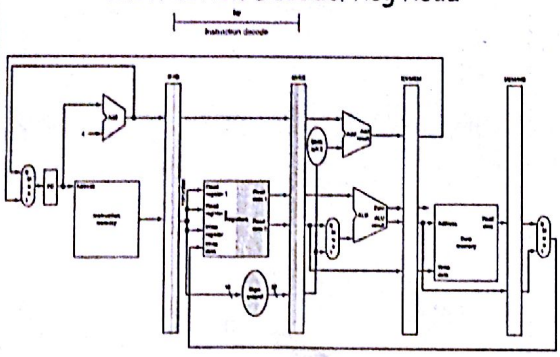
- Pipeline datapath with registers

1st half write | 2nd half read

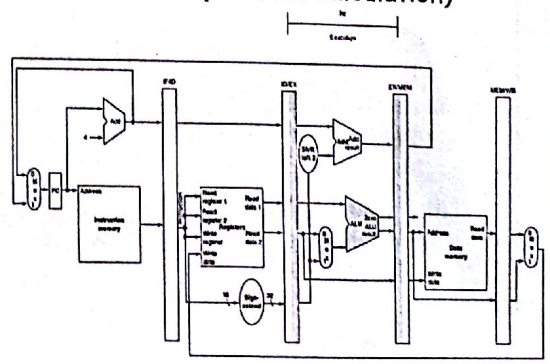
Instruction Fetch



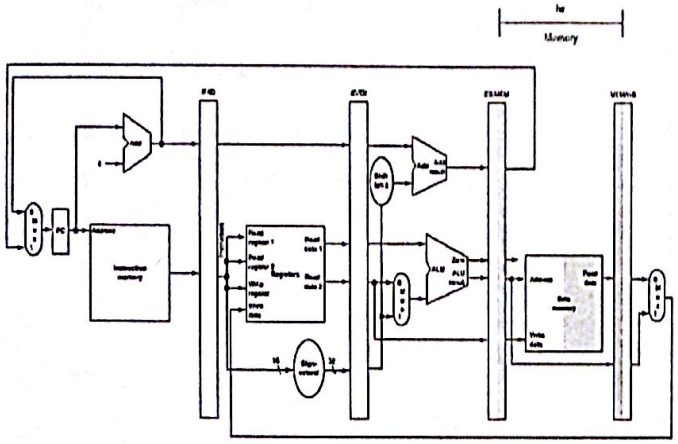
Instruction Decode/Reg Read



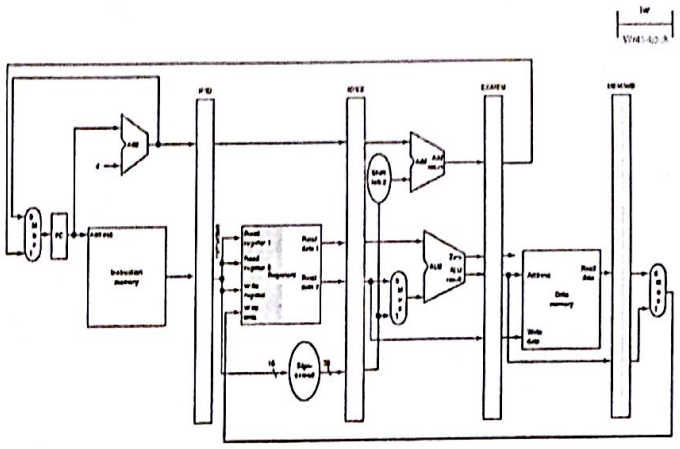
Execute (Address calculation)



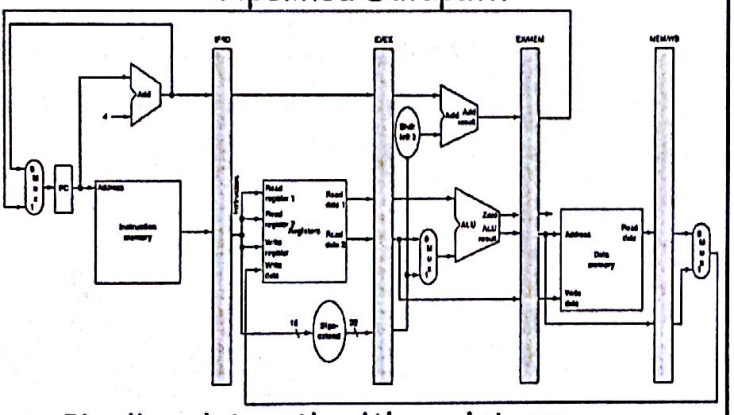
Memory Access



Writeback



Pipelined Datapath

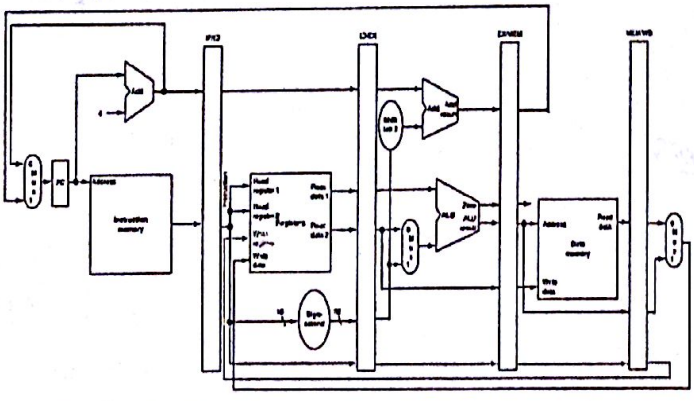


- Pipeline datapath with registers

Pipeline registers

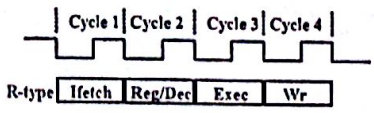
- Length of Pipeline registers
- Book says (Fig 4.35)
  - IF/ID : IR (32), PC+4 (32) : 64 bits
  - ID/EX: IR (32), PC+4 (32) + RegA + RegB : 128 bits
  - EX/MEM: ALUout(32) + zero(1) + PC+4+SX(imm) (32) : 97
  - MEM/WB: ALUout (32) + MemData(32) : 64
- To be refined:
  - (see next slide)
  - Other control bits (IR not going through)

Corrected Datapath



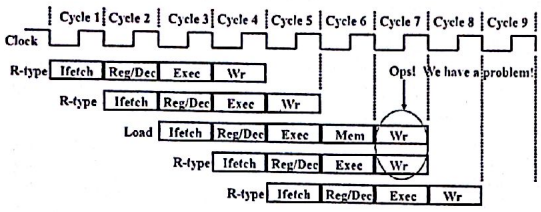
- Carry destination register through pipeline registers to WB stage

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
  - Update PC
- Wr: Write the ALU output back to the register file

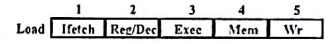
Pipelining R-type and Loads



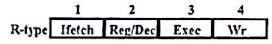
- We have pipeline conflict or structural hazard:
  - Two instructions try to write to the register file at the same time!
  - Only one write port

Key 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

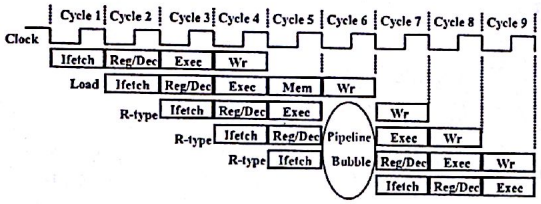


  - R-type uses Register File's Write Port during its 4th stage



  - 2 ways to solve this pipeline hazard.

Soln.1: Insert "Bubble"

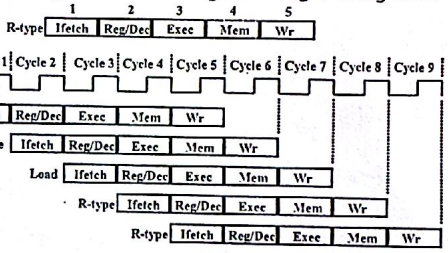


- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.
- No instruction is started in Cycle 6!

Values being clashed

Soln.2: Delay R-type's Write

- 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.

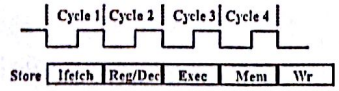


bubbles hurt "b" it kills a ~~a~~ fetch.
2 seems worse but actually doesn't affect ~~operations~~ throughput too much.

Does soln. 2 lose performance?

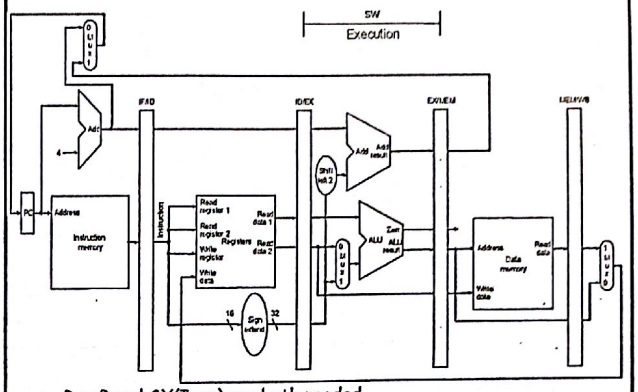
- Does it increase R-type's latency?
- Does it worsen R-type's CPI?
- What is the bottomline performance concern in pipelining?

Four Stages of Store



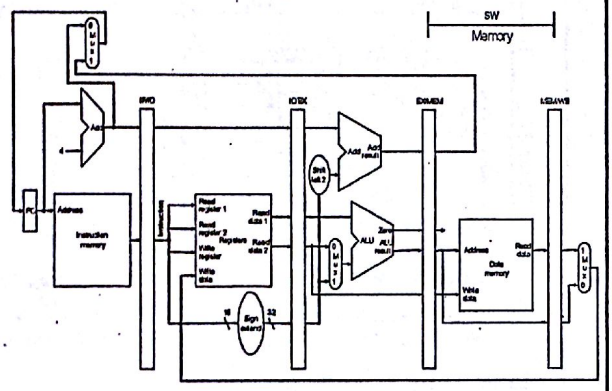
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

Exec Stage of Store

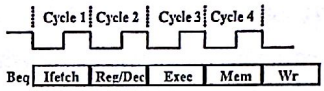


- Reg B and SX(Imm) are both needed

Mem stage of Store



Three** Stages of Beq



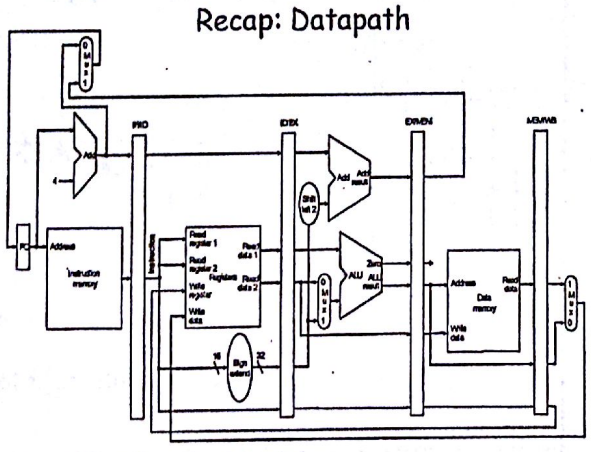
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec:
  - Registers Fetch and Instruction Decode
- Exec:
  - compares the two register operand,
  - select correct branch target address
  - latch into PC
- Four stages as in book
  - Assume one delay slot (shadow instruction)
  - One "predict not taken" instruction

assume one delay happens & let compiler fill it with indep. operation

methods to solve dependencies

guess what happens eg BEQ

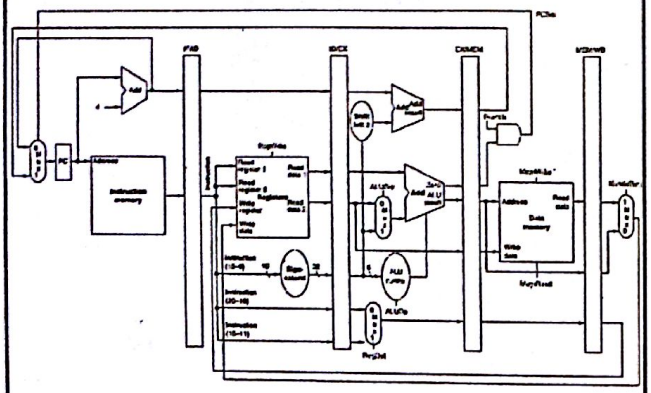
Recap: Datapath



Outline

- Pipeline Datapath
- Control
  - Relatively simple for pipelined processors
    - Similar to single cycle control

Pipelined Datapath with Controls



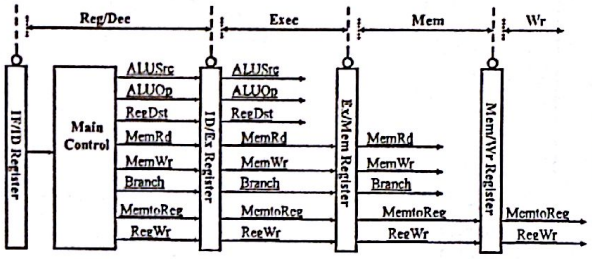
Pipeline Control vs. Single cycle control

- Similar (generation)
- What about
  - PCWrite? IR-write?
  - Write enable for the pipeline registers?
- Pipelined processor implementation
  - Common Stages
    - Instruction fetch stage (IF)
    - Decode and Register read (ID)
  - Instruction specific stages
    - Execute (Exec)
    - Memory access (Mem)
    - Write back (WB)

Generating controls

- Simplify the problem
  - Reduced pipeline control to single-cycle control (Almost)
  - Generate controls once
  - Consume (i.e., use and discard) signals as you proceed along the pipeline stages
- Identify Stage of consumption for all control signals

Focus on Control



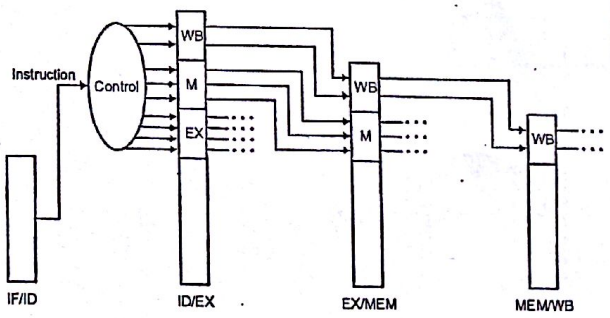
- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later

Meaning of controls

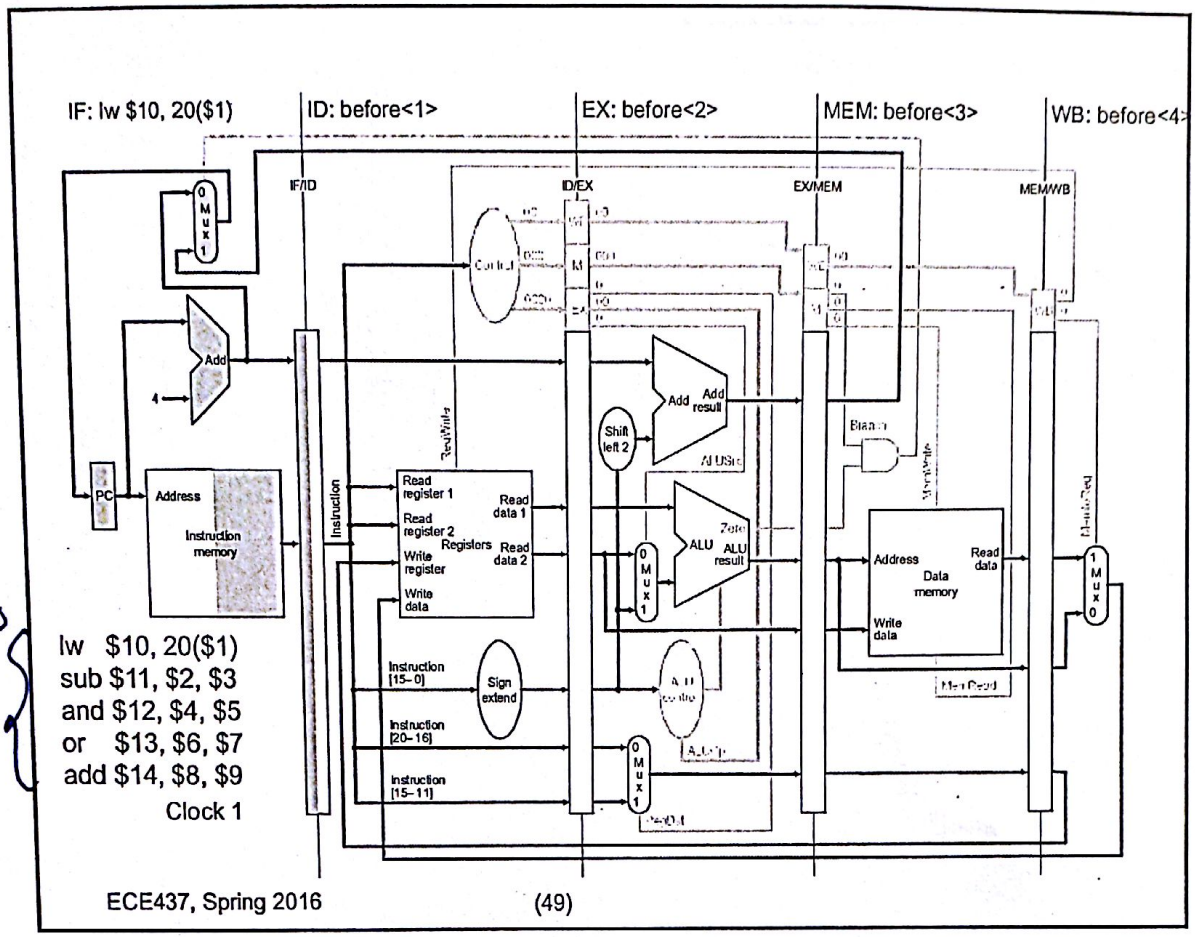
- RegWr : 1-> write, 0-> no write
- MemToReg : 1-> MDR, 0-> ALUOut
- RegDst : 1->rd, 0->rt
- ALUOp<1:0> : 00->Add,01->Sub,10->'funct'
- ALUSrc : 0->RegB, 1->SX(Imm)
- Branch: 0->non-branch inst, 1-> branch
- MemRead: 1->memread, 0-> no memread
- MemWrite: 1->memwrite, 0->no memwrite
- ALU control abstracted away (as before)
  - Inputs: ALUop (2 bits), 6 "funct" bits from IR

WB <1:0>
Exec <3:0>
Mem <2:0>

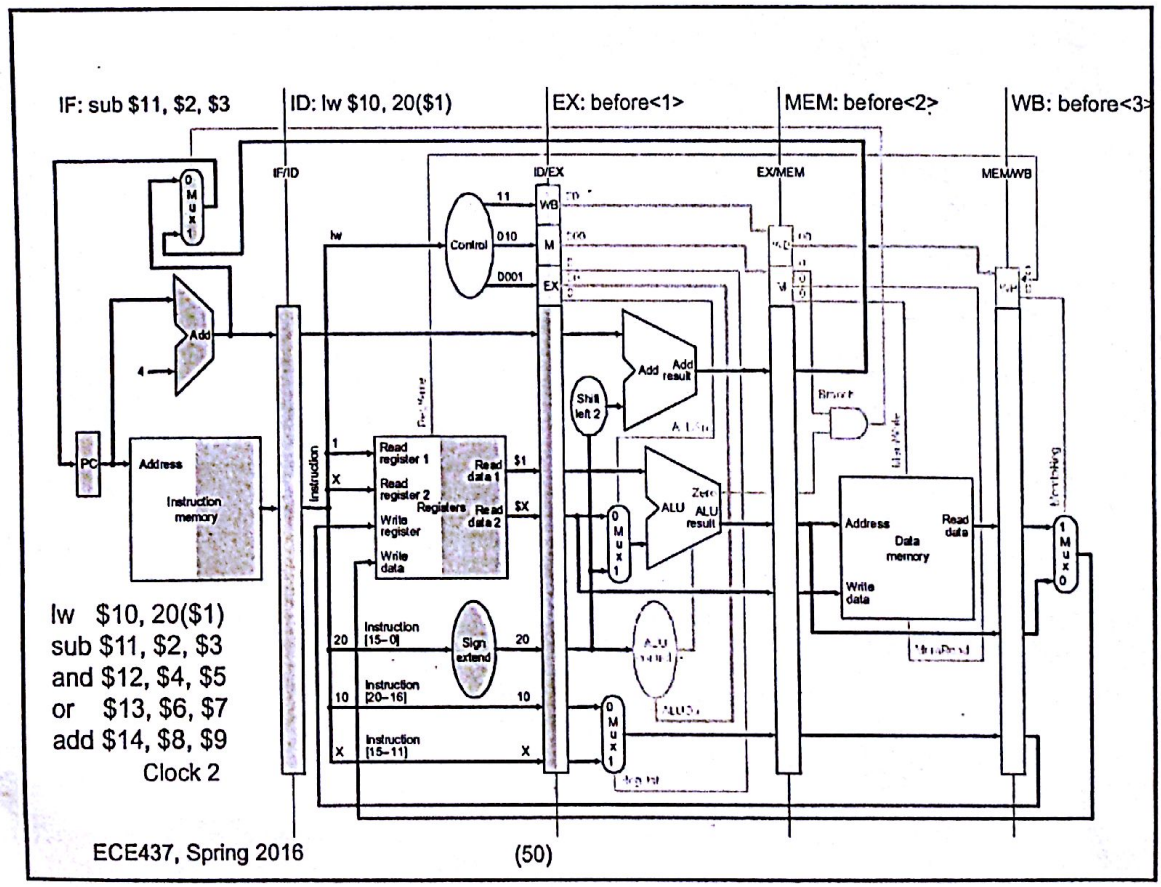
Extended Pipeline Registers

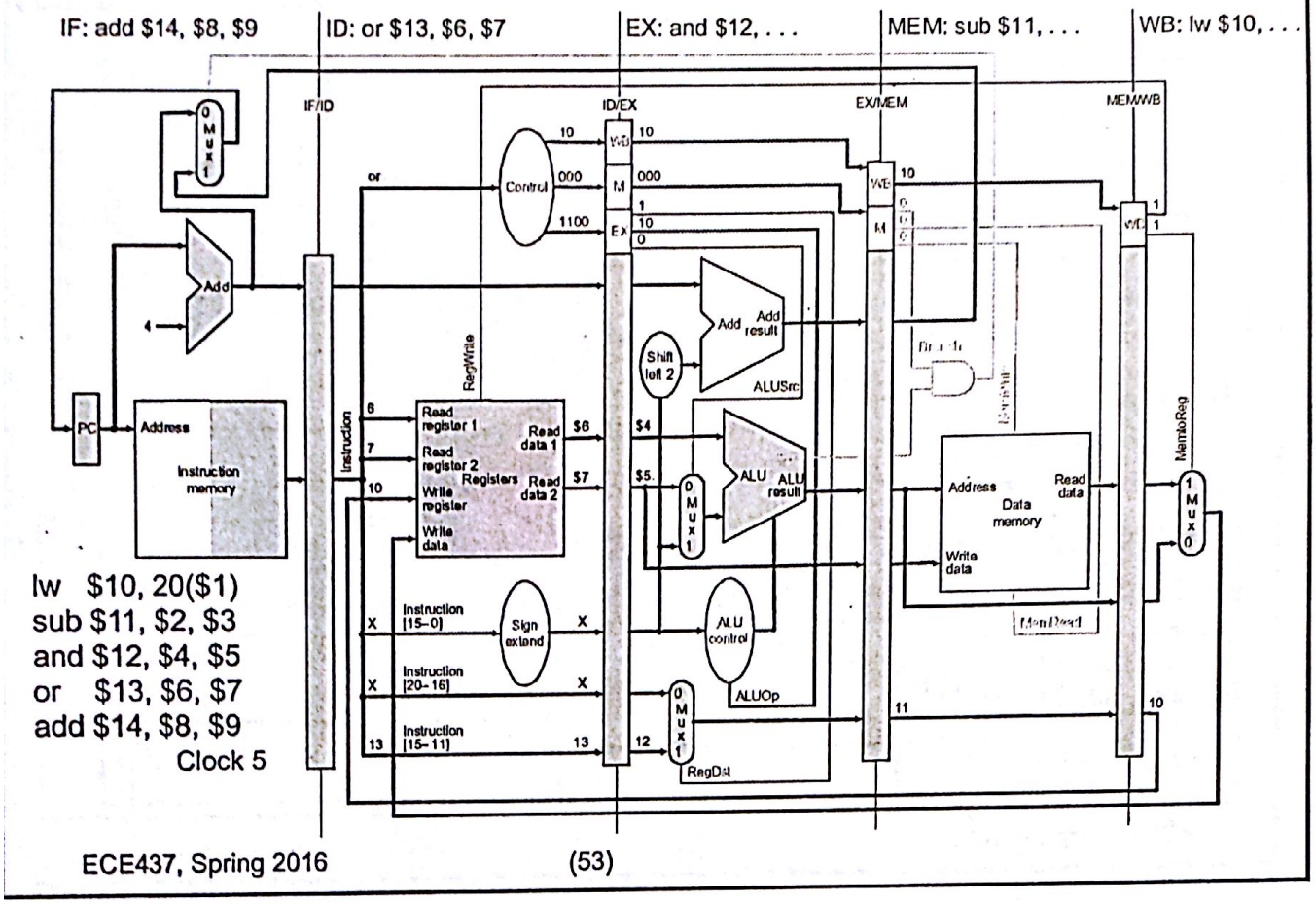


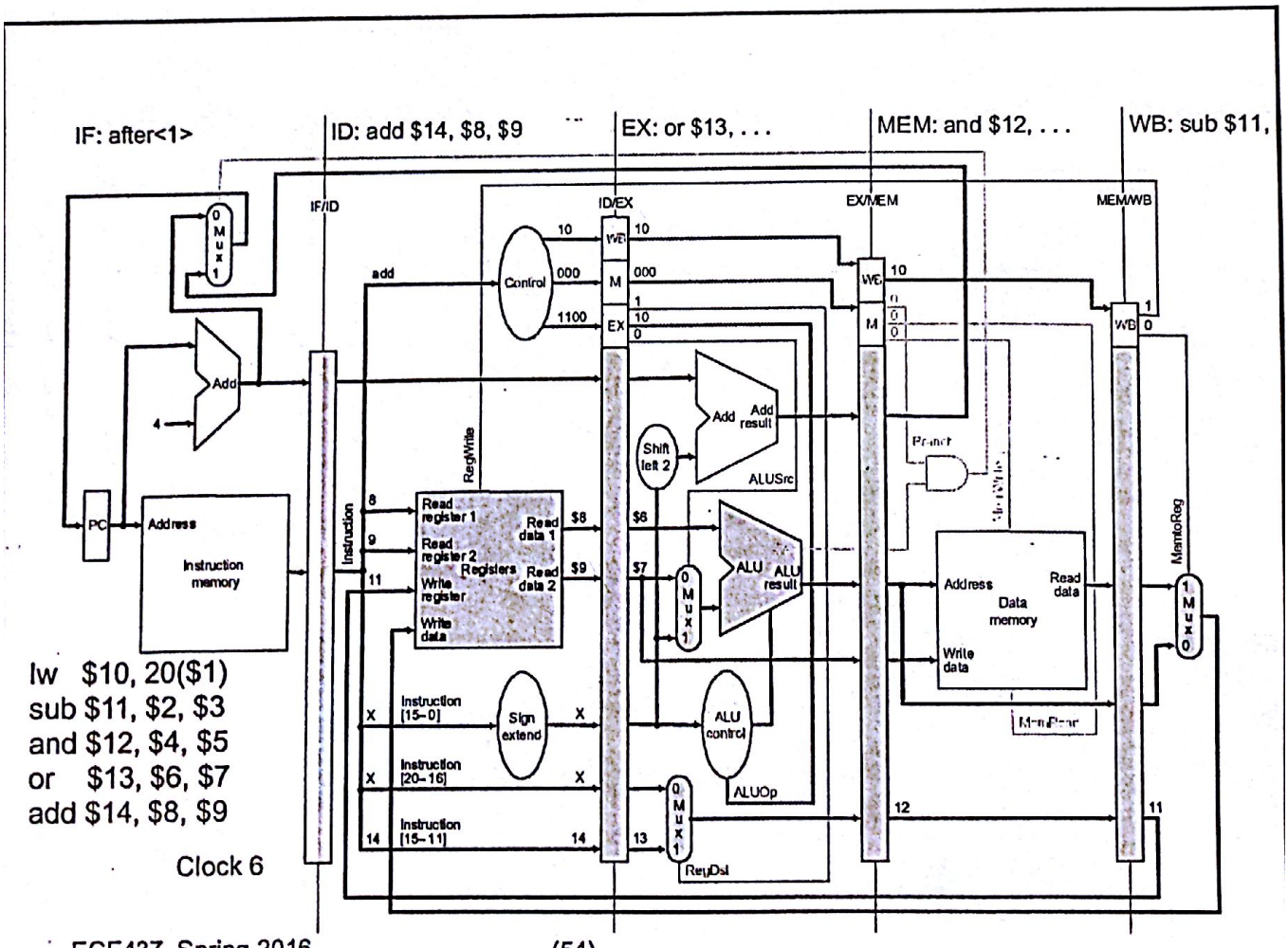
- Simple extensions to carry control state

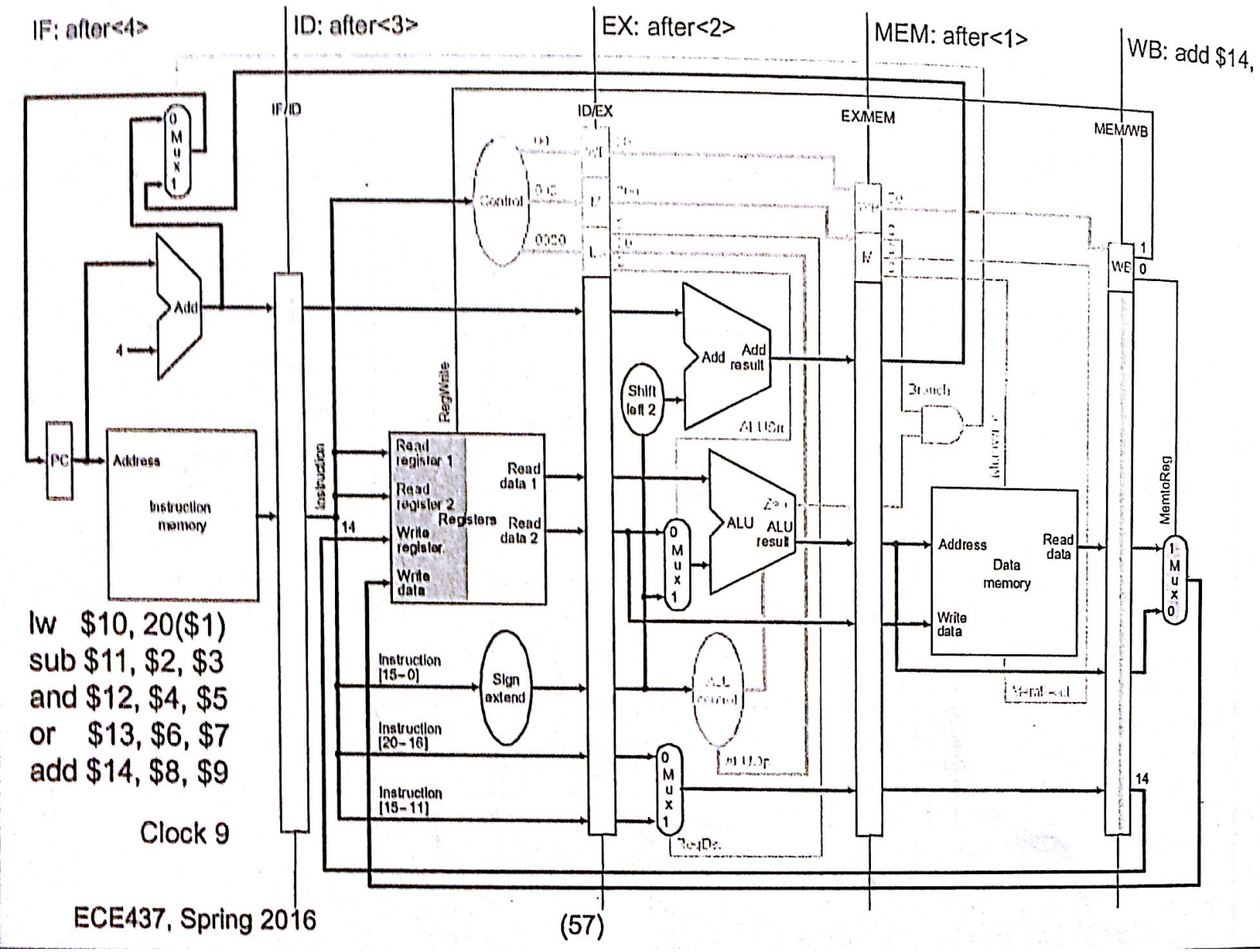


no dependency code!!!









Implementing Pipeline control

- How do we design the control logic block?
  - Similar to single-cycle implementation
  - Derive logic expressions
    - E.g. MemtoReg = lw
    - ALUSrc = lw OR sw
    - RegWrite = R-type OR lw
  - Implement Combinational logic
    - PLA implementation
    - ROM implementation

Pipeline registers

- Preliminary estimates
  - IF/ID : IR (32), PC+4 (32) : 64 bits
  - ID/EX: IR (32), PC+4 (32) + RegA + RegB : 128 bits
  - EX/MEM: ALUout(32) + zero(1) + PC+4+SX(imm) (32) : 97
  - MEM/WB: ALUout (32) + MemData(32) : 64
- Corrections:
  - ALUout and MemData
  - Destination register (5 bits)
  - Other control bits (IR not going through)

Do NOT have control unit in each state.

Implementing Pipeline Control

- Exercise
  - Compute required bit-width of pipeline registers
  - Before the next lecture

Summary

- Only slightly more complicated than single cycle
  - not really, only because we:
    - ignored complications of forwarding, branch prediction, pipeline bubbles, squashing mispredicted instructions (after branches)
  - Need deeper understanding of hazards
    - need to modify datapath as well

Today

- Pipeline datapath and control assuming independent instructions (no hazards)
- Data hazards
  - Types
  - Detecting RAW hazards
  - Handling RAW hazards (Partial)
    - Datapath
    - Control behavior

Any complications

- Definitely:
  - Need to maintain "illusion" of sequential execution
  - Execution is actually overlapped.
- Pipeline Hazards
  - structural hazards: attempt to use the same resource two different ways at the same time
    - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
  - data hazards: attempt to use item before it is ready
    - E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
    - instruction depends on result of prior instruction still in the pipeline
  - control hazards: attempt to make a decision before condition is evaulated
    - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
    - branch instructions

Runaway Analogy

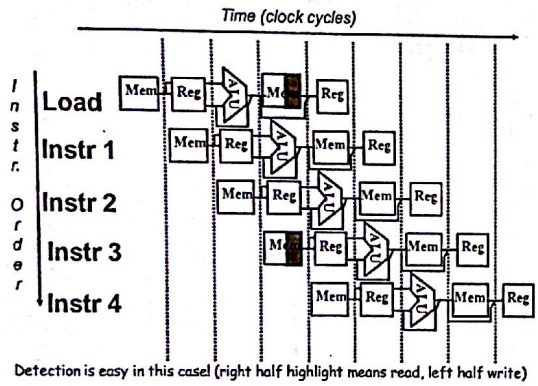
Hazards

- Structural hazards
  - Two instructions need the same hardware
  - should not overlap.
- Data Hazards
  - Data not ready
- Control Hazards
  - Which instruction to fetch? Not known.
  - eg. BRA

Hazards

- Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards
- Delays
  - Pipeline stalls/bubbles
  - Reduce speedup

Single Memory: Structural Hazard



Detection is easy in this case! (right half highlight means read, left half write)

Structural Hazards

- Single memory (suppose)
- If 1.3 memory accesses per instruction
  - How?
  - 1 per instruction for instruction fetch
  - Fraction for data load/store
    - Depends on instruction mix
    - 20% load + 10% store
    - 15% load + 15% store
- CPI is at least 1.3 (otherwise memory is used more than 100%)

1 to get inst. 0.3 (30%) for load/store

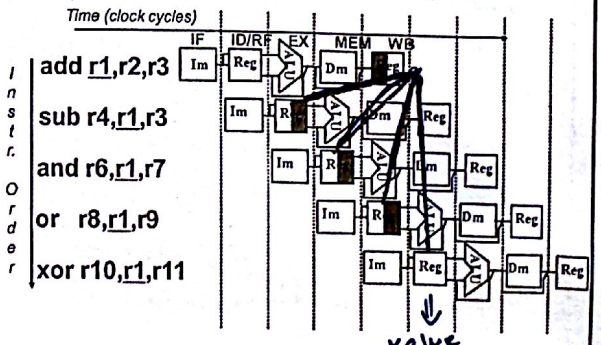
30% memory access on avg.

Data Hazards

```
add r1 ,r2,r3
sub r4, r1 ,r3
and r6, r1 ,r7
or  r8, r1 ,r9
xor r10, r1 ,r11
```


Hazards on r1

- Dependencies backwards in time

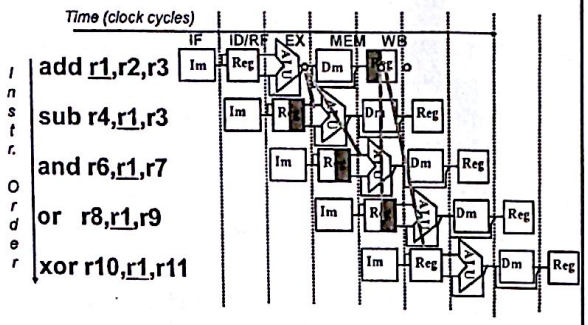


value ready here only

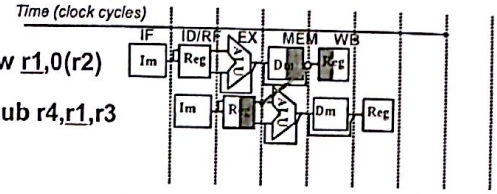
If we read the reg & write -ve, we can access one block early

where data created at first.

Data Hazard Solution



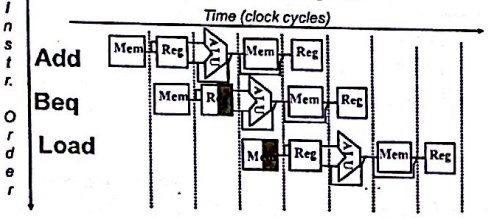
Forwarding (a.k.a. bypassing)



- Can't solve with forwarding:
  - Must delay/stall instruction dependent on loads

Control Hazard: Solutions

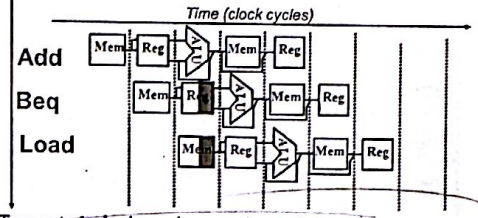
- Stall: wait until decision is clear
  - Its possible to move up decision to 2nd stage by adding hardware to check registers as being read



- Impact: 2 clock cycles per branch instruction => slow

Control Hazard: Solutions

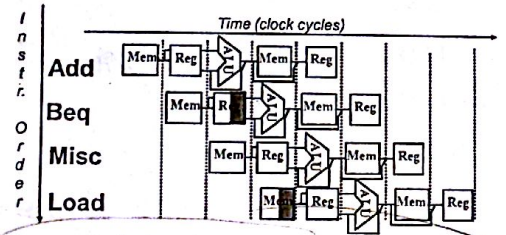
- Predict: guess one direction then back up if wrong
  - Predict not taken



- Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right 50% of time say)
- More dynamic scheme: history of 1 branch (90%)

Control Hazard: Solutions

- Redefine branch behavior (takes place after next instruction) "delayed branch"



- Impact: 0 clock cycles per branch instruction if can find instruction to put in "slot" (50% of time)
- As launch more instruction per clock cycle, less useful

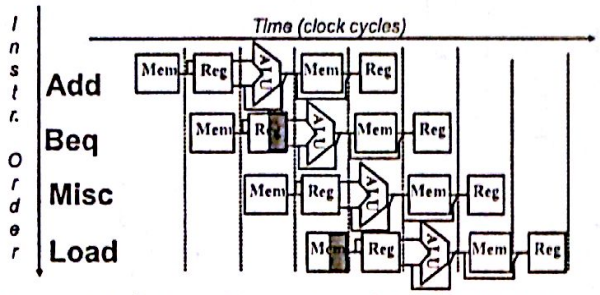
Summary: Hazards

- Structural hazards
  - Two instructions need the same hardware
  - Delay (pipeline bubble)
- Data Hazards
  - Data not ready
  - Forward/bypass (not for loads)
- Control Hazards
  - Which instruction to fetch? Not known.
  - Delayed branch, Predict not taken

delayed branch (shadow instr)
=> let compiler deal with empty delay.

Control Hazard: Solutions

- Redefine branch behavior (takes place after next instruction) "delayed branch"



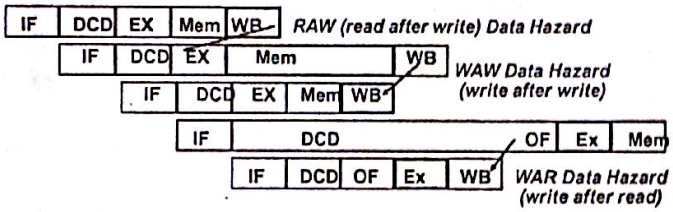
- Impact: 0 clock cycles per branch instruction if can find instruction to put in "slot" (50% of time)
- As launch more instruction per clock cycle, less useful

Summary: Hazards

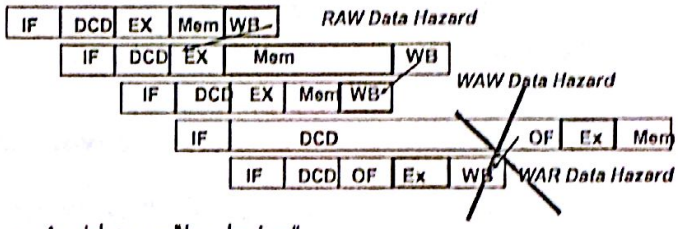
- Structural hazards
  - Two instructions need the same hardware
  - Delay (pipeline bubble)
- Data Hazards
  - Data not ready
  - Forward/bypass (not for loads)
- Control Hazards
  - Which instruction to fetch? Not known.
  - Delayed branch, Predict not taken

Data Hazards

- Challenge: maintain illusion of sequential execution
- Types of data hazards
  - RAW, WAR, WAW



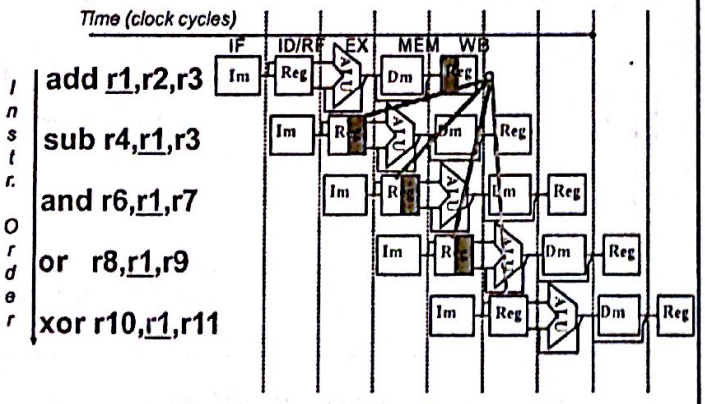
Data Hazards



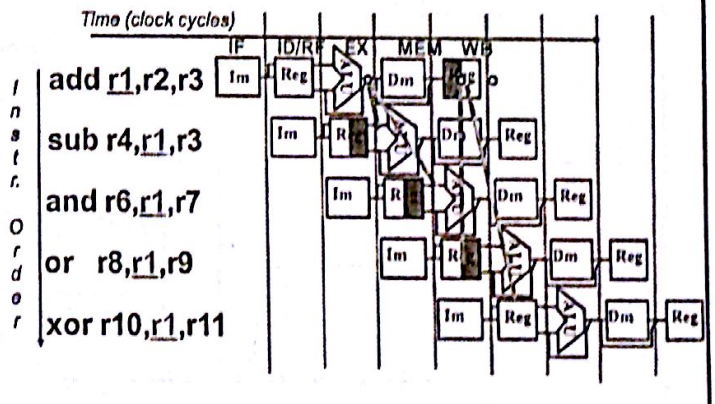
- Avoid some "by design"
  - eliminate WAR by always fetching operands early (DCD) in pipe
  - eliminate WAW by doing all WBs in order (last stage, static)
- Detect and resolve remaining ones
  - stall or forward (if possible)

Hazards on r1

- Dependencies backwards in time



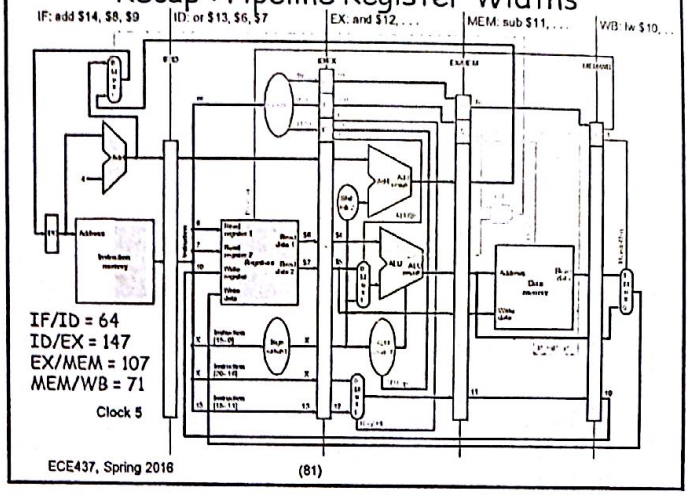
Data Hazard Solution



Handling RAW Hazards

- Pre-requisite for handling RAW hazard
  - Detection!
  - Need to know:
    - Pending writes
      - available results that haven't been written back to registers
    - Operand Reads
      - Later instructions that potentially use these values
  - Instructions may not write to register file (store, branch)

Recap : Pipeline Register Widths



Logic equations for Hazard Detection

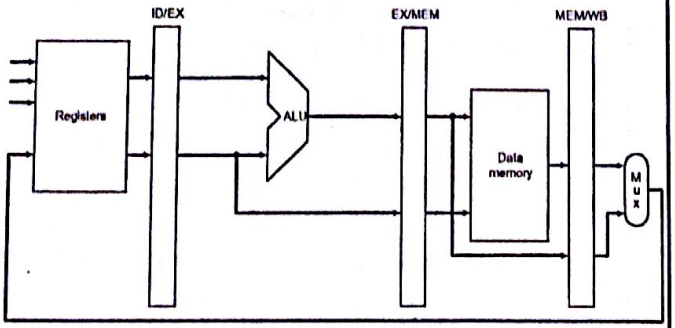
- Restatement of equations
- Text book version
  - WB stage is not really a hazard
    - Data is written in first half of cycle, read in 2nd half
  - EX/MEM.RegisterRd = ID/EX.RegisterRs
  - EX/MEM.RegisterRd = ID/EX.RegisterRt
  - MEM/WB.RegisterRd = ID/EX.RegisterRs
  - MEM/WB.RegisterRd = ID/EX.RegisterRt

Lookahead: Forwarding datapath

- We know how to detect RAW hazards
- Now,
  - Modify Datapath to enable forwarding
  - Desired control behavior

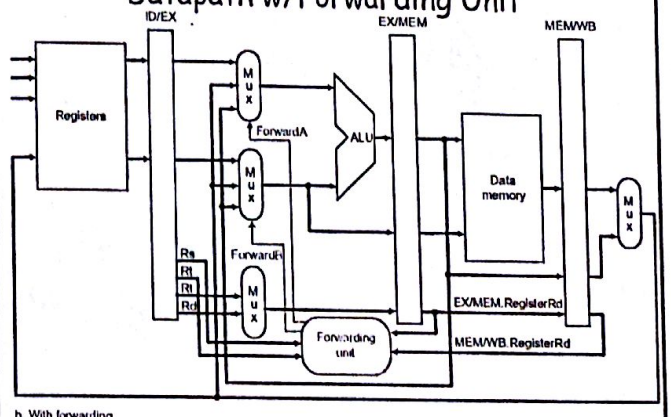
Base Pipelined Datapath

- Simplified representation of pipelined datapath
  - To avoid clutter



a. No forwarding

Datapath w/Forwarding Unit



b. With forwarding

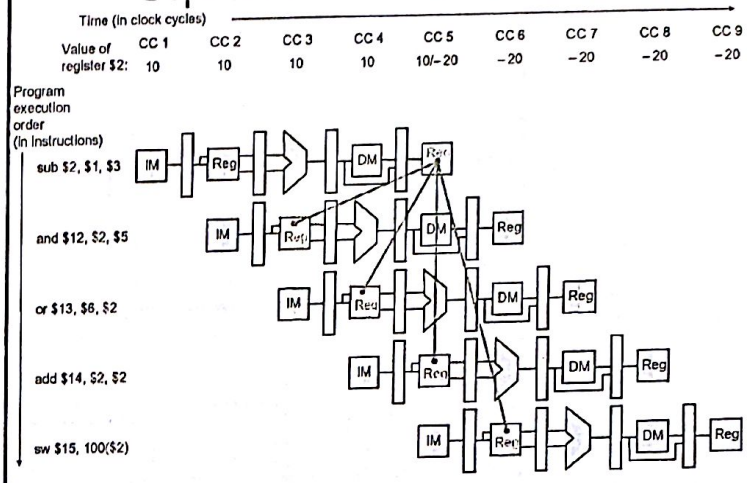
- ForwardA/ForwardB: 01->Mem, 10->EX

Data Hazards and Forwarding: Walkthrough

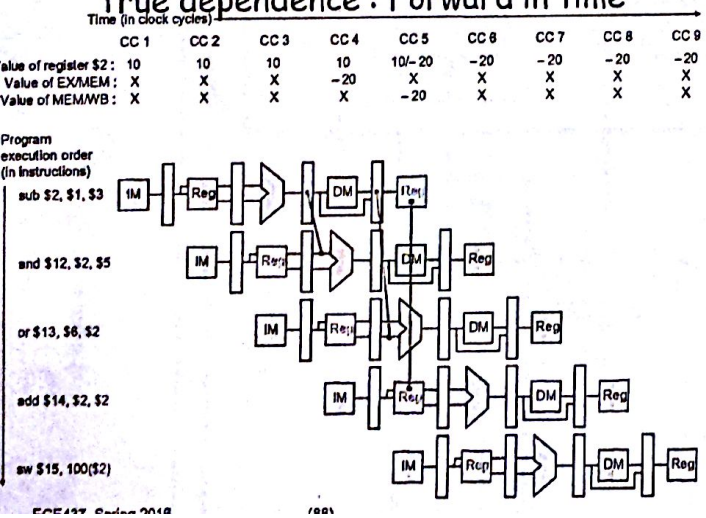
- Code snippet
  - identify hazards
  - identify forwarding paths

```
sub $2, $1, $3
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```

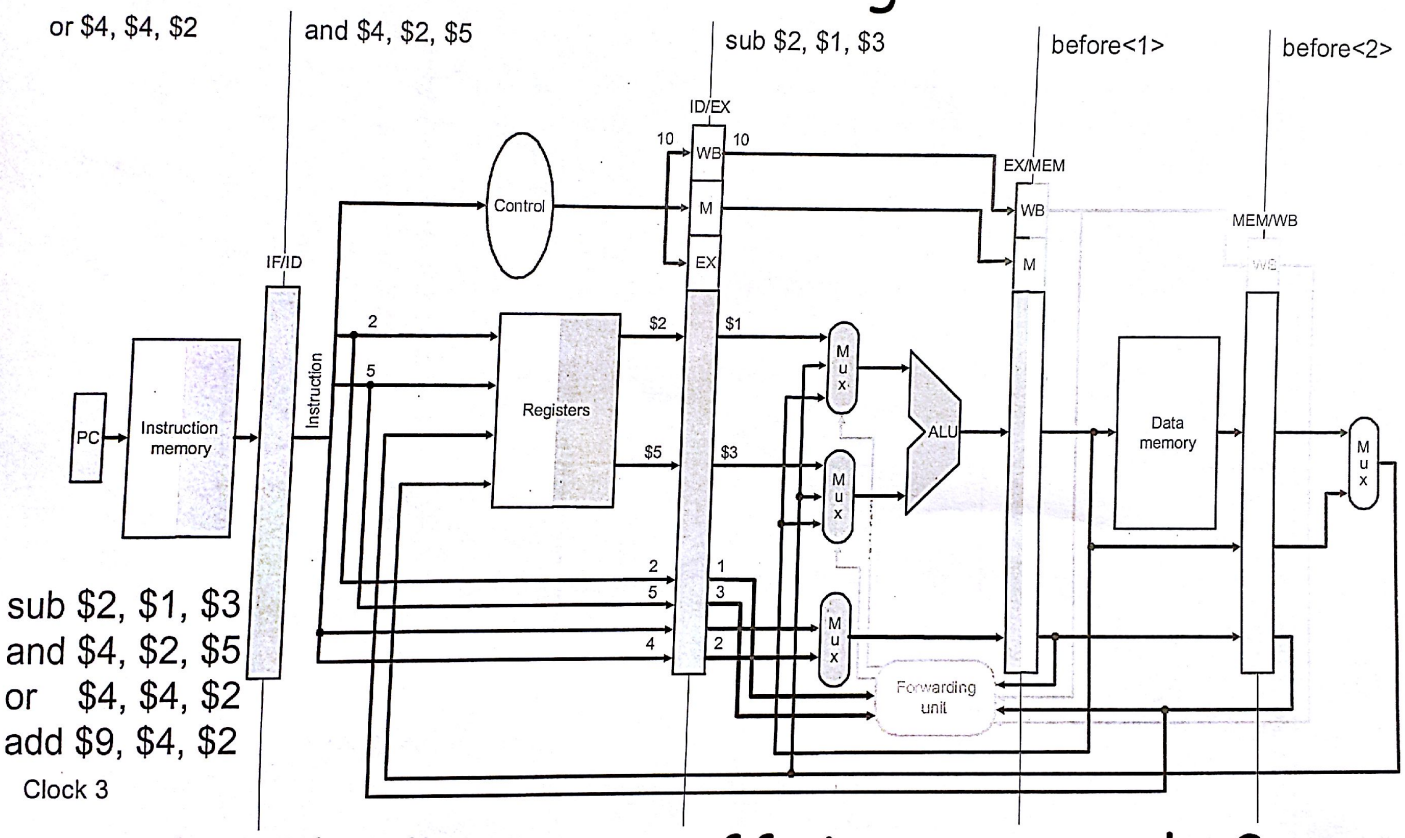

Dependence : Backward in time



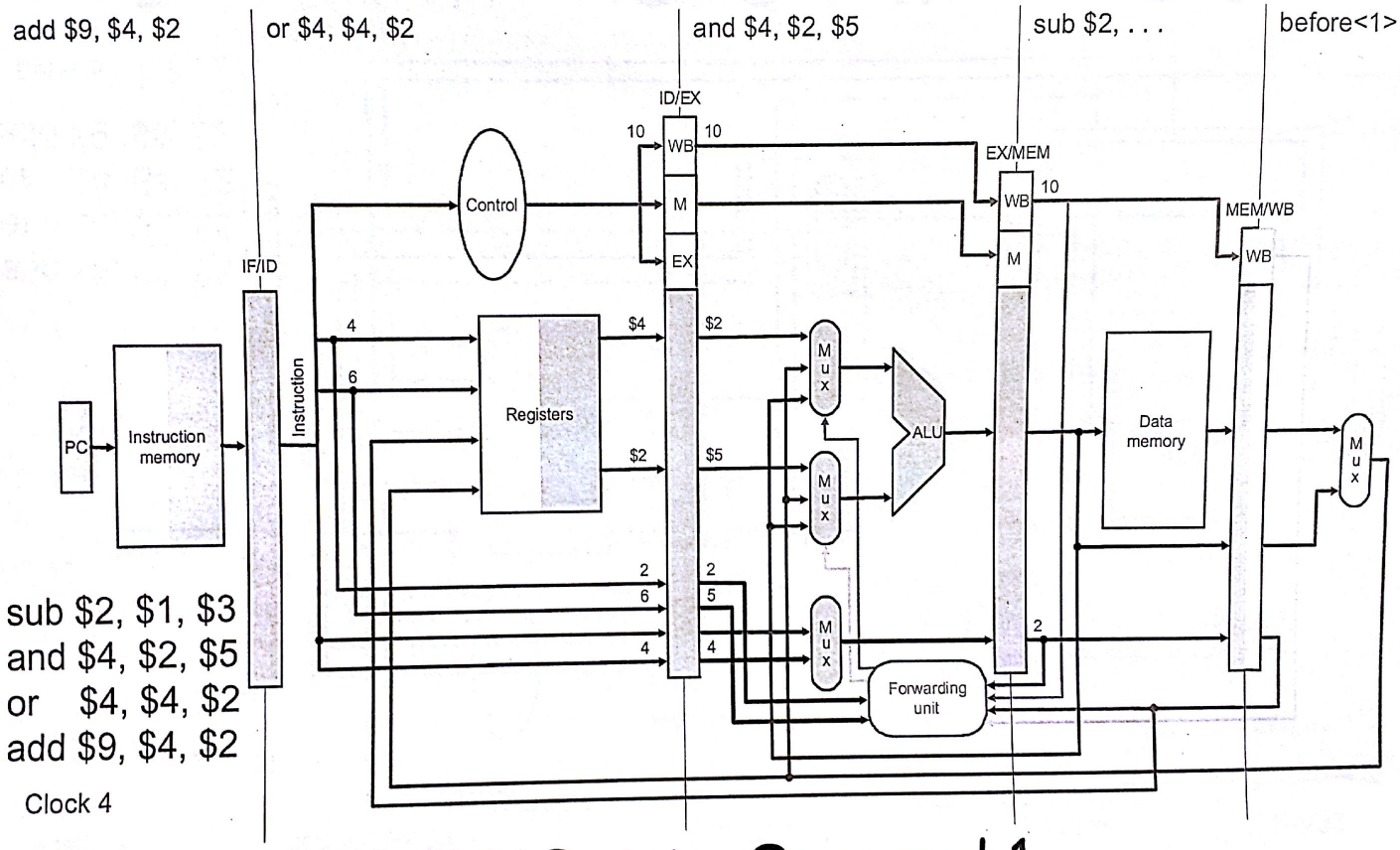
True dependence : Forward in time



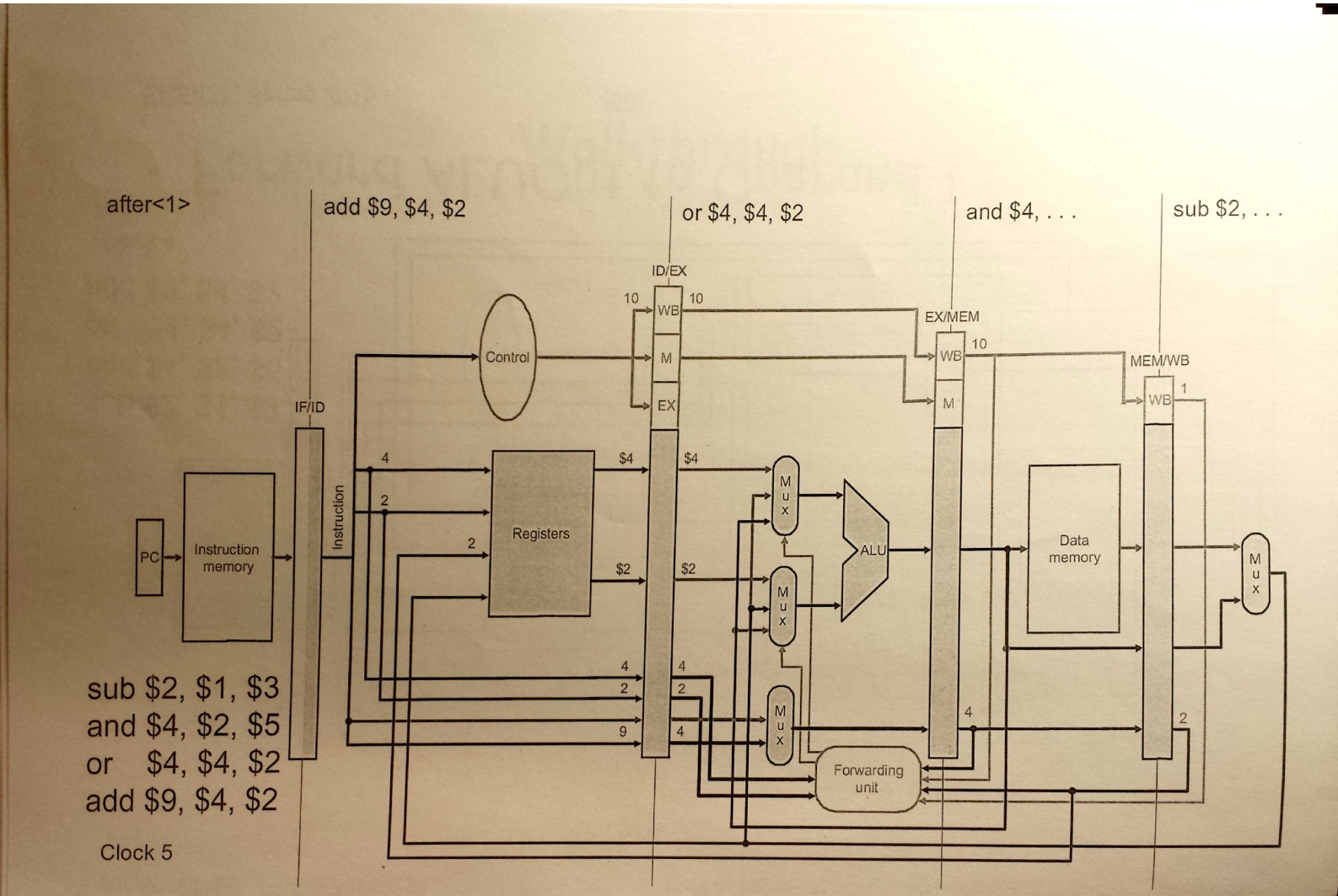
Walkthrough



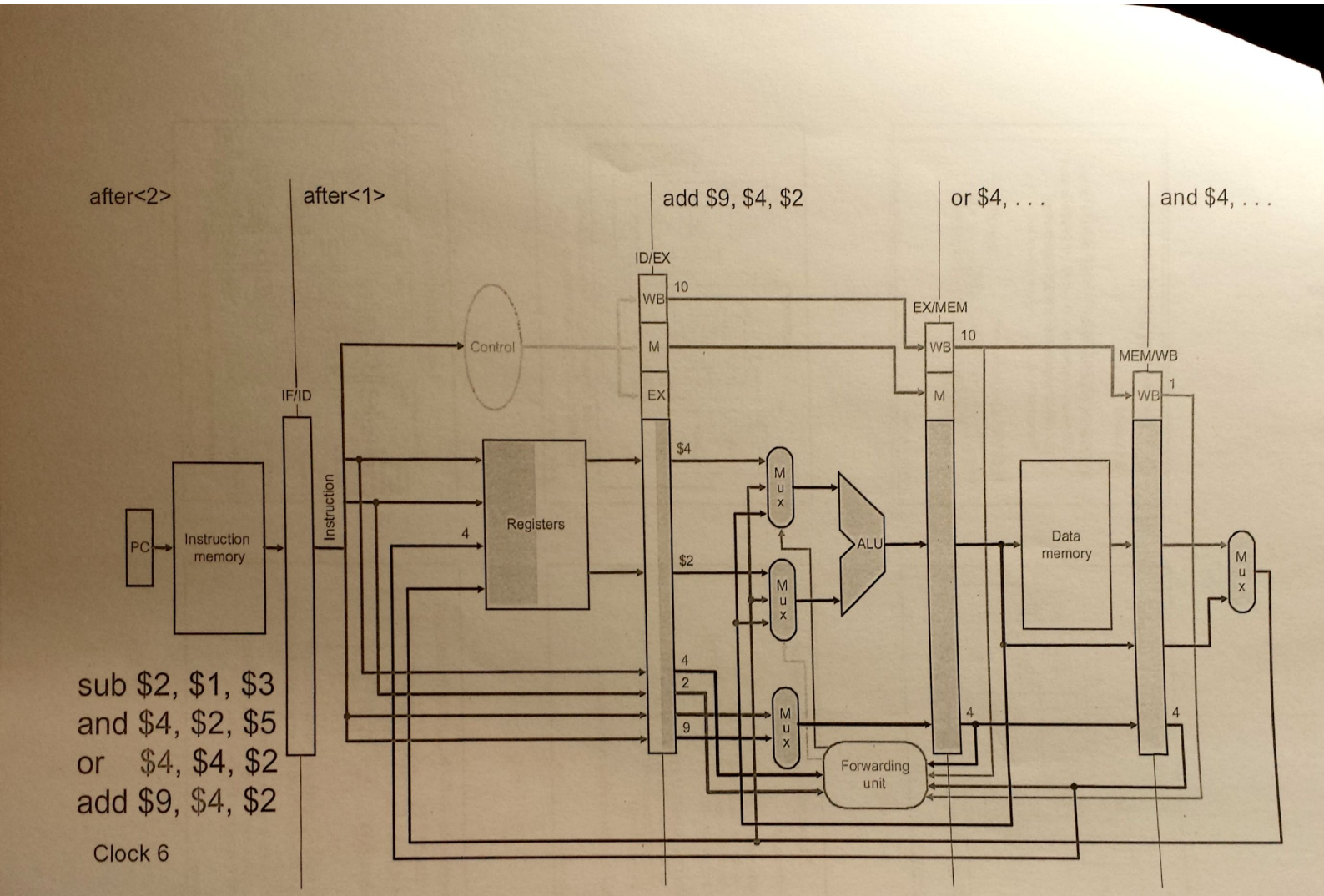
- Skip the boring stuff, jump to cycle 3



- Forward ALUOut to Operand 1

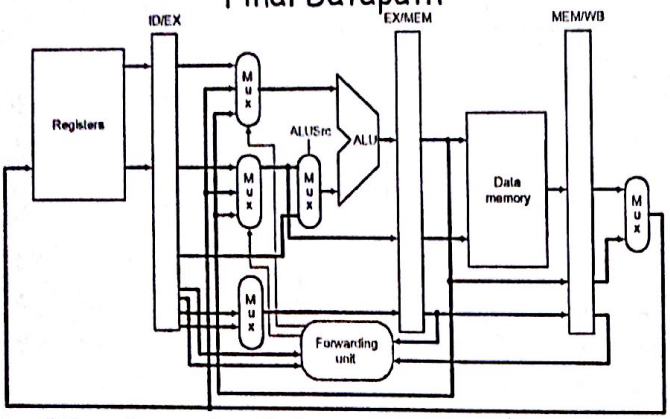


- Forward ALUout to Op1, Mem to Op2



- Two candidates match, forward the latest

Final Datapath



- "Imm" can be 2nd operand (Fig 4.57)

Forwarding Control Behavior

- EX hazard

If (EX/MEM.RegWrite AND // not store or branch
EX/MEM.RegsterRd != 0 AND // Result is used
EX/MEM.RegisterRd = ID/EX.RegisterRs)
ForwardA = 10

If (EX/MEM.RegWrite AND
EX/MEM.RegsterRd != 0 AND
EX/MEM.RegisterRd = ID/EX.RegisterRt)
ForwardB = 10

Forwarding Control Behavior

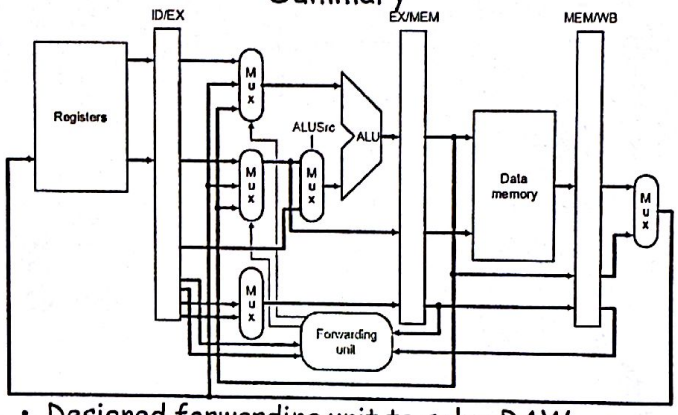
- MEM hazard

If (MEM/WB.RegWrite AND
MEM/WB.RegsterRd != 0 AND
MEM/WB.RegisterRd = ID/EX.RegisterRs)
ForwardA = 01

If (MEM/WB.RegWrite AND
MEM/WB.RegsterRd != 0 AND
MEM/WB.RegisterRd = ID/EX.RegisterRt)
ForwardB = 01

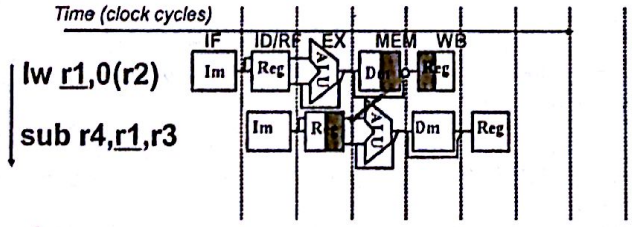
- Does this fully meet our requirements ?

Summary



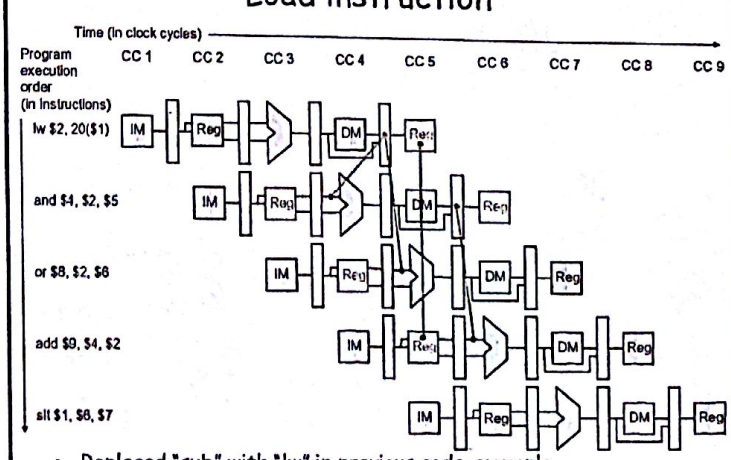
- Designed forwarding unit to solve RAW hazards for R-type instructions

Lookahead: RAW hazard with load inst



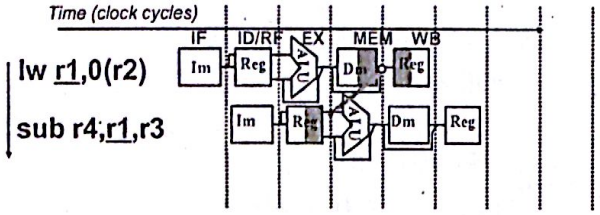
- Forwarding as solution to RAW hazard
  - possible if no (true) dependence going backwards in time
  - True for R-type instructions
    - Data available after EX stage (i.e., at ALUOut)
  - Not true for load instruction

Load instruction



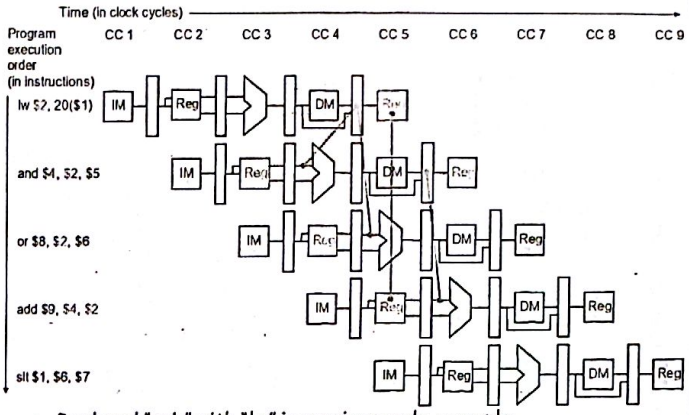
- Replaced "sub" with "lw" in previous code-example

Lookahead: RAW hazard with load inst



- Forwarding as solution to RAW hazard
  - possible if no (true) dependence going backwards in time
  - True for R-type instructions
    - Data available after EX stage (i.e., at ALUOut)
  - Not true for load instruction

Load instruction

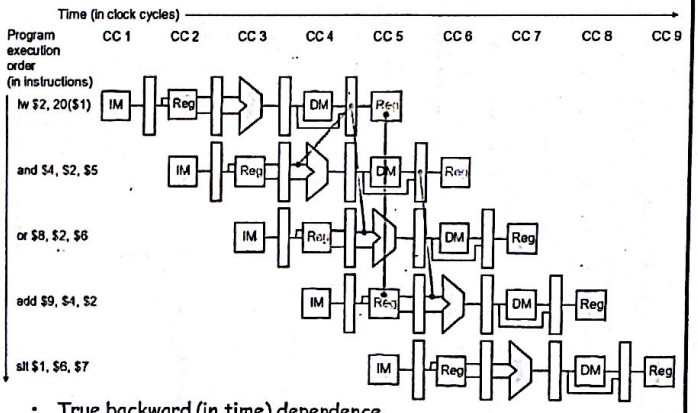


- Replaced "sub" with "lw" in previous code-example

Solution

- Catch-all solution for hazards
  - Stall
    - always works, but hurts performance
    - Use as last resort
- Challenge:
  - Modify pipeline implementation to support stalls when hazards are detected

Load instruction



- True backward (in time) dependence

Hazards with load instruction

- True dependencies: backward in time
- Stall the pipeline
- Minor change in terminology
  - If forwarding can solve it, it is not a hazard!
  - "Hazard" refers only to true backward dependencies in time.

Handling the hazard

- As before
  - Detection
    - Logic equations to detect hazard
  - Actual stalling
    - Datapath/control modifications to achieve stalling

Detection

- Conditions
  - Preceding instruction must read memory
    - MemRead must be asserted
  - Destination of preceding instruction (rt) must be one of operands of current instruction

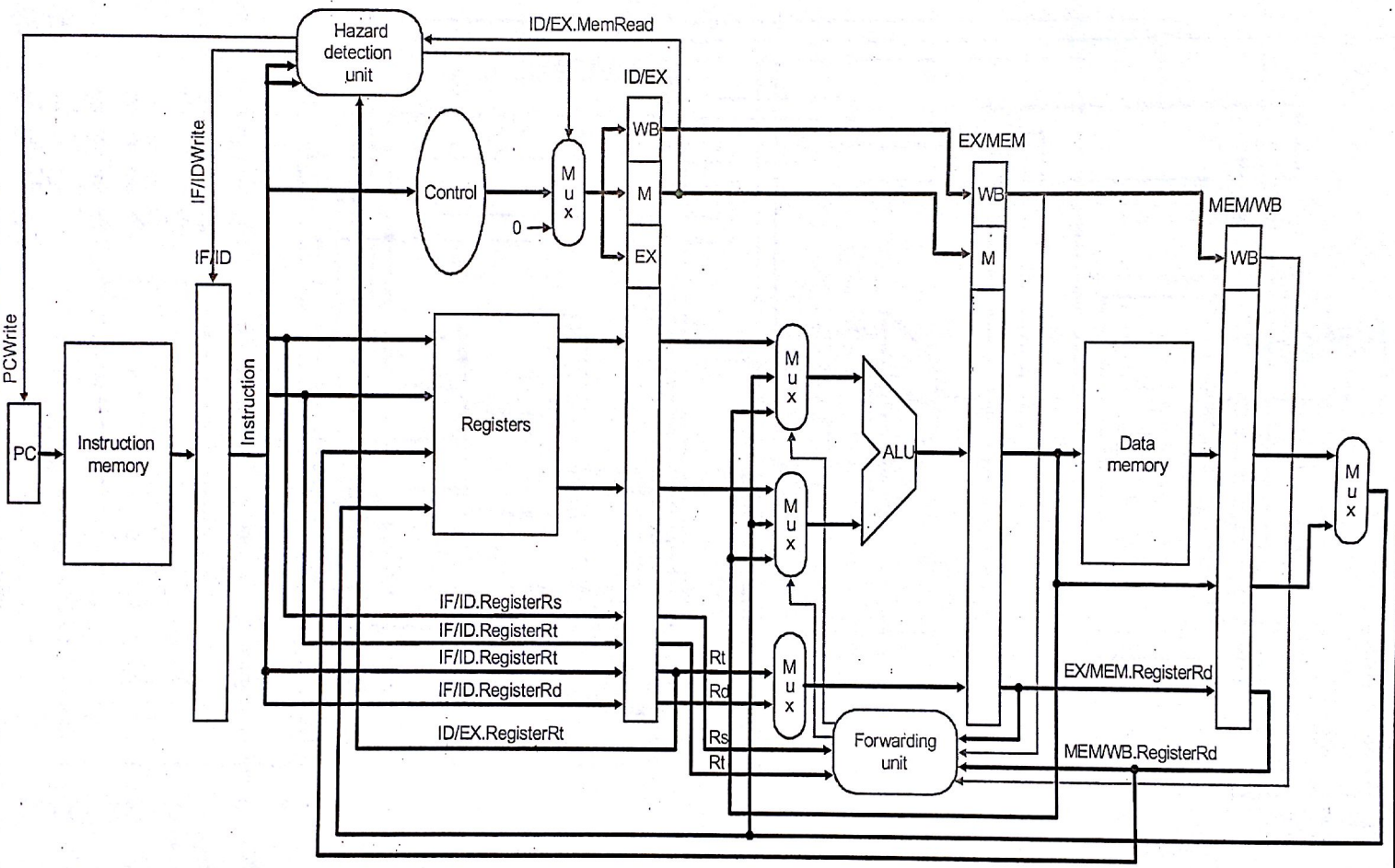
```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```

- Logic equations- restate above conditions formally
  - If(ID/EX.MemRead AND
    ((ID/EX.RegRt = IF/ID.RegRs) OR
    (ID/EX.RegRt = IF/ID.RegRt)))
    STALL

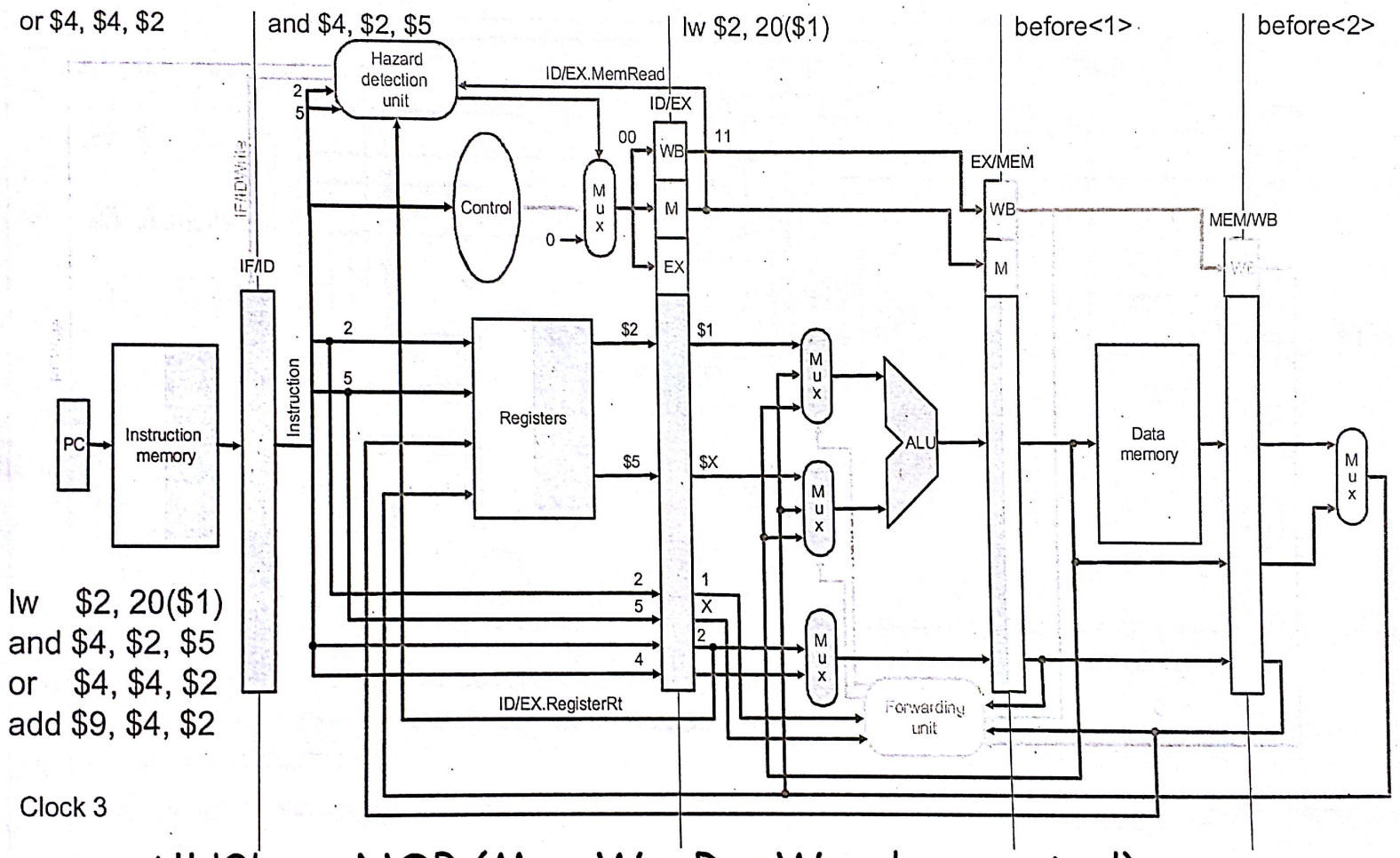
Stalling the pipeline

- Instruction cannot proceed
  - Following instruction must be stalled too.
  - Otherwise state in pipeline registers is overwritten
- Preceding instructions may proceed as usual
- Solution
  - inject NOP into EX/Mem pipeline
  - Prevent writes to PC and IF/ID register

Datapath

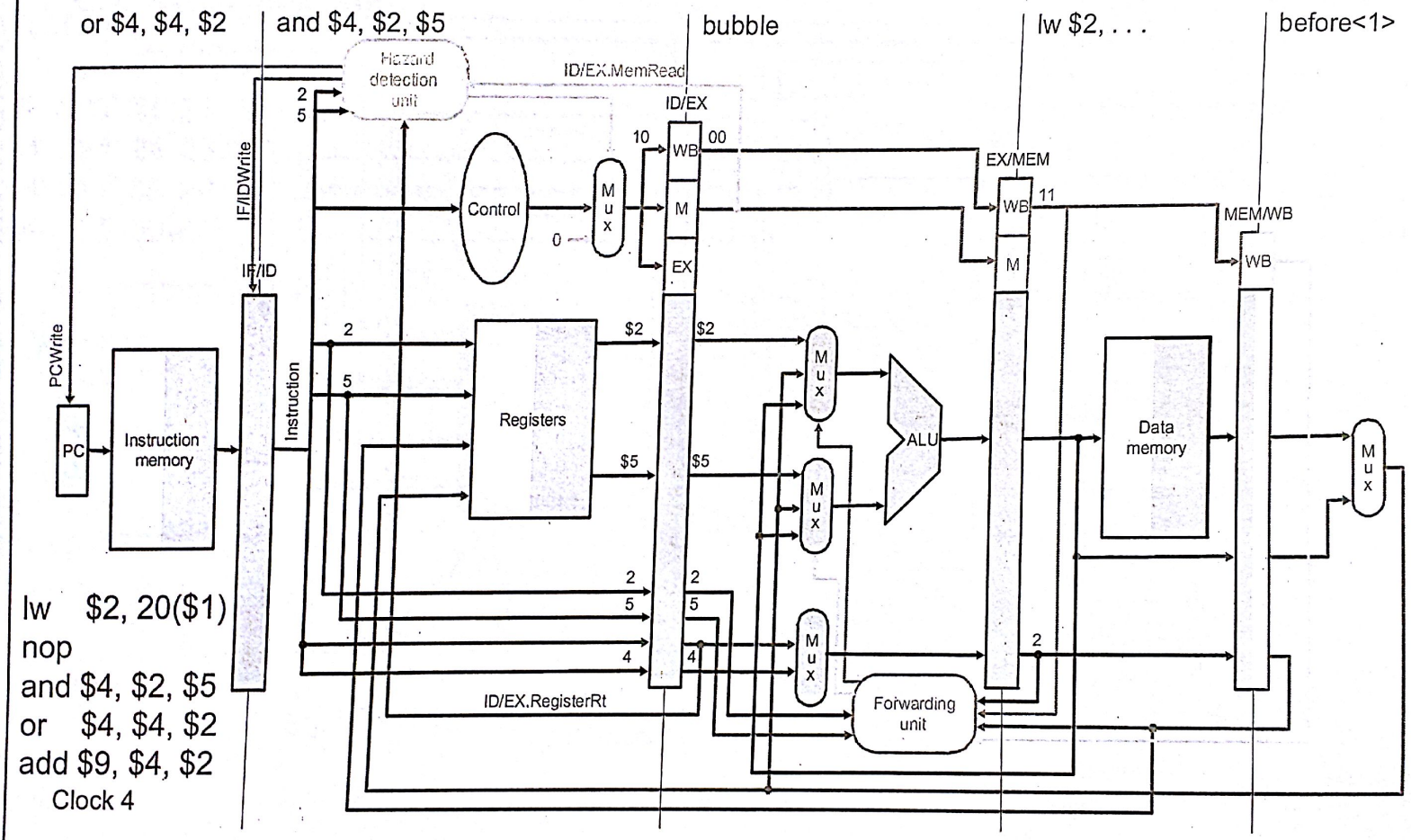


Walk-through (2 of 6)

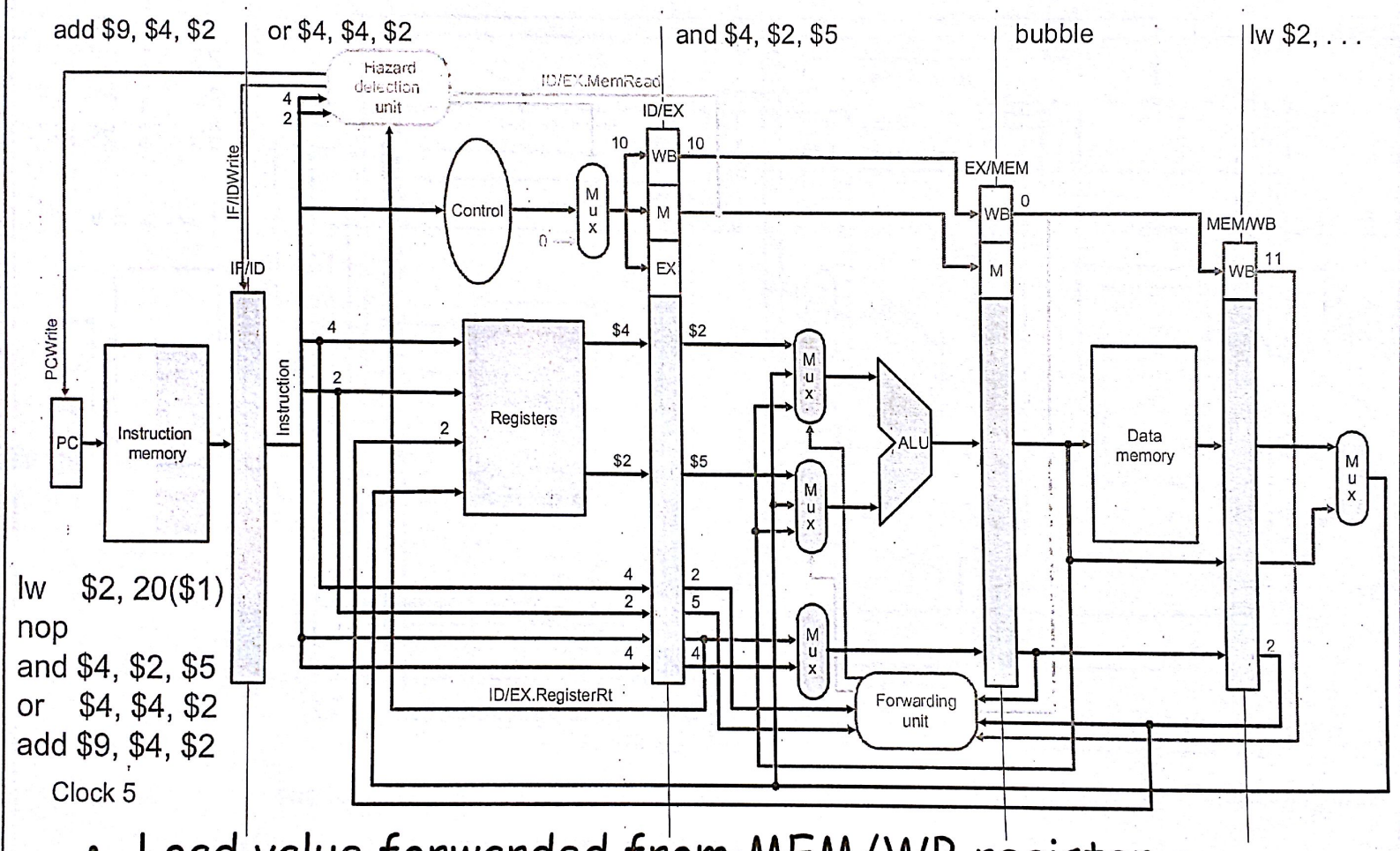


- All '0's => NOP (MemWr, RegWr, deasserted)

Walk-through (3 of 6)

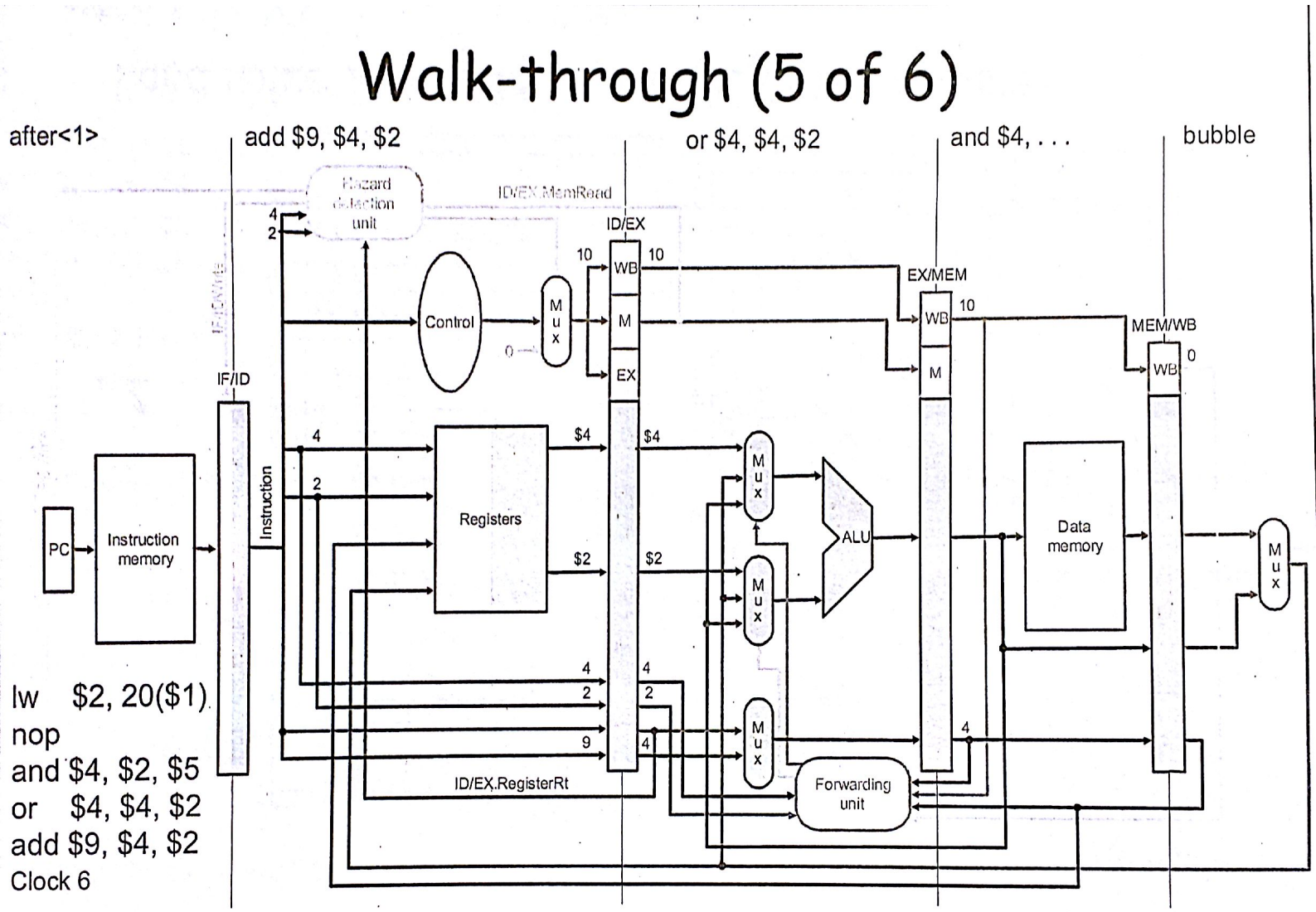


Walk-through (4 of 6)



- Load value forwarded from MEM/WB register

Walk-through (5 of 6)

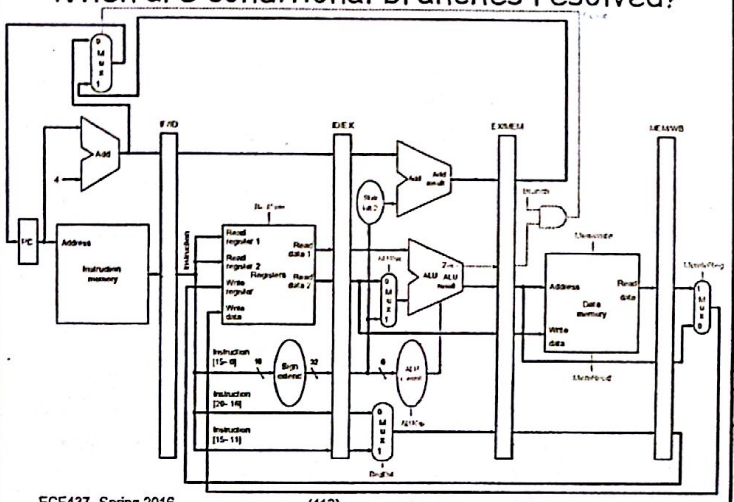


- $4 value forwarded from EX/MEM register

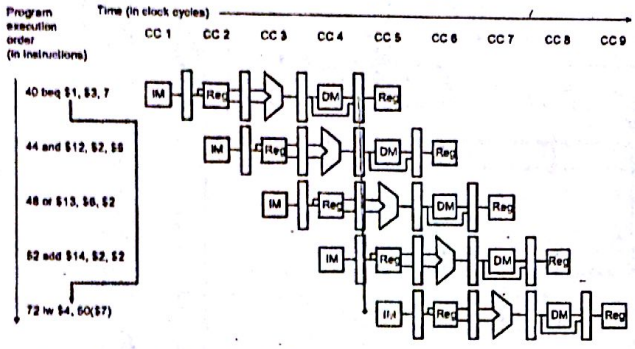
RAW Hazard with Loads: Summary

- True backward dependencies in time
  - Need to stall
- Stall achieved by
  - Detecting hazard (remember logic equation)
  - Inserting NOP (all EX/MEM/WB controls set to 0)
  - Preventing IF/ID register and PC from being overwritten
- Next Branch/Control Hazards

When are conditional branches resolved?



Branch Hazards

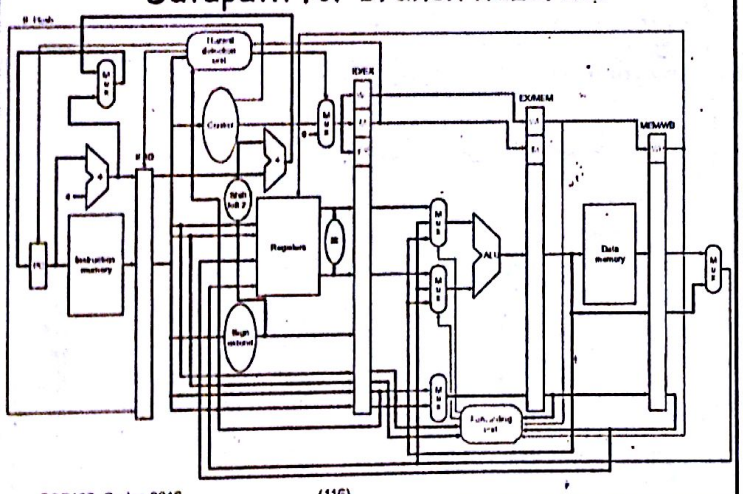


- Branch resolved in the MEM stage
- If taken,
  - PC<- PC + 4 + SX(Imm*4)
  - 40 + 4 + 7*4 = 72

Control/Branch Hazards

- Branch resolved in the MEM stage
  - But next instruction has to fetched in the next cycle
  - Reduce the penalty by moving decision earlier in pipeline
    - Need additional comparator (r1=r2?) and adder (PC+4+SX(IMM)*4)
    - Value needed in earlier stage
      - what if r1/r2 write is pending?
      - Forwarding and/or stalling
  - Reduced penalty from 3 cycles to 1 cycle

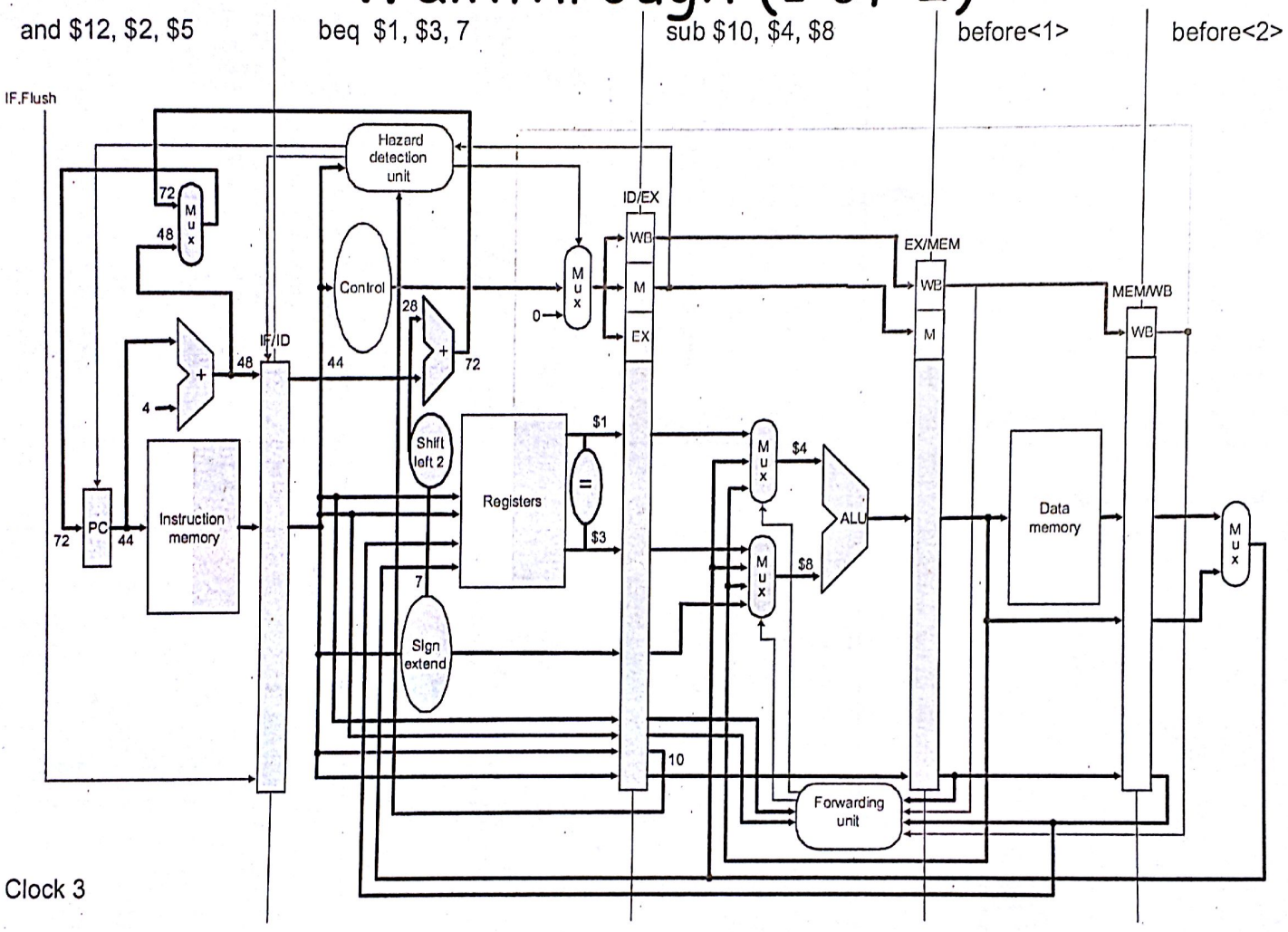
Datapath for branch hazards



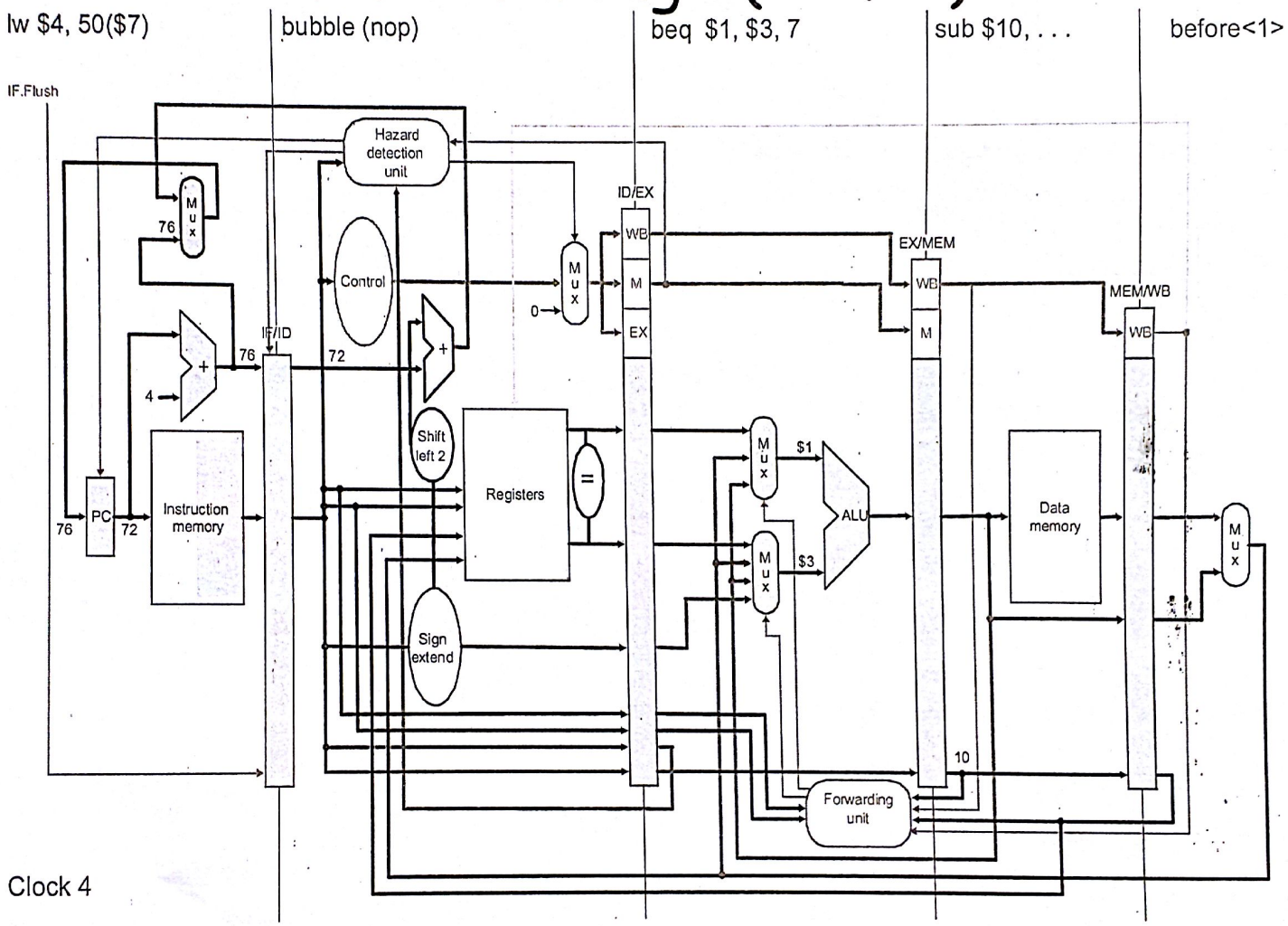
Can we do anything about the 1cycle stall?

- Two solutions
  - Predict branch is always not taken
    - More sophisticated prediction schemes
  - Delay slots
    - Compiler's problem
- Walkthrough example for solution #1
  - Predict not taken

Walkthrough (1 of 2)

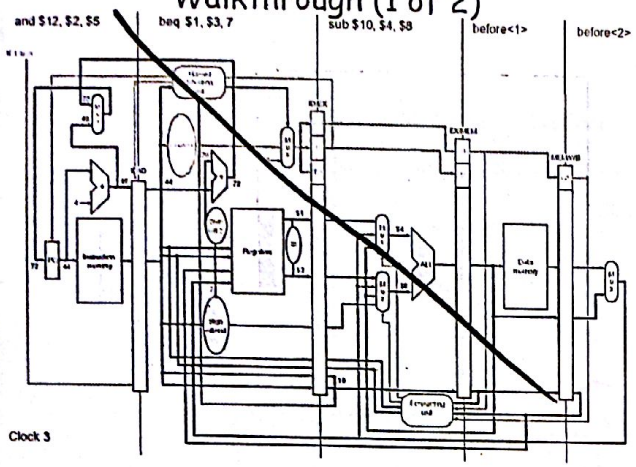


Walkthrough (2 of 2)

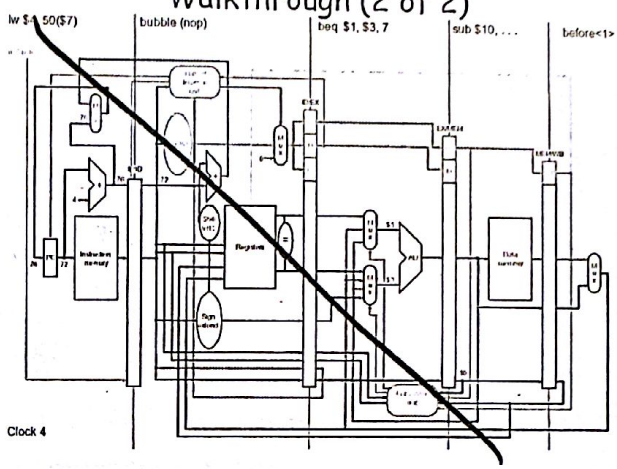


SEE FULL SIZE

Walkthrough (1 of 2)



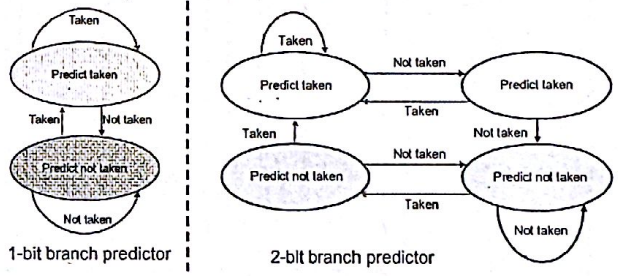
Walkthrough (2 of 2)



Dynamic Branch Prediction

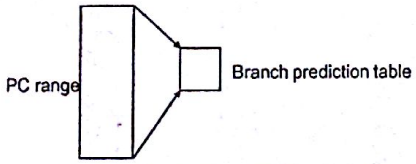
- Better than static prediction
  - Branches are predictable
  - ~90% of program execution time is spent in ~10% of code (inner loops)
  - Think of a program loop of N iterations
    - Taken N-1 times
    - Not taken last time

Dynamic Branch Prediction



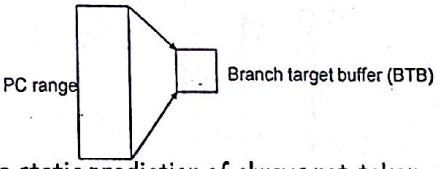
- How does hardware "learn" branch behavior?
- Store each branch instruction's history ***
  - If a branch was taken "recently", predict taken
    - One bit saturating counter
    - Two bit counters

Branch Prediction



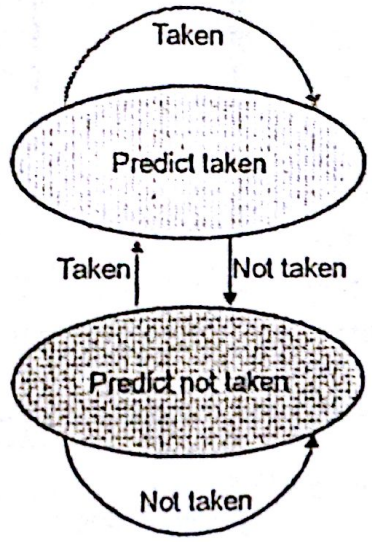
- Store each branch's history ***
  - Not really
- Keep a small table indexed by program counter
- PC is large (32 bit number)
- Mapping to number of table entries
  - E.g. 16-entry branch prediction table
  - Mapping: use last 4 bits of PC
- Problem: Multiple branches may map to same entry in table -- Aliasing

Branch Prediction: Branch Target Buffer

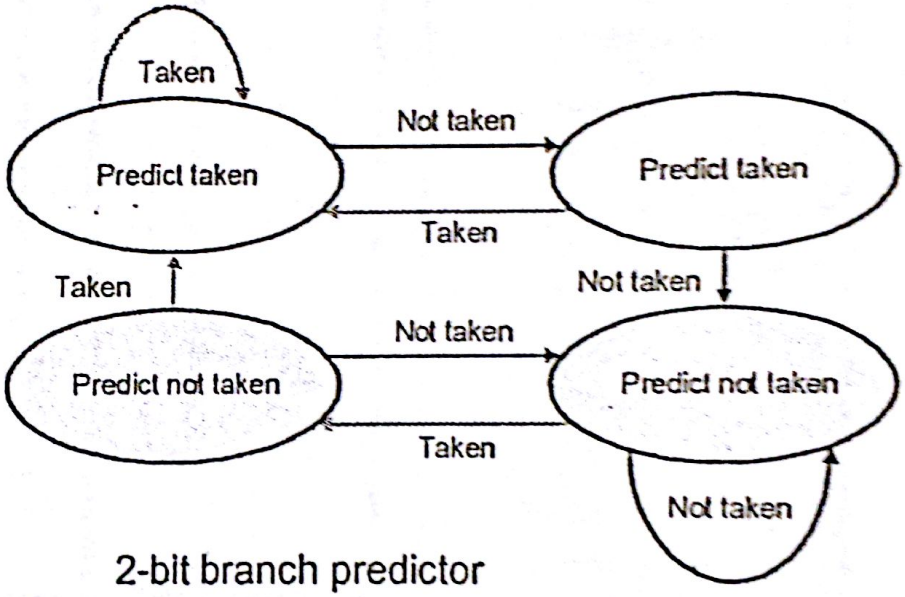


- Unlike static prediction of always not-taken, dynamic prediction will predict both taken and not-taken
- For taken prediction, we have a problem (what?)
- There is something about branch targets that helps us solve the problem (what?)
- Keep a small table of br targets indexed by PC (BTB)
  - WHILE fetching (potentially) a branch, use branch PC to look up prediction AND target of branch so NEXT cycle you can fetch taken instruction if predict taken else fetch not-taken instruction - back-to-back fetch for both taken or not-taken
- Many-to-few mapping from PC to BTB

Dynamic Branch Prediction



1-bit branch predictor



2-bit branch predictor

- How does hardware "learn" branch behavior?
- Store each branch instruction's history ***
  - If a branch was taken "recently", predict taken
    - One bit saturating counter
    - Two bit counters

Recap

- Branch instructions
  - Control flow hazard
  - Static branch prediction
    - Predict not taken
    - Squash instruction if prediction incorrect
- Dynamic Branch prediction
  - 1-bit and 2-bit state machines to track history of branches
  - Finite table
    - Potential for "aliasing"
    - Multiple branches map to the same predictor

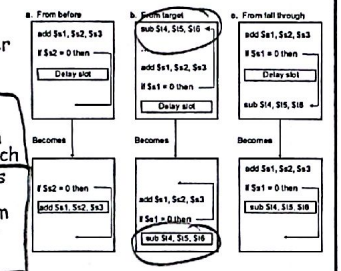
Delayed Branch

10	lw	r1, r2(35)		10	lw	r1, r2(35)
14	addl	r2, r2, 3		14	addl	r2, r2, 3
20	sub	r3, r4, r5		20	sub	r3, r4, r5
24	orl	r8, r9, 17		24	beq	r6, r7, 100
20	beq	r6, r7, 100		30	orl	r8, r9, 17
34	add	r10, r11, r12		34	add	r10, r11, r12
100	and	r13, r14, 15		100	and	r13, r14, 15

- Delayed branch: inst after branch always executed
  - Invisible to programmer
  - Compiler and/or assembler transforms code

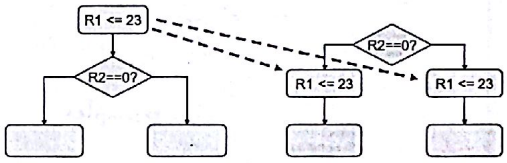
Easy way*** to hide branch hazard delay

- Delayed branch
  - Instruction after branch always executes
  - Find an independent instruction from before the branch
  - Find instructions from Taken (target) OR from Not Taken (fall-through) code section
- *** For Architects



eg. 1

Ideal delay slot operation



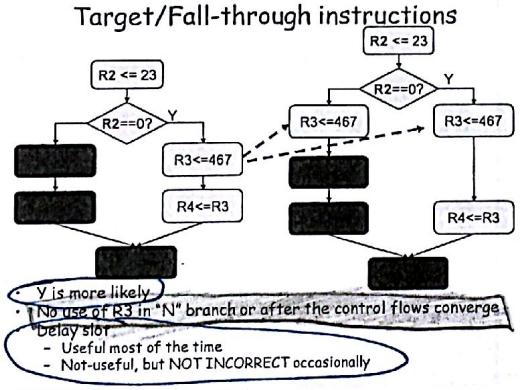
a gets the value 6

a <= b

- Independent instructions to fill delay slot
- Code transformation preserves original semantics

eg. 2

Target/Fall-through instructions



- Y is more likely
- No use of R3 in "N" branch or after the control flows converge
- Delay slot
  - Useful most of the time
  - Not-useful, but NOT INCORRECT occasionally

What next?

- Exceptions
  - Multiple instructions in flight
  - PC has changed
- Advanced topics
  - Superscalar, dynamically scheduled processors, etc
- Real machines
  - ~~Core 2~~ pipeline, Niagara Pipeline Pentium 4

Exception → program transfer (internal)
(Traps) eg. On illegal op. code, divide 0
VM when page fault'

in VM. → Precise :
→ Imprecise :

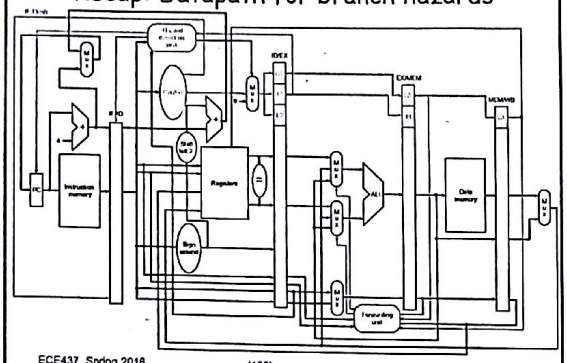
Interrupts → external event triggered.
Asynchronous.

disk: milli sec.
solid state: nano sec.

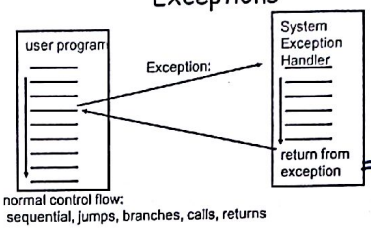
eg. illegal op code

MIPS defines running an exception handler incase of an exception.

Recap: Datapath for branch hazards



Exceptions



=> returns to PC, not PC+4.

∴ of virtual memory impl. come back so that changes made in handler would (maybe) run this instruction.

- Exception = unprogrammed control transfer
  - system takes action to handle the exception
    - must record the address of the offending instruction
  - returns control to user
  - must save & restore user state
- Allows construction of a "user virtual machine"

Interrupt, Exception, Trap?

- Interrupts
  - caused by external events
  - asynchronous to program execution
  - may be handled between instructions
  - simply suspend and resume user program
- Traps
  - caused by internal events
    - exceptional conditions (overflow)
    - errors (parity)
    - faults (non-resident page)

    } examples
  - synchronous to program execution
  - condition must be remedied by the handler
  - instruction may be retried or simulated and program continued or program may be aborted
- MIPS convention:
  - External: Interrupts
  - Internal: Exception

Exception Semantics

- MIPS architecture defines the instruction as having no effect if the instruction causes an exception.
- When get to virtual memory we will see that certain classes of exceptions must prevent the instruction from changing the machine state.
- This aspect of handling exceptions becomes complex and potentially limits performance => why it is hard
  - Precise interrupts vs Imprecise interrupts] refer to exceptions.

Exceptions

- Pipeline Semantics
  - No instruction after the exception causing instruction may execute
  - Every instruction preceding the exception causing instruction must complete execution

MIPS Exceptions

- All exceptions jump to same handler code
  - "Cause" register
- We consider
  - Illegal instructions
  - Arithmetic overflows
- Handler behavior
  - Save PC of offending instruction (How? PC+4 has already been written to PC)
  - Use special register EPC (why not use $31 like jal?)
  - Set cause register appropriately (0=ILL; 1=OVF)
  - Jump to handler at fixed address

illegal instr, overflow

can be over-written (risky)

overly simplified ex.

Say I get lw instr. but the value to load is in disk & hence unavailable. The exception is thrown, program will come back later. In this time exception handler gets ready for

Virtual memory

-> 2 programs run together. They have total addr space. Still do not coincide with each other.

-> Virtual memory (Translation mechanism) gets maps different addr. in diff places.

Datapath modifications

- Pipeline complications
- What stage is exception detected?
  - Overflow?
    - In EX stage, Also squash (convert to nop) EX stage
  - Illegal Instruction?
    - In ID stage, squash (convert to nop) ID stage
    - Similar to RAW hazard
  - What about external interrupts?
- Overflow in instruction i, illegal instruction in instruction i+1
  - Simultaneous exceptions
  - Hardware sorting

If both then must prioritize -

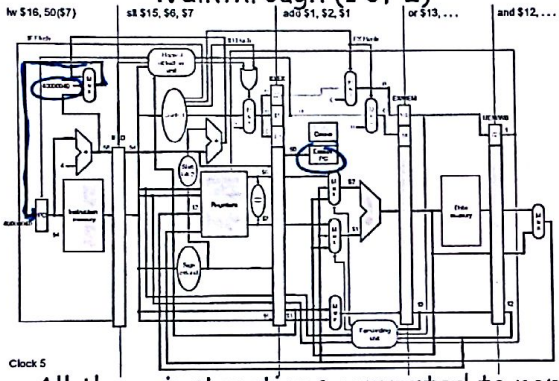
In real cases, must do more crazy shit.

Walk-through: Code snippet

- Main Code
  - 40 sub $11, $2, $4
  - 44 and $12, $2, $5
  - 48 or $13, $2, $1
  - → 4C add $1, $2, $1
  - 50 slt $15, $6, $7
- Exception Code
  - ~~[EPC]~~ sw $25, 1000($0)

0x40000040 (Cause PC for exceptions)

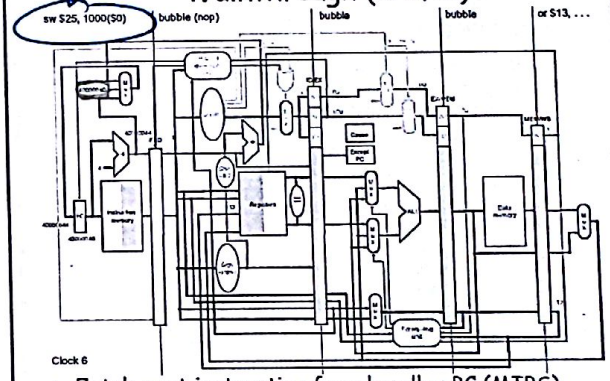
Walkthrough (1 of 2)



- All three instructions converted to nop

1st inst of exception handler.

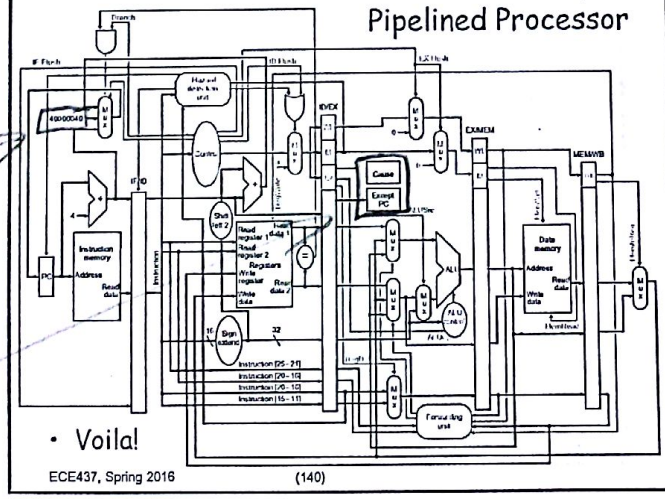
Walkthrough (2 of 2)



- Fetch next instruction from handler PC (MIPS)

Note: except PC saves PC+4 but we want PC, so, must have subtraction circuit.

Pipelined Processor



- Voila!

Understanding Performance

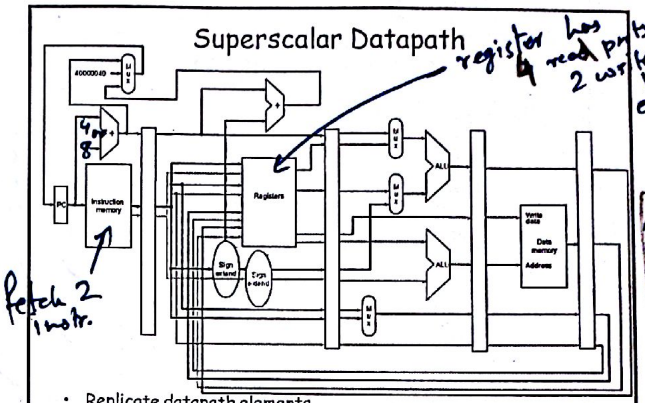
- Iron law: Insts/prog * CPI * cycletime
- With pipelining:
  - CPI ~ 1 (with ideal memory, good branch prediction and few data hazards)
  - Cycletime : determined by critical path of one stage

defacto standard [even in phones]

Superscalar Processor

- What does it mean?
  - Scalar processors (operate on scalar quantities)
  - Vector (operate on vectors) [eg. add 2 vectors of arrays]
  - Superscalar: multiple scalar operations in one cycle
  - More than one instruction per cycle

Superscalar Datapath



register has 4 read ports, 2 write ports. etc...

Fetch 2 instr.

ARM microcontroller less than $1 also use this

- Replicate datapath elements
- Static Multiple issue datapath

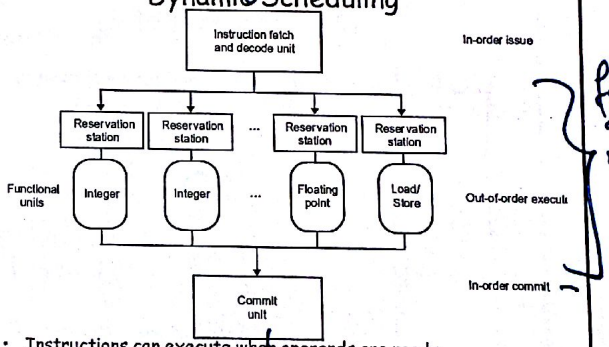
Dynamic Scheduling

- No need to suffer hazards if other useful work can be achieved
- Load Hazard results in pipeline stall
  - But other instructions are ready
  - "Oh! But we cannot execute instructions out of order" - Not really

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, $t3
```


enter in order

Dynamic Scheduling



function ops when values are ready

- Instructions can execute when operands are ready
- Instructions can "commit" when all preceding instructions have committed

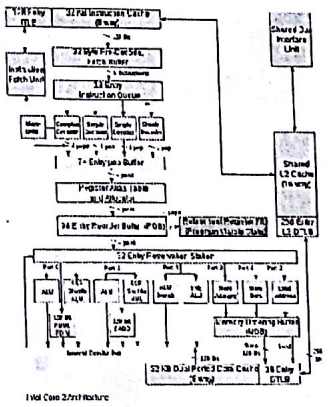
exit in order.

Real machines

- Let's examine Core 2
  - Microarchitecture more or less stable
  - Technology has improved

Core 2

- 45nm
- Multiple decodes
- 14 stage pipeline
  - (went as high as ~31 in Pentium 4 line)
  - Many other considerations
    - Pipelining for yield
- Source: Wikipedia



came as 24 core server when intel was on 2 core

similar but 6 stages.

kills many hazards by picking different threads.

operating system selects running threads.

0 bubble pipeline

Sun Niagara

- Not too dissimilar
- 4 threads
- Eight such processors on a chip
- March/April 2005 Issue of IEEE MICRO

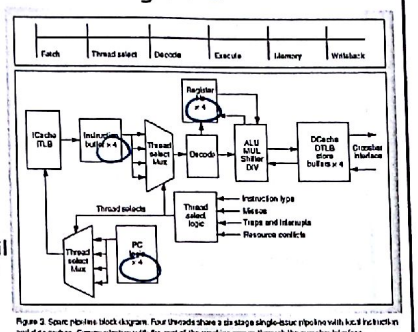
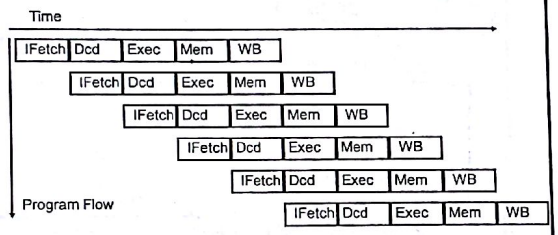


Figure 3. Sparc pipeline block diagram. Four threads share a six-stage single-issue pipeline with local instruction and data caches. Communication with the rest of the machine occurs through the crossbar interface.

Pipelining Performance

- Start with ideal assumption
- Gradually introduce realism
  - Delay through all stages not equal
  - Structural hazards
  - Data (RAW) hazards
  - Control Hazards
  - Speedup

Pipelined Execution Representation



- Ideal speedup =?

Review: Ideal speedup

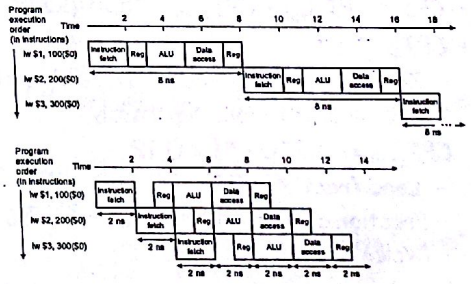
- All instructions are executed in P pipeline stages in a multicycle path (i.e. CPI = P)
- Cycletime = t ns (say)
- Instr. Count = n
- Old time = P x t x n
- New time = n x t + (P-1) x t
- Speedup = P/(1 + (P-1)/n) ≤ P
- P is some constant, n is large => Speedup ≈ P

Why pipeline?

- Suppose we execute 100 instructions
- Single Cycle Machine
  - 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
- Multicycle Machine
  - 10 ns/cycle x 4.2 CPI (due to inst mix) x 100 inst = 4200 ns
- Ideal pipelined machine
  - 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns

Better model

- Next dose of reality: non-uniform stage delays



Ideal speedup is number of stages in the pipeline. Do we achieve this?

$$\text{SpeedUp} = \frac{N \times t \times (CPI \times P)}{(N-1) \times t \times CPI + 1 \times t \times (CPI \times P)} = \frac{PN}{P+N-1} \Big|_{N \to \infty} = P$$

Non-uniform stages

Maximum Speedup ≤ Number of stages

Speedup ≤ $\frac{\text{Time for unpipelined operation}}{\text{Time for longest stage}}$

Recap exercise

- A single cycle processor implementation can be pipelined in two ways
- Pipeline A uses a 5-stage pipeline
  - the 5 stages account for 15%, 10%, 20%, 20%, 35% of the delays respectively
- Pipeline B uses a 3-stage pipeline
  - the stages are balanced
- If instructions are all independent, which pipeline implementation is the better option

speedup = $\frac{100}{35}$ = 2.85

speedup = $\frac{100}{33.3}$ = 3

Third Dose of Reality

- Structural hazards
  - E.g. single memory
  - Say 30% in instructions are memory operations
  - 1.3 memory accesses/instruction
  - CPI is atleast 1.3 (otherwise memory is used more than 100%)
  - State of the art: Two memories (caches) to eliminate structural hazards

can inst. & data fetch only one at a time.

Fourth Dose of Reality

- Data hazards
  - We can handle R-type RAW hazards with zero penalty (forwarding)
  - Loads require stalls
    - Instruction mix: 20% loads, 80% other
    - Hazards: 60% of load values are used by the immediate next instruction
    - CPI = 0.8*1 + 0.2* (0.6*2+0.4*1) = 1.12
  - What about WAR and WAW hazards?

CPI is throughput dependent.

here we tie the NOP to the load instruction.

= 1 + stall cycles

here we don't tie NOP with operation (ie). Simply look at how often we stall.

Alternate view (Easier to compute)

- CPI = CPI_{ideal} + CPI_{stall}
- CPI_{ideal} = 1
  - Includes cost of executing all instructions
  - Reason about stalls separately
- CPI_{stall} = 0.2*0.6*1 = 0.12
  - Load fraction = 0.2
  - Fraction of loads that cause stall = 0.6
  - Cycles of stall each time= 1

Fifth Dose of Reality

- Branch Hazards
  - Stall depends on where branch is resolved
  - Assume ID stage (with extra hardware)
    - 1 cycle penalty
      - Can fill delay slot with useful instructions
      - Can predict branch outcome
  - Branches constitute 20%, delay slot can be filled 90% of the time
    - CPI = 0.8*1 + 0.2 * (0.9*1 + 0.1*2) = 1.02
  - Branches constitute 20%, prediction accuracy is 90%
    - CPI = 0.8*1 + 0.2 * (0.9*1 + 0.1*2) = 1.02

where prediction was wrong.

Alternate view (Easier to compute)

- $CPI = CPI_{ideal} + CPI_{stall}$
- $CPI_{ideal} = 1$
  - Includes cost of executing all instructions
  - Reason about stalls separately
- $CPI_{stall} = 0.2*0.1*1 = 0.02$
  - Branch fraction = 0.2
  - Fraction of branches that cause stall = 0.1
  - Cycles of stall = 1

Mix and match

- Detailed instruction mix
  - Load frequency and hazard frequency
  - Branch frequency and branch misprediction ratio
  - Deal with each term separately

Develop ability to correlate concepts

- 5-stage pipeline with no other hazards
  - Inst mix: 20% branches, 80% other
- Branch prediction in ID stage
  - Scheme A: 70% accuracy
  - Scheme B: 90% accuracy, but 10% increase in cycle time
  - Which is better?

Find CPI first. Then, multiply with cycle time

Develop ability to correlate concepts

- Requires CPI computation, iron law
  - CPI(A) = 0.8*1 + 0.2*(2*0.3 + 1*0.7) = 1.06
  - CPI (B) = 0.8*1 + 0.2*(2*0.1 + 1*0.9) = 1.02
  - Cycletime (A) = t
  - Cycletime(B) = 1.1*t
  - Insts/prog is the same for both
  - Iron law:
    - CPI(A) *cycletime(A) = 1.06 * t
    - CPI(B) * cycletime(B) = 1.02 * 1.1 * t = 1.122 * t

Alternate View

- Count CPI for instructions and stalls separately
- $CPI = CPI_{ideal} + CPI_{stall}$
- CPI_{ideal} (A) = 1
- CPI_{stall} (A) = 0.2*0.3*1 = 0.06
  - Branch fraction = 0.2
  - Fraction of branches that cause stall = 0.3
  - Cycles of stall = 1
- CPI_{ideal} (B) = 1
- CPI_{stall} (B) = 0.2*0.1*1 = 0.02
  - Branch fraction = 0.2
  - Fraction of branches that cause stall = 0.1
  - Cycles of stall = 1

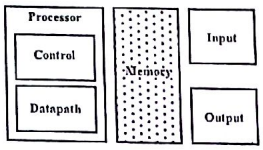
Think: Why is this view equivalent to the view on the previous slide?

Finally multiply by 1.1 for comparison.

Summary

- Exceptions
  - Know how to handle the easy cases
    - What to squash, what not to
  - Know how complicated exceptions can be
- Read Chapter 4 NOW
  - Maximize impact
  - Study while lecture material is "warm".
    - 2-3 hours now vs. 6-8 hours later

Outline



- Memory
  - Technology, organization, motivation for hierarchical organization

Memory

- Storage elements
  - registers, latches.
    - Small
    - In processor
    - Expensive to add (??)

Processor Datapath

~1 cycle

- SRAM (Caches) ← bigger cache is better. many even over-size per clock speed.
  - Medium
  - Onchip or board, close to processor
  - Costly

~20 cycle

- DRAM (Main memory)
  - Large, 50ns access time
  - Today 0.4¢/MB (16GB for $67.98)
  - Spring 2016 (0.36¢/MB 8GB/$30) Spring 2011 0.62¢/MB, Spring 2008, $0.02/MB (4GB for $79 *)
  - In 2006 $0.06-0.10/MB (512MB for 35-45$ *)

~300 cycles

Memory subsystem

- Disk/Tape etc.
  - Large, far from processor
  - Slow (~ms)
  - Today 4¢/GB (2TB for $80)
  - Spring 2016 (2.4¢/GB 4TB/$100) Spring 2010, 7.5¢/GB (1TB for $75*) Fall 2008, $0.11/GB (1TB for $108.99) Spring 2008, $0.22/GB (500GB for $112 *)
  - In 2006 $0.35-0.40/GB (200GB for 70$ *)

I/O subsystem

Memory Hierarchy Technology

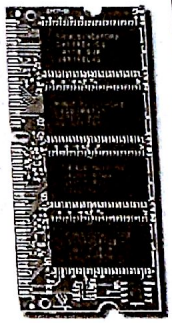
- Random Access: ← can access all/any memory directly (with similar time)
  - "Random" is good: access time is the same for all locations
  - DRAM: Dynamic Random Access Memory
    - High density, low power, cheap, slow
    - Dynamic: need to be "refreshed" regularly
  - SRAM: Static Random Access Memory
    - Low density, high power, expensive, fast
    - Static: content will last "forever" (until lose power)
- "Not-so-random" Access Technology:
  - Access time varies from location to location and from time to time
  - Examples: Disk, CDROM
- Sequential Access Technology: access time linear in location (e.g., Tape)
- The Main Memory: DRAMs + Caches: SRAMs

→ refresh all row in 50ms

DRAM

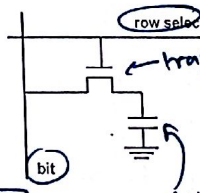
- Dynamic RAM
  - Dense, 1T/bit-cell
  - Forgets after a while
  - 16Mb : 4K x 4K cell-array
  - 24 bit address
    - 12 bit for row, 12 for column—reflected in the interface
- Implementation
  - Word/byte DRAM built as DIMM/SIMMs

In 3B7 we need 13 bits for addressing.



1T DRAM cell

- Charge on capacitor
- Write:
  - 1. Drive bit line
  - 2.. Select row
- Read:
  - 1. Precharge bit line to Vdd
  - 2.. Select row
  - 3. Cell and bit line share charges
    - Very small voltage changes on the bit line
  - 4. Sense (fancy sense amp) ← sense changes
    - Can detect changes of ~1 million electrons*
  - 5. Write: restore the value
- Refresh
  - 1. Just do a dummy read to every cell.



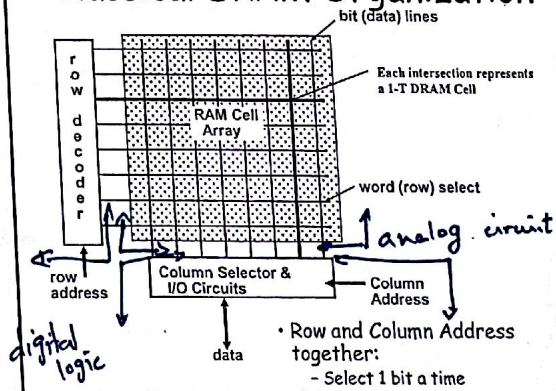
Bit line is plain metal. So on Hi-Z, it acts as capacitor & saves old value

reading destroys value & so must read again. write

write op:
set row high.
set bit low or high.
The value is written.
→ If we want to change/write sub-row, must read whole row first.

read op

Classical DRAM Organization



- Row and Column Address together:
  - Select 1 bit a time

DRAM Optimizations

- Fast Page Mode:
  - Row once, vary column address
  - Now universal
- EDO DRAM: Extended data out
  - FPM plus pipelining
- Synchronous DRAM
  - Tied to system clock, increasing bus-speed
  - SDRAM-DDR, DDR-2?
- RDRAM (Rambus)

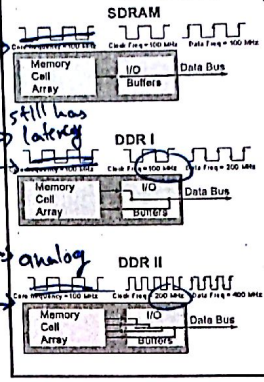
fast page mode

If many reads on same row, does all together

transfer data on bus changes. Analog tech still same.

company that made design & making money out of it.

DRAM organizations



still has latency

analog

- DRAM core unchanged
- Organization/data transfer optimizations
- Compare SDRAM vs. DDR vs DDR2

- *Picture source: http://www.lostcircuits.com/ via xbitlabs.com*

DDR 3

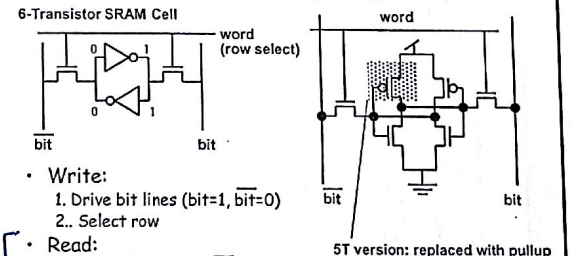
- Same trajectory
  - Improved transfer rates (Digital part)
    - 8 transfers vs. 4 transfers
- Other changes
  - Voltage reduction (low power)
  - Technology change

SRAM

- Data is static (as long as power is applied) (static != non-volatile)
- Logically, two cross-connected inverters with switches
  - CMOS inverter, MOS switch
  - 6-transistor implementation

6 transistor

6T SRAM Cell



- Write:
  1. Drive bit lines (bit=1, bit=0)
  2. Select row
- Read:
  1. Precharge bit and bit to Vdd
  2. Select row
  3. Cell pulls one line low
  4. Sense amp on column detects difference between bit and bit

5T version: replaced with pullup to save area

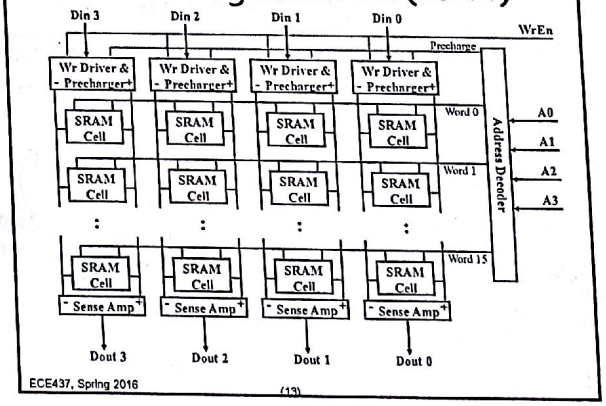
same value

Read is (not) destructive. → No need to write after read.

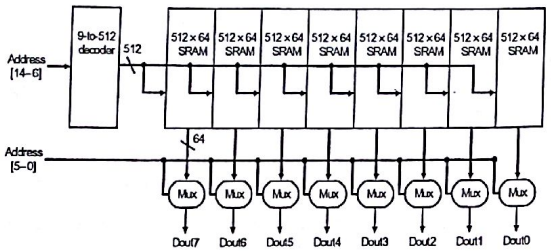
differential sense amp since one of bit & bit will change more than the other.

Also, NO refresh.

SRAM Organization (16x4)



SRAM Organization



- Internal arrays may be different
  - 32Kx8 array realized with 8 512x64 arrays

Memory Technology Summary

- Performance of Main Memory:
  - Latency: Cache Miss Penalty
    - *Access Time*: time between request and word arrives
    - *Cycle Time*: time between requests
  - Bandwidth: I/O & Large Block Miss Penalty (L2)
- Main Memory is *DRAM*: Dynamic Random Access Memory
  - Dynamic since needs to be refreshed periodically (8 ms)
  - Addresses divided into 2 halves (Memory as a 2D matrix):
    - *RAS or Row Access Strobe*
    - *CAS or Column Access Strobe*
- Cache uses *SRAM*: Static Random Access Memory
  - No refresh (6 transistors/bit vs. 1 transistor /bit)
  - Address not divided
- *Size*: DRAM/SRAM *256-512,*
- *Cost/Cycle time*: SRAM/DRAM *32-64*

Technology Trends

DRAM

Year	Size	Cycle Time
1983	256 Kb	220 ns
1986	1 Mb	190 ns
1989	4 Mb	165 ns
1992	16 Mb	145 ns
1995	64 Mb	120 ns
1998	256 Mb	
2001	1 Gb	60ns
2004	4 Gb	50ns**
2008	16 Gb	45ns

	Capacity	Speed (latency)
Logic:	2x in 3 years	2x in 3 years-Nope!
DRAM:	4x in 3 years	2x in 10 years
Disk:	4x in 3 years	2x in 10 years

3D Ram is new & coming out.

Problem

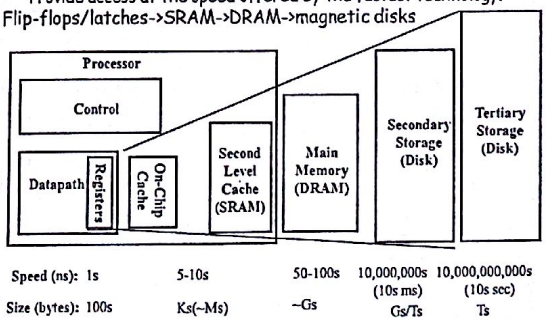
- Why do we care about the memory system?
  - CPU only as fast as mem-system can supply
- Understand SRAM/DRAM technology
- Exploit program characteristics to overcome processor-memory gap

Challenge: Proc-Mem Gap

- Fact: Large memories are slow (and cheap), fast memories are small (and expensive)
- How do we create a memory that is large, cheap and fast (most of the time)?
  - Hierarchy
  - Parallelism

The Memory Hierarchy

- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.
- Flip-flops/latches->SRAM->DRAM->magnetic disks

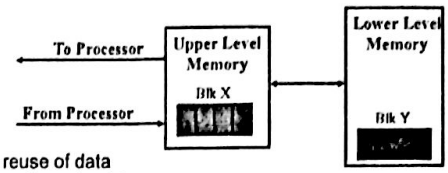


's' is not second.
it is like one-s of ns... .
hundreds of bytes...

Locality

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
- A library
  - Finding the few books you want: Slow
  - One you found the books
    - Reading various chapters: Fast
    - Switching between books: Fast
  - Library -> Memory: Larger the better
  - Books at table -> Cache: Size is limited but access is faster

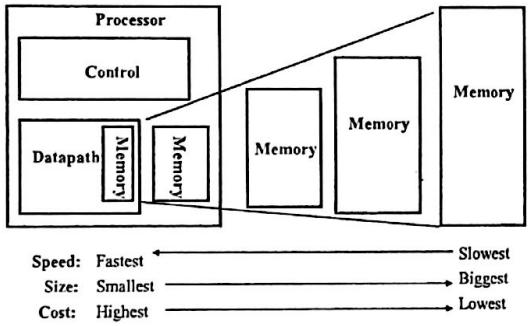
Two flavors of locality



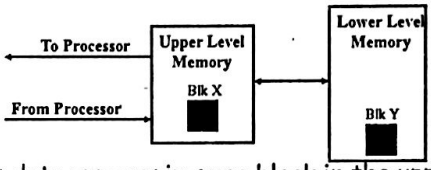
Temporal locality – reuse of data
Spatial locality – use of nearby data

- Temporal Locality (Locality in Time):
  - ⇒Keep most recently accessed data items closer to the processor
  - ⇒Odds are you'll refer to books on your table more than once
- Spatial Locality (Locality in Space):
  - ⇒Move blocks consists of contiguous words to the upper levels
  - ⇒Odds are you'll read contiguous pages/chapters

Tradeoffs



Illusion of Speed and Capacity



- Hit: data appears in some block in the upper level (example: Block X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: Time to access the upper level which consists of RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieved from a block in the lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor
- Hit Time << Miss Penalty

Why caches work

- Ahmdahl's law and caches
  - Big win if: cache-hits are common case
  - Property of programs: "Locality"
- Average memory access time
  - F(hit-time, miss-rate, miss-penalty)
  - 5ns + 10% *50ns = 5 + 5 = 10ns

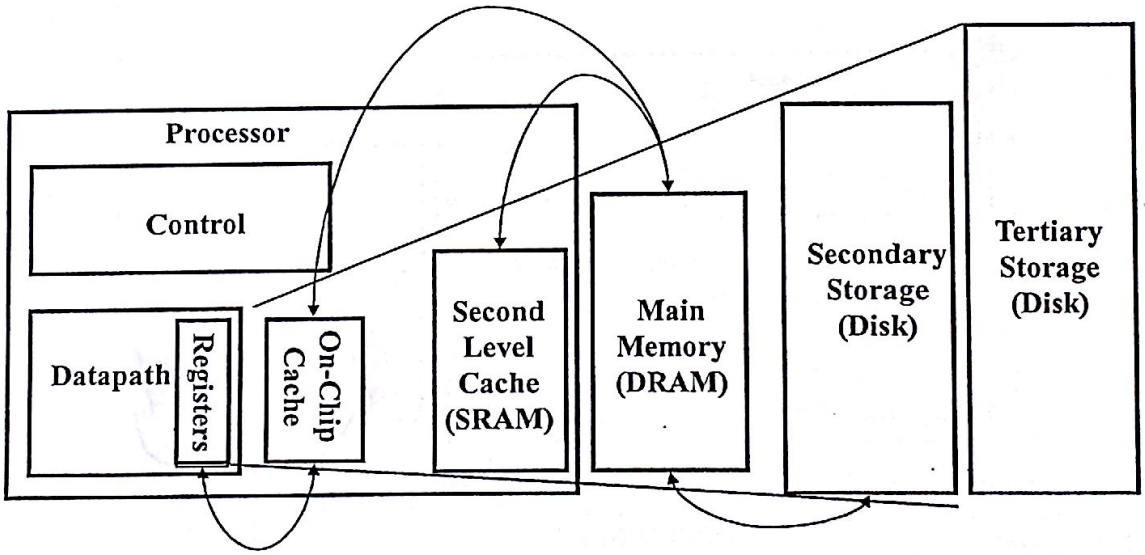
Lookahead: Caches

- Several issues:
  - Determine hit/miss: How do we know if a data item is in the cache?
  - If it is, how do we find it?
  - If it isn't, where do I place it?
  - Replacement: What do we do with data that was present?
  - Who manages this? Compiler? Hardware? Software/OS/Programmer?

Lookahead: Caches

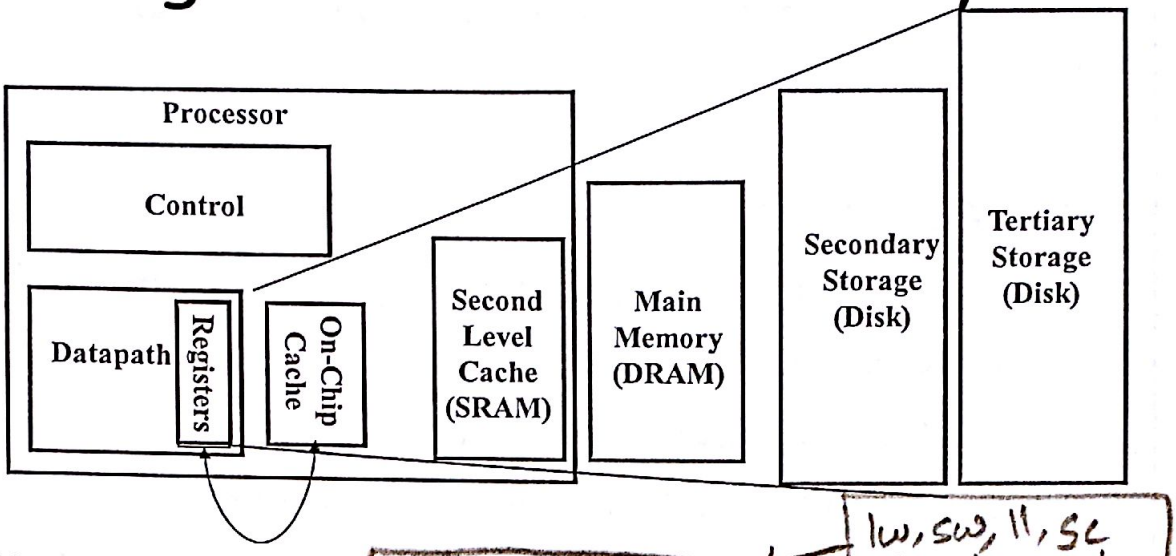
- Several issues:
  - Determine hit/miss: How do we know if a data item is in the cache?
  - If it is, how do we find it?
  - If it isn't, where do I place it?
  - Replacement: What do we do with data that was present?
  - Who manages this? Compiler? Hardware? Software/OS/Programmer?

Managing the memory hierarchy



- Whose responsibility is it?
  - Short answer: it depends on the level

Register<->Main memory

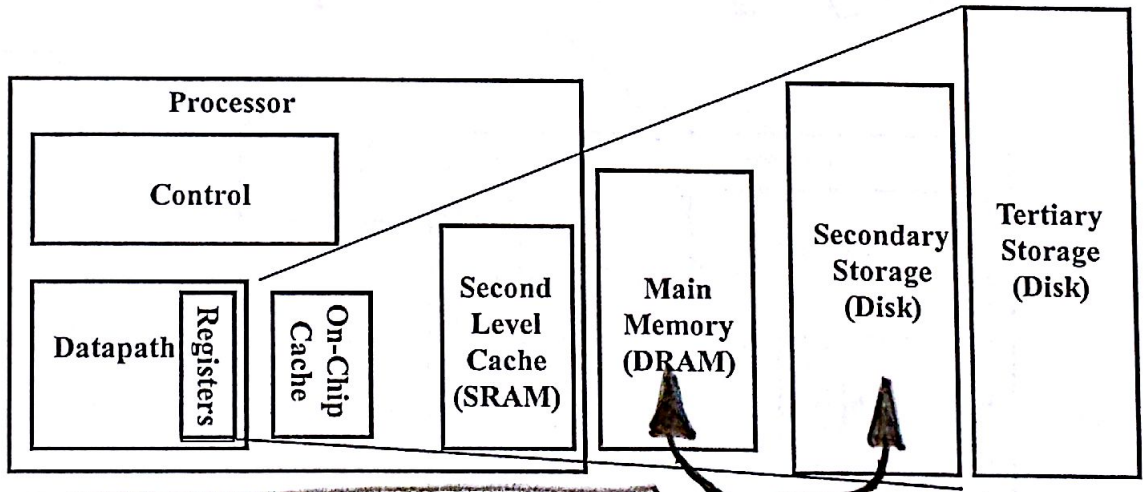


lw, sw, ll, sc given by code.

- Managed explicitly by compiler/programmer
  - "Word" granularity
  - Load/store ties memory locations to registers (allocation)
  - Register temporaries ("spill" to memory when needed)
- Complexity!

granularity: "Word"

Disk<->Memory



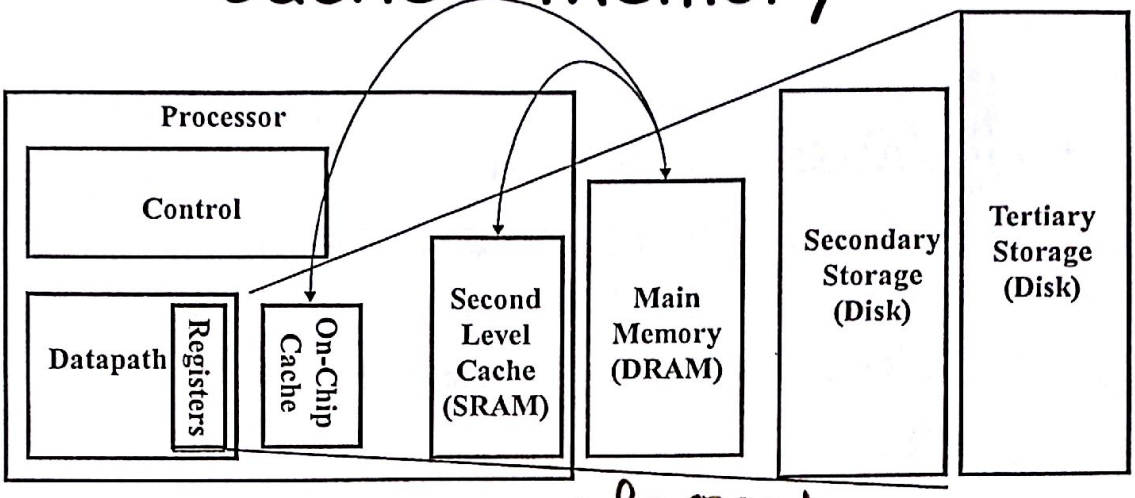
- Programmer: Explicit file read/write
- Disk-block/Page granularity
- OS: Automatic transparent to user
  - Virtual memory
  - Illusion of large memory, protection
  - More later

make system call like fopen

brings pages into RAM

granularity: "Page"

Cache<->Memory



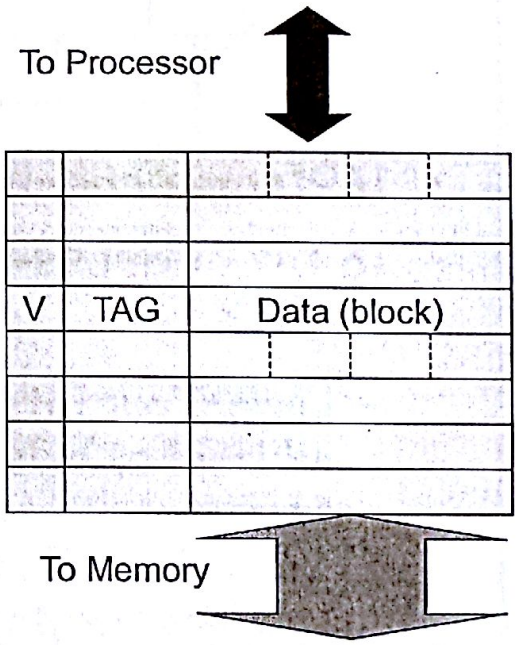
for speed

- Hardware managed: needs to be fast
- Automatic: to avoid complexity of explicit management
- "Block" granularity to exploit spatial locality
- Retain recently accessed blocks to exploit temporal locality

granularity: "Block"

Cache Operation

- Tag, data, valid
- Tag:
  - Mapping larger space (all addresses) to a smaller space (cache)
  - To identify which block (address) is resident
- Data:
  - Block: more than one word
- Valid:
  - Not everything in cache is meaningful
- Frame (block-frame/cache-frame)



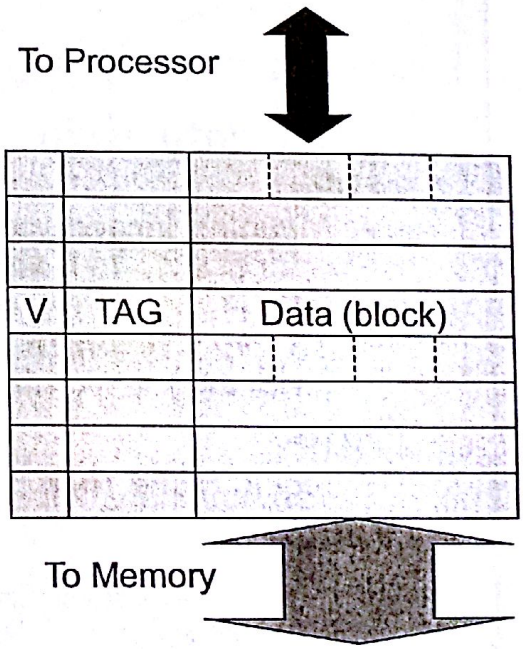
Cache Operation

- Hit/Miss detection
  - If (incoming tag == stored tag)
    - Hit //i.e. block is resident in cache
    - Return word to processor
  - Else
    - Miss
      - Make space : replace some other block
      - Get block from memory
      - Put block in "data" part, set tag using new address tag

if(tagmatch)
return word
else
make space
get memory
return word
to fill space

Example of cache operation

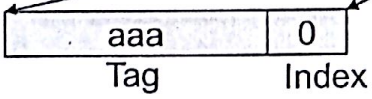
- 16-frame cache
- 16 bit address-space
- Use***:
  - Lower four bits of address as index of frame
  - All other bits of address as tag

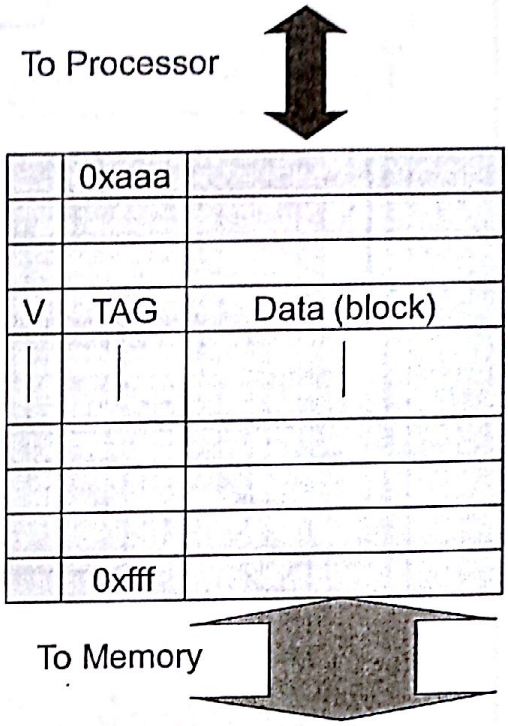


Cache Operation

- Cache operation for the following address-stream
- Assumes one word cacheblocks

0xaa10
0xaaa0
0xffff
0xaaa0
0xffff
0xaaa0





Cache Operation

- Tag comparators
- Hit detection
- How many frames?
- What is the cache size?
- What if block size is more than one word?

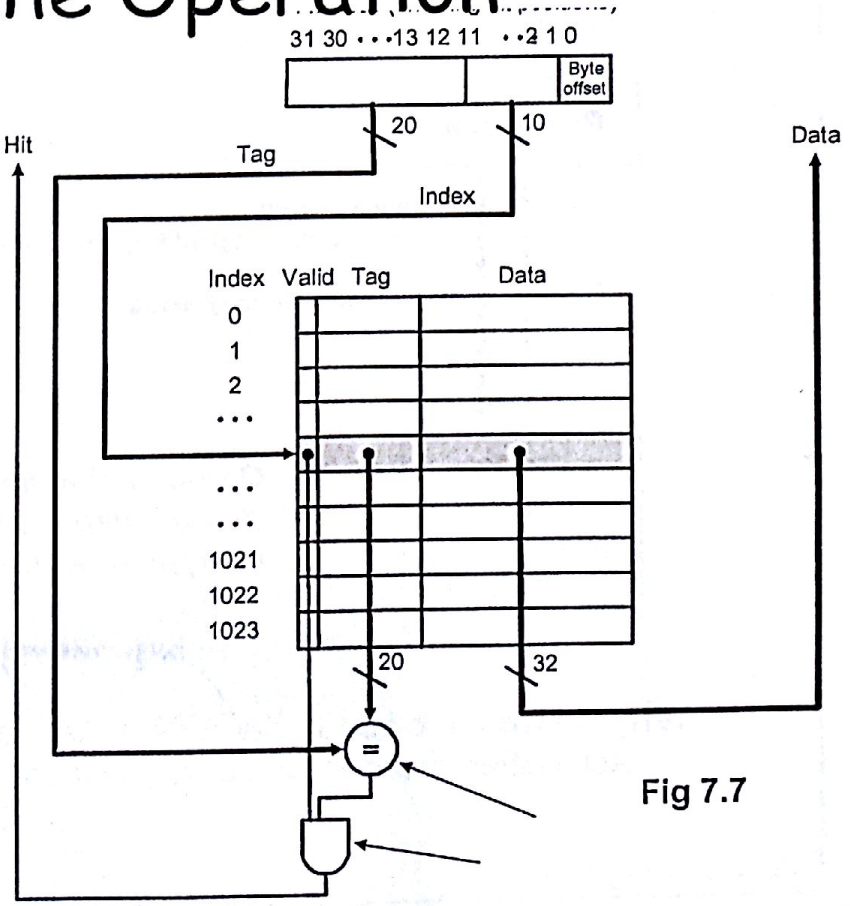
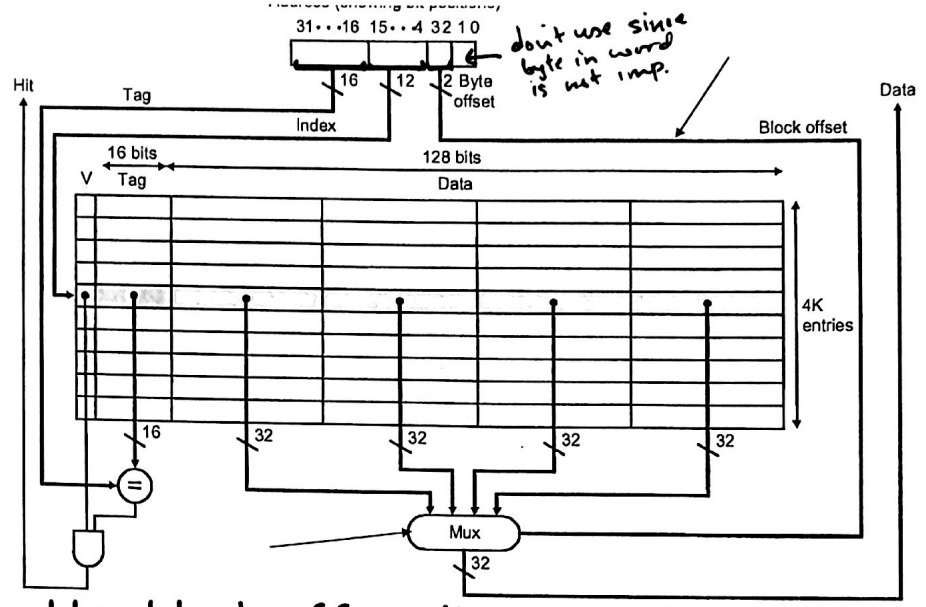


Fig 7.7

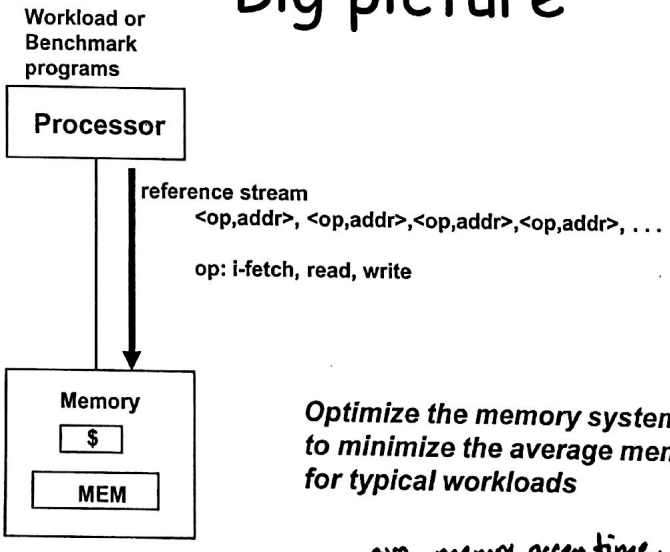
here a block has 4 words in it.

Multi-word Cache Blocks



- Use block-offset lines to select word

Big picture



Optimize the memory system organization to minimize the average memory access time for typical workloads

avg. memory access time.

- Why do we care about AMAT? AMAT affects CPI
  - Remember there is 1.x memory ops per instruction

Impact on Performance

- Suppose a processor executes at
  - Clock Rate = 2 GHz (0.5 ns per cycle)
  - CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control

- Suppose that 5% of memory operations get 100 cycle (50ns) miss penalty

95% memory access goes to cache.

- CPI = ideal CPI + average stalls per instruction
  = 1.1(cyc) +(0.30 (datamops/ins)
  x 0.05 (miss/datamop) x 100 (cycle/miss))
  = 1.1 cycle + 1.5 cycle
  = 2. 6

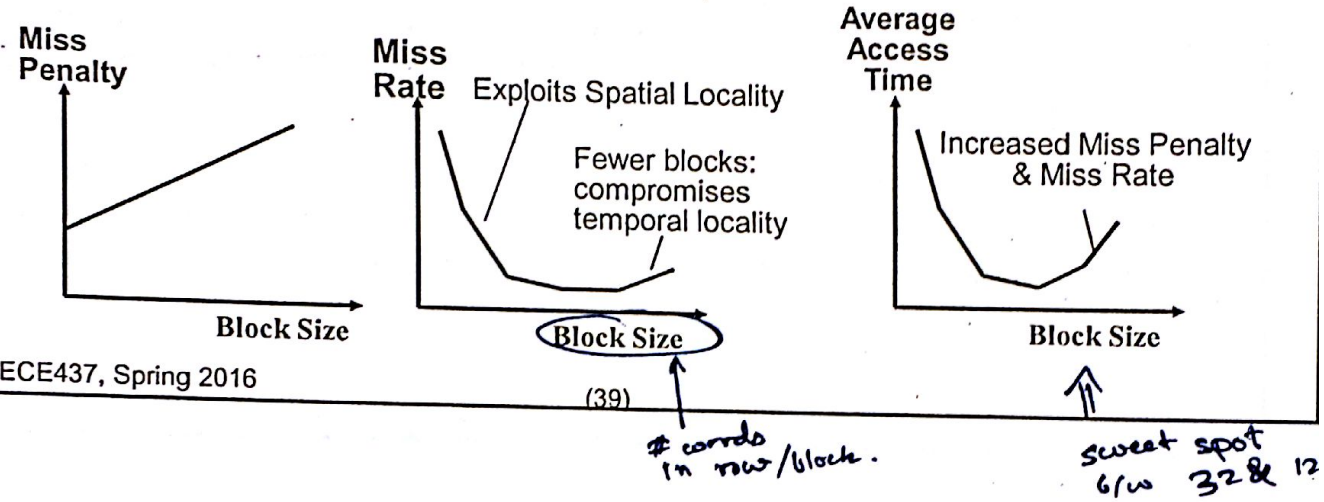
data memory operations

- ~58 % of the time the processor is stalled waiting for memory!

- A 0.5% instruction miss rate would add an additional 0.5 cycles to the CPI

Block Size Tradeoff

- In general, larger block size take advantage of spatial locality BUT:
  - Larger block size means larger miss penalty:
    - Takes longer time to fill up the block
  - If block size is too big relative to cache size, miss rate will go up
    - Too few cache blocks
- In general, Average Access Time:
  - = Hit Time + Miss Penalty x Miss Rate



Block Size

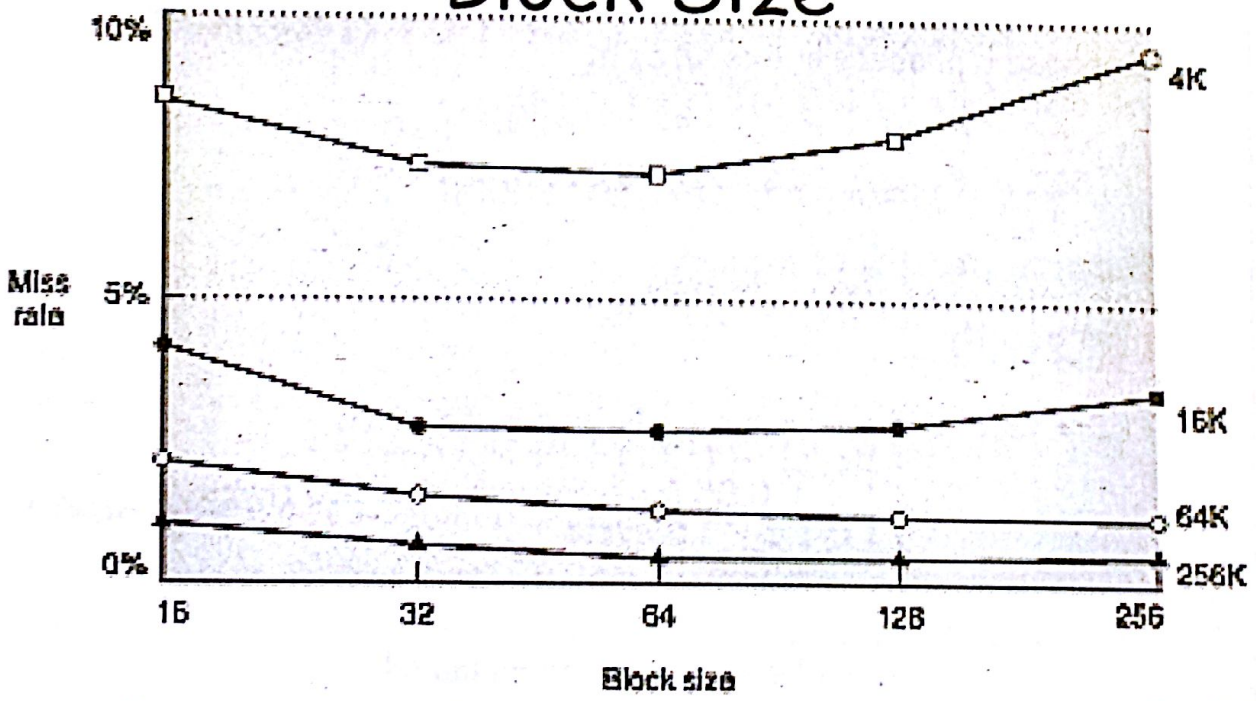


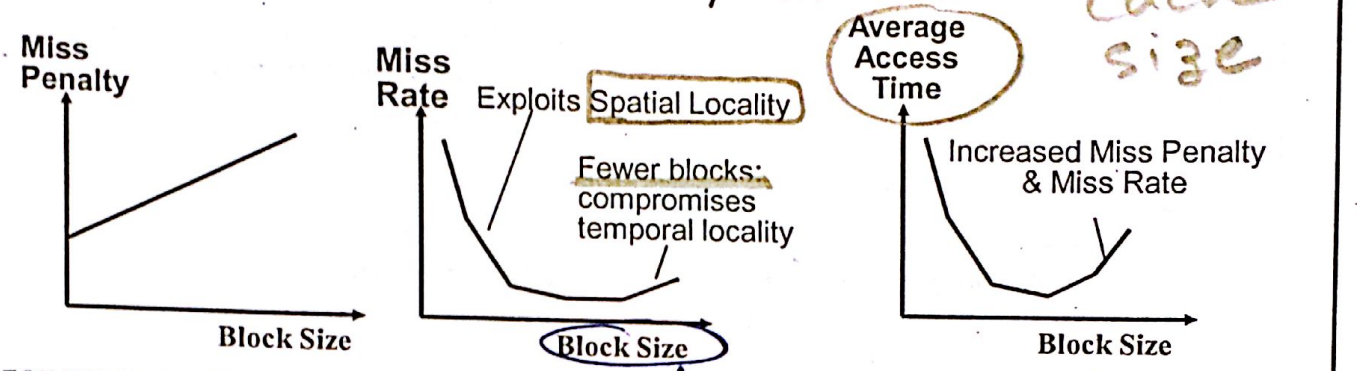
Fig 7.7

- Measurements from real programs
- Bottomline: Block size chosen by experiment, typically 16-128 bytes

Block Size Tradeoff

- In general, larger block size take advantage of spatial locality BUT:
  - Larger block size means larger miss penalty:
    - Takes longer time to fill up the block
  - If block size is too big relative to cache size, miss rate will go up
    - Too few cache blocks
- In general, Average Access Time:
  - = Hit Time + Miss Penalty x Miss Rate

assume constant cache size



# of words in row/block.

sweet spot b/w 32 & 128

Block Size

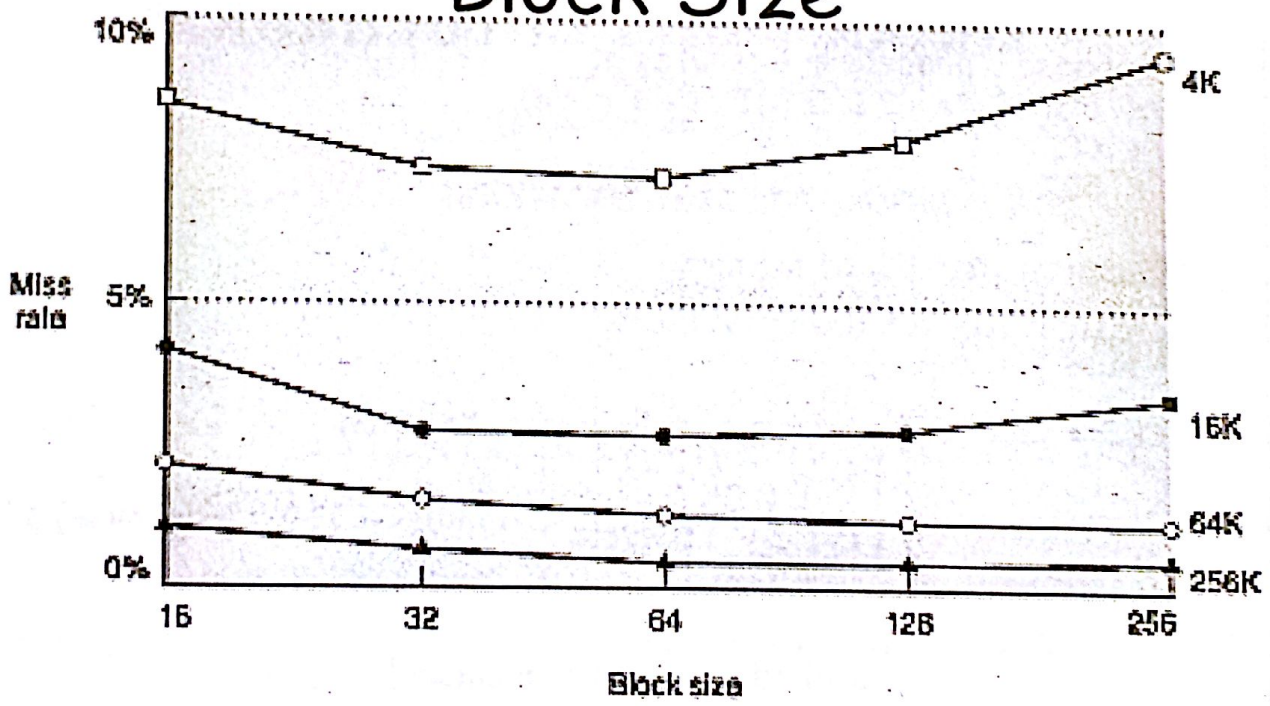


Fig 7.7

- Measurements from real programs
- Bottomline: Block size chosen by experiment, typically 16-128 bytes

→ 4 words – 32 words

Checkpoint

- Summary:
  - Cache management in hardware
  - Caches terminology and organization
    - Frames
    - Blocks
    - Tags
  - Example of Cache operation
- Next: 4 questions
  - Where is a block placed?
  - How is a block found?
  - Which block is replaced?
  - What happens on a write?

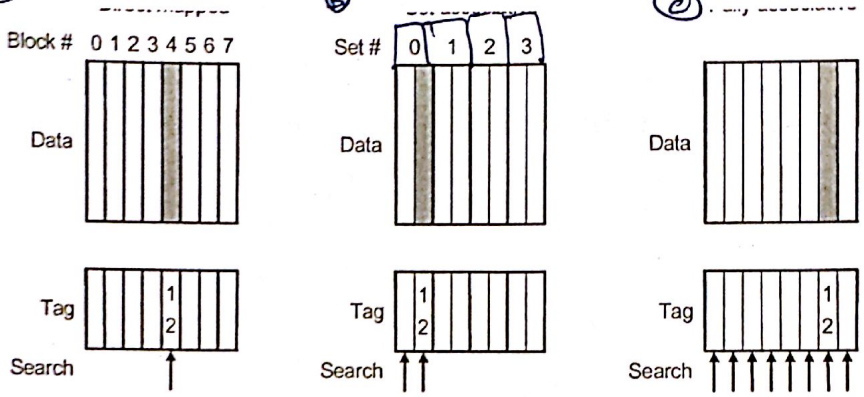
4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level? *(Block placement)*
- Q2: How is a block found if it is in the upper level?
  (Block identification)
- Q3: Which block should be replaced on a miss?
  (Block replacement)
- Q4: What happens on a write?
  (Write strategy)

w/o A & B, a scenario where B is better is if we access a no. that falls in same place, eg. ~~After~~ 12 & 4. In (A), there will always conflict & give 100% miss. In (B), both stored, so no miss after first read.

Q1. Block Placement (ways)

(A) (B) (C)



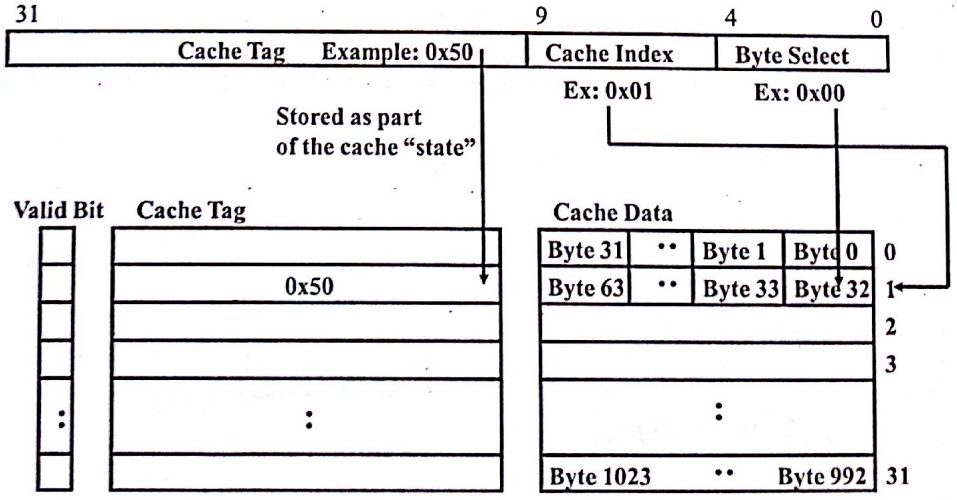
- In previous example:
  - Block may reside in one fixed frame. (frame[addr mod 16])
- Other points in the design space
- Fully-associative
  - Block a reside in any frame
- N-way set-associative
  - Block may reside in a set of N-frames

eg. (B) is a 2 way set associative cache.

[32 - (N-way +1)] bits are tag bits

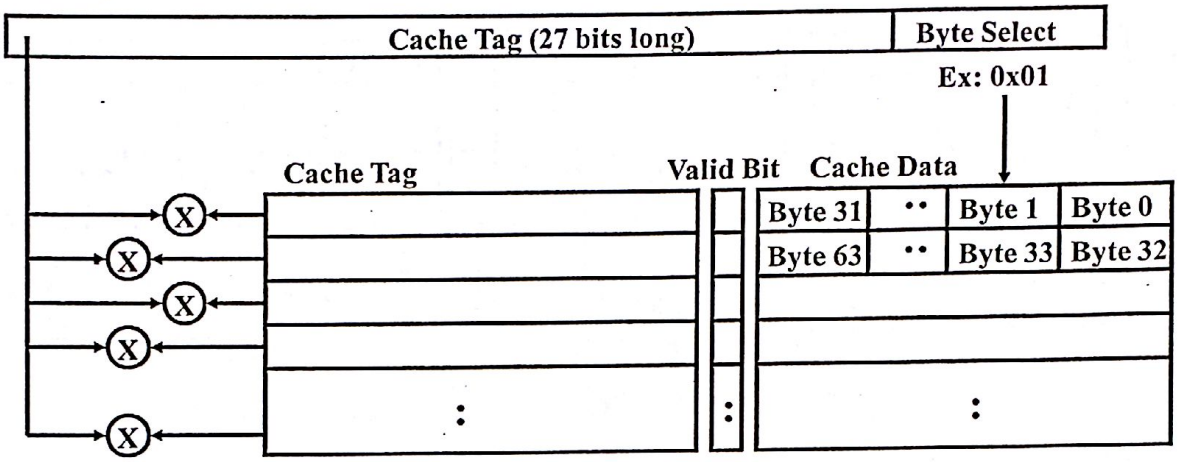
Example: 1 KB Direct Mapped Cache with 32 B Blocks

- For a 2 ** N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2 ** M)



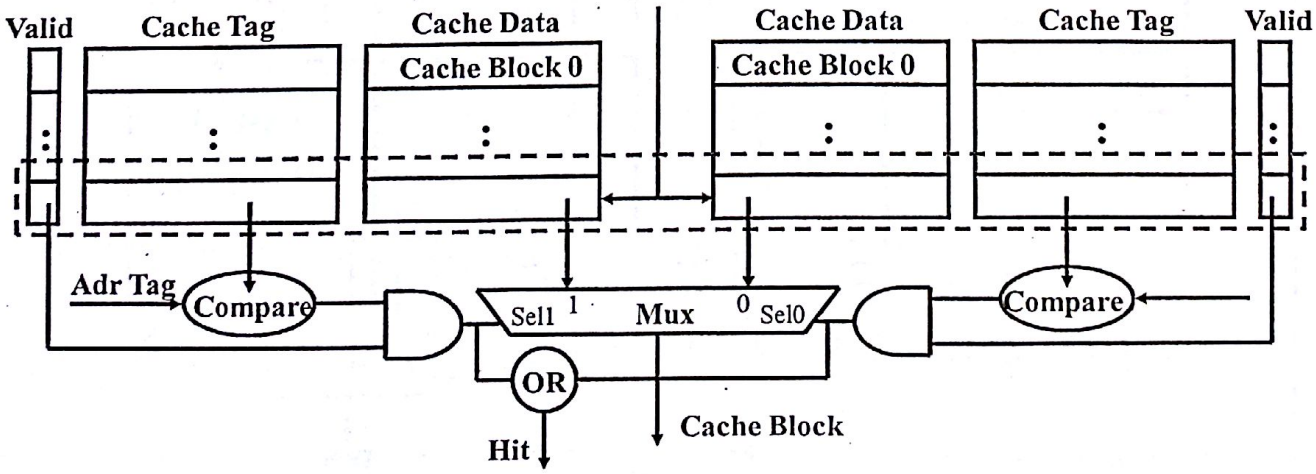
Another Extreme Example: Fully Associative

- Fully Associative Cache
  - Forget about the Cache Index
  - Compare the Cache Tags of all cache entries in parallel
  - Example: Block Size = 32 B blocks, we need N 27-bit comparators
- By definition: Conflict Miss = 0 for a fully associative cache



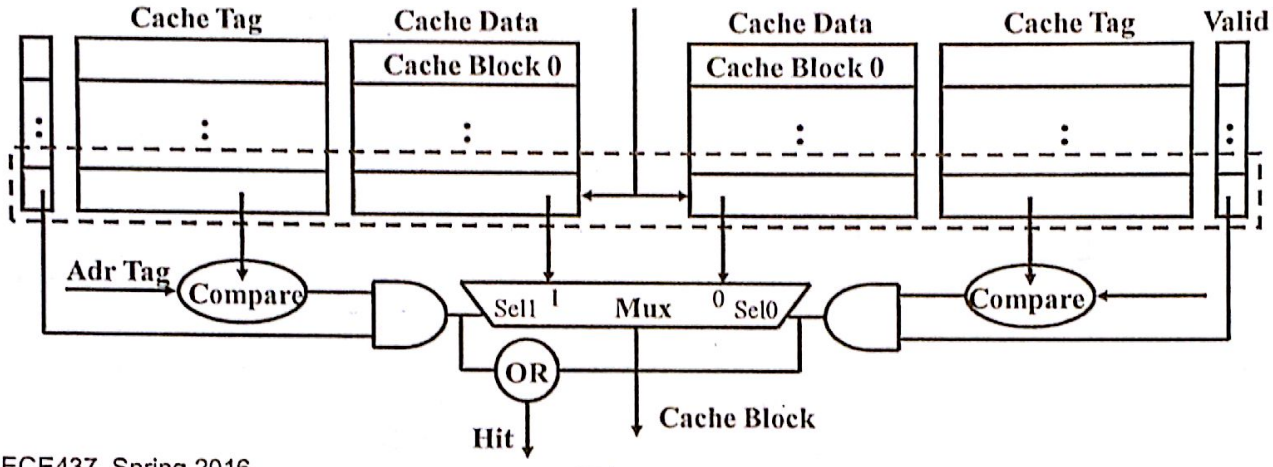
A Two-way Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Cache Index selects a "set" from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result

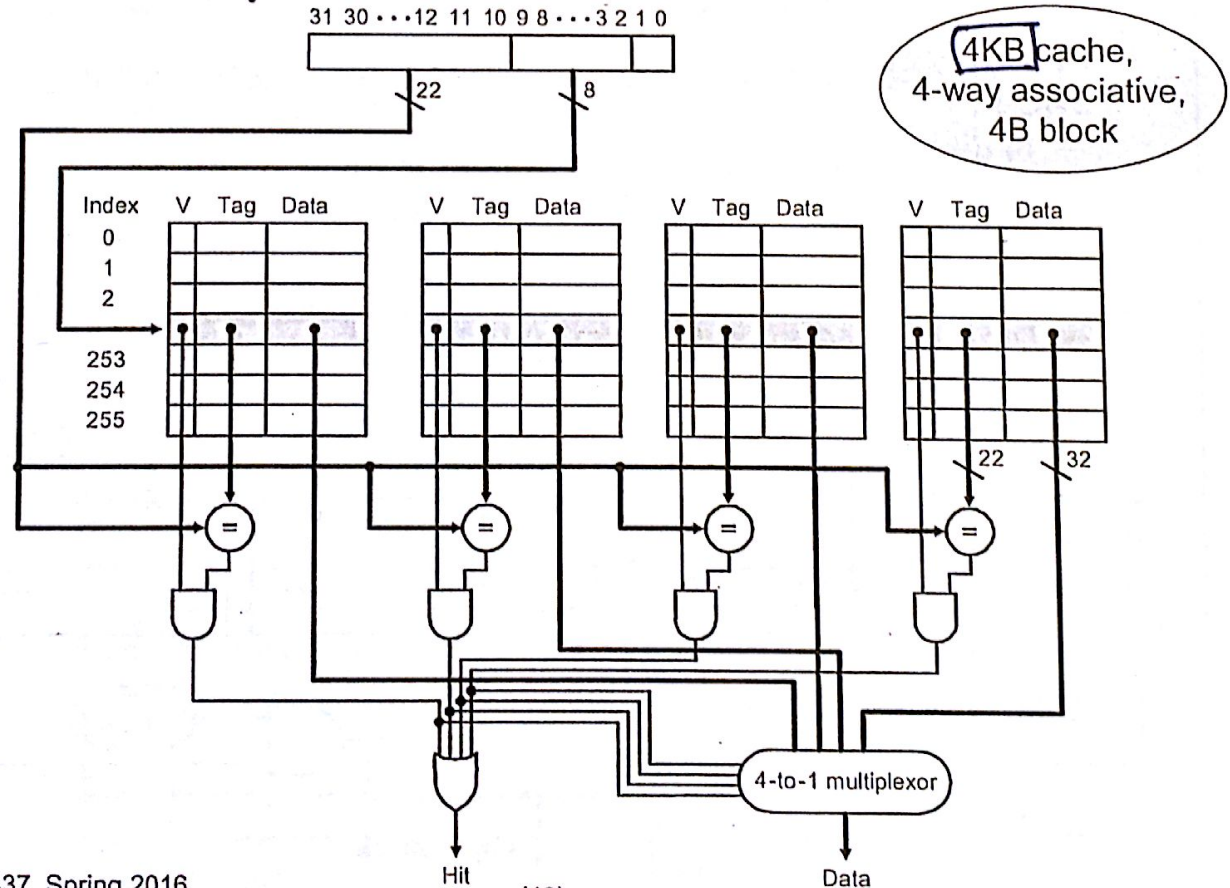


Disadvantage of Set Associative Cache

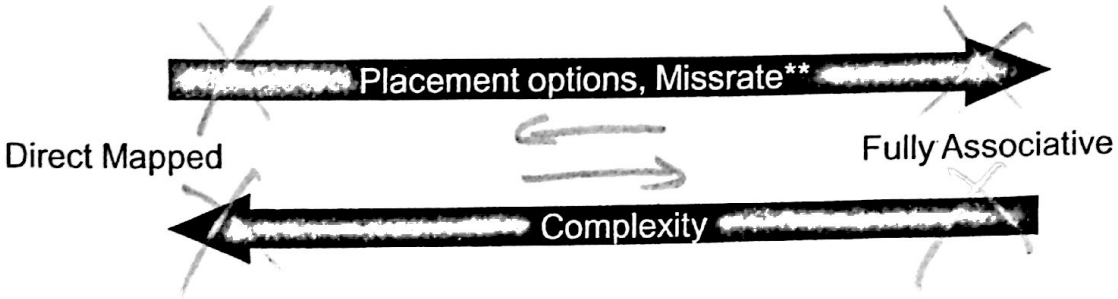
- N-way Set Associative Cache versus Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss decision and set selection
- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue. Recover later if miss.



4-way set associative cache

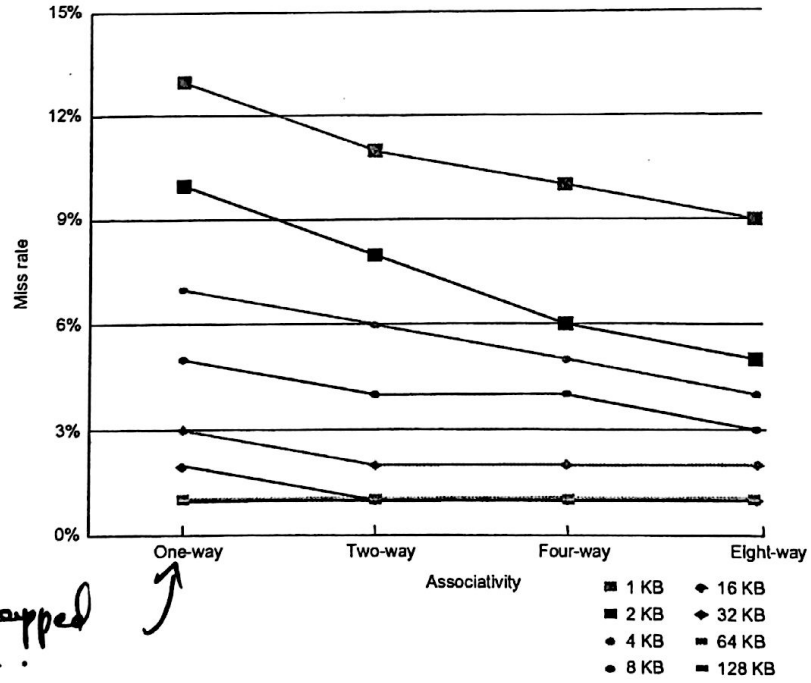


Associativity spectrum



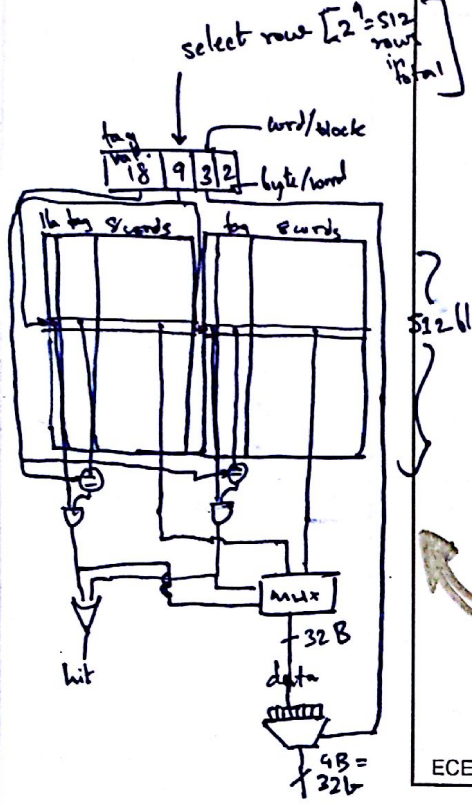
- Conflict misses reduced with higher associativity
- Associative search is complex, suited for smaller caches
- Increasing associativity:
  - Increases tag bits, shrinks index bits
  - Increases comparator size (~ tag bits)

Performance



direct mapped cache!

- A little associativity goes a long way



Organization Methodology

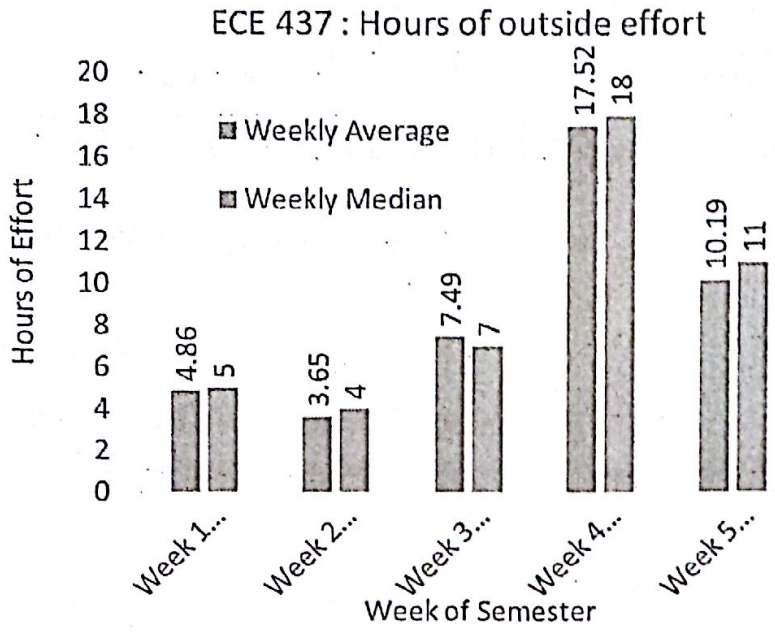
- How to determine:
  - Number of bits for
    - Index, tag and block offset
- Walkthrough example(s)
  - 32KB, 32B block, 2-way associative cache

Cache Organization

- Cache size = 32 KB (CS)
- Block size = 32 B (BS)
  - Frames (F) = CS/BS = 1024 (= 1K)
- Associativity = 2-way (A)
  - Number of frames/way = F/A = 512

- Address-bits = 32 bits (Ad)
  - Block-offset bits (b) = lg(BS) = lg(32) = 5
  - Index bits (i) = lg(FpW) = lg(512) = 9
  - Tag bits (t) = Ad - i - b = 32 - 9 - 5 = 18

Survey Results



Cache Block Diagrams

- 96KB, 3-way set associative, 64Byte block cache
- Direct-mapped, 16KB, 128 byte block cache

March 3rd

In fully associative cache, conflict miss = 0 (always). Hence, a miss must be of another kind.

Miss Classification

- Compulsory (cold start or process migration, first reference): first access to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: If you are going to run "billions" of instruction, Compulsory Misses are insignificant
- Conflict (collision): ∵ not enough set associativity
  - Multiple memory locations mapped to the same cache location
  - → Solution 1: increase cache size
  - → Solution 2: increase associativity
- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- Invalidation: other process (e.g., I/O) updates memory

ECE437, Spring 2016 (59)

Can solve by prefetching values.
Can also solve by increasing block size.
eg. Cache is 64kb size but program runs 16 kb serial again & again. Here program has locality but cache not adequate

Source of Cache Misses Quiz

	Direct Mapped	N-way Set Associative	Fully Associative
Cache Size: Small, Medium, Big?	big	medium	small
Compulsory Miss:	same	same	same
Conflict Miss	high	med	zero (0)
Capacity Miss	low	med	high
Invalidation Miss	same	same	same

Choices: Zero, Low, Medium, High, Same

multicore

cache size
If all has same capacity, then these could also be same

ECE437, Spring 2016 (60)

Q3. Block Replacement

- Q3: Which block should be replaced on a miss? *(Block replacement)*
  - Easy for Direct Mapped
  - Set Associative or Fully Associative:
    - Random
    - FIFO (Not a good idea, in general)
    - LRU (Least Recently Used)
    - NRU (Not most Recently Used)

Smart
Not Stupid

OPT (Ideal)
LRU
NRU
Simple heuristics capture large fraction of opportunity. Sweat for the last 10%

ECE437, Spring 2016 (62)

look at freq of usage. least freq. one is replaced.
track most freq used. Replace any of the others.
P(replace 2nd most freq) = 1/#slots
still good.

Exercise

- Give an example of an address stream where
  - 2-way associative cache is better than direct-mapped cache
  - Direct mapped cache is better than 2-way cache.
- Use 16 entry caches, assume LRU replacement

v. imp. Solve once atleast.

ECE437, Spring 2016 (63)

write immediately to memory. Allows throwing away of a block at any time ∵ memory is up to date.

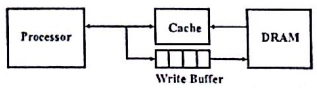
Q4. Write strategy

- Q4: What happens on a write? *(Write strategy)*
- *Write through*—The information is written to both the block in the cache and to the block in the lower-level memory.
- *Write back*—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each?
  - WT: read misses cannot result in writes
  - WB: no writes of repeated writes
- WT always combined with write buffers so that don't wait for lower level memory

ECE437, Spring 2016 (64)

keeps data in cache. It checks if addr is clean or dirty. If dirty, memory to be updated.

Write Buffer for Write Through



- A Write Buffer is needed between the Cache and Memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4-8
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
  - Can tolerate bursty behavior
- Memory system designer's nightmare:
  - Store frequency (w.r.t. time) -> 1 / DRAM write cycle
  - Write buffer saturation

ECE437, Spring 2016 (65)

write buffer ↓
here data goes to WB from cache. The WB slowly writes into DRAM.
Issue: if too many sw from processor against DRAM write, then will have saturation problem

this again means no space for reads.

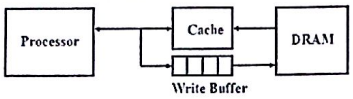
LRU implem.
each index has a rank from 0 to n-1.
when touching an address, that index's rank goes to 0 and all ranks before it increase by 1.

eg		
0	1	2
1	(1)	0
2	3	3
3	4	4
(4)	0	1
5	5	5
6	6	6
7	7	7

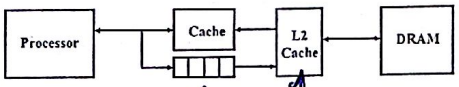
write no allocate

If write miss, instead of getting block into cache & writing to it, can just write to memory. Mostly, it will be final write. If it is a hot block, v. likely that it will be read. The read miss will get the block in cache anyways.

Write Buffer Saturation



- Store frequency (w.r.t. time) > 1 / DRAM write cycle
  - If this condition exists for a long period of time (CPU cycle time too quick and/or too many store instructions in a row):
    - Store buffer will overflow no matter how big you make it
    - CPU Cycle Time <= DRAM Write Cycle Time
- Solution for write buffer saturation:
  - Use a write back cache
  - Install a second level (L2) cache:



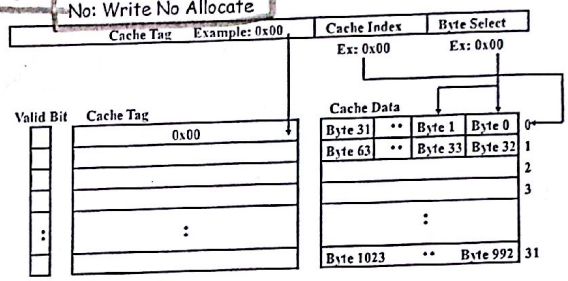
write through model

write back model

Write-miss Policy: Write Allocate versus Not Allocate

- Assume: a 16-bit write to memory location 0x0 and causes a miss
  - Do we read in the block?
    - Yes: Write Allocate
    - No: Write No Allocate

or write miss, element also put in cache.



AMAT Impact quiz

	Cache hit time	Miss penalty	Missrate
Cache Size: Small, Medium, Big?	↑	–	↓
Associativity :Low-to-high	↑ (↑ search)	–	↓
Block size: Low to high	– (small ↑)	↑ (have more to bring)	↓*
Replacement Policy FIFO to LRU	–	–	↓*

Conflict miss reduces, compulsory miss reduces.

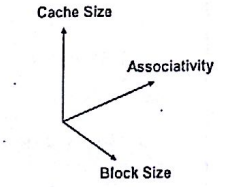
- Average Memory Access time
- AMAT = hit time + miss-rate * miss-penalty
- Choices: same, increasing or decreasing
- Ignore last row for now

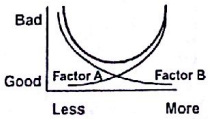
Improving Cache Performance

- Reduce Hit time
  - small and simple-> direct mapped
- Reduce miss rate
  - Large cache, large blocksize, associative,
- Reduce miss penalty
  - Reduce block-size
- Remember Amdahl's law
  - Common case : hit
  - Reduce miss-rate at the cost of hit time

Cache design space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins





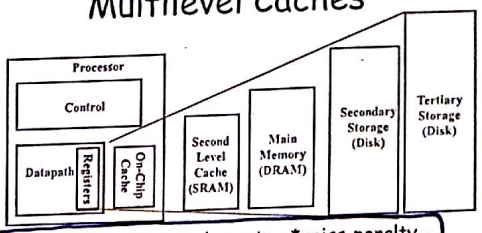
Practical design issues

- Multi-level Caches
- Split Cache vs. unified cache

Practical design issues

- Multi-level Caches
- Split Cache vs. unified cache

Multilevel Caches



- $AMAT_{L1}$ = hit $time_{L1}$ + $miss\text{-}rate_{L1}$ * $miss\text{-}penalty_{L1}$
- What is $miss\text{-}penalty_{L1}$?
  - Access time of memory
- Put in a large L2 cache between L1 and memory
  - What is the $miss\text{-}penalty_{L1}$?
  - $AMAT_{L2}$ = hit $time_{L2}$ + $miss\text{-}rate_{L2}$ * $miss\text{-}penalty_{L2}$

adding L2 reduces miss penalty of L1. It is equal to $AMAT_{L2}$.

Multilevel Caches

- Cycle time = 0.5ns (~ 2GHz clock)
- Main memory access = 50ns = 100 cycles
- L1 miss rate = 5%
- Without 2nd level cache
  - $AMAT_{L1}$ = 1 + 5% * 100 = 6 cycles
- With 2nd level cache
  - L2 miss-rate = 2% (local miss-rate)
  - L2 hit time = 10 cycles ← this means it can be a big cache since 10 clocks used to get/set data.
  - $AMAT_{L2}$ = 10 + 2% * 100 = 12 cycles
  - $AMAT_{L1}$ = 1 + 5% * 12 = 1.6

AMAT Impact: Answers

	Cache hit time	Miss penalty	Missrate
Cache Size: Small, Medium, Big?	inc	same	dec
Associativity :Low-to-high	inc	same	de
Block size: Low to high	same	inc	dec**
Replacement Policy FIFO to LRU	same	same	dec*
Multi-level Caches	same	dec*	same

having L2 will help reduce miss penalty. [look at eg. in prev slide].

Split caches

- One for instruction, one for data
- Split cache
  - Instructions account for 75% of mem accesses
  - I-missrate = 5%, D-missrate = 6%
  - AMAT = (1 + 0.05*10)*0.75 + (1 + 0.06*10) * 0.25,
  - = 1.525
- Unified Cache
  - Aggregate missrate = 4%
  - AMAT = (1 + 0.04*10) = 1.4???
  - For modern pipelined processor:
    - single-memory structural hazard

↑ this is worse, but allows avoiding of pipeline hazards.

State of the art

- 2-3 levels of SRAM cache
- Split I- and D-caches at Level 1
- Direct mapped (or low associativity) at Level 1
- Higher associativity at subsequent levels

Why can unified be good?
↑ No need to hard divide Instr & data to 25% for eg.

Today, L1 split to prevent pipeline hazards. L2 & beyond, we use unified for DYNAMIC MEMORY SHARING.

Summary

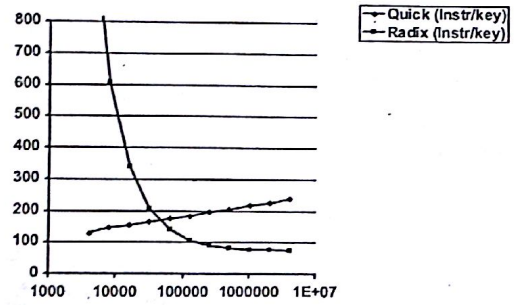
- Memory technology (Capacity/cost/speed)
- Need for hierarchy
- Performance
  - AMAT, ideal vs. real CPI
- Cache management:
  - Associativity, indexing, write handling, multi-word blocks etc.
- Diagrams of arbitrary cache organizations
- Next:
  - Cache-friendly coding techniques
  - Virtual Memory

Software interaction

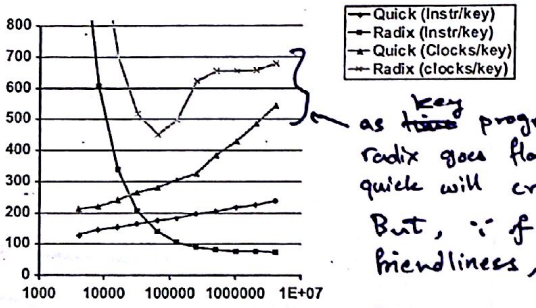
- RAM model of computation
  - All memory accesses take the same amount of time
  - Theoretical Model – has nothing to do with DRAM
- Reality:
  - Caches introduce non-uniformity
  - Hits take less time than misses
- Quicksort
  - fastest comparison based sorting algorithm when all keys fit in memory: $\Theta(n \lg(n))$
- Radixsort
  - also called "linear time" sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys: $\Theta(n)$

For n v big 'n', caches don't really help.

QS vs. RS : Instructions



QS vs. RS : Time, Instructions

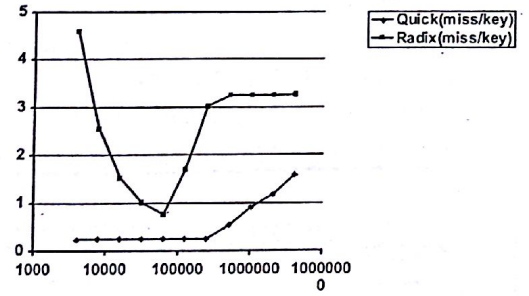


as ~~time~~ key progress, radix goes flat and quick will cross & rise.

But, if cache friendliness, quick is better.

Radix is cache unfriendly since it must pass through all keys again & again for each digit.

QS vs. RS : Cache misses



- RAM model results are still valid... but at much larger input sizes
- How does one create practical, fast algorithms?
- Cache-aware programming/compilation

Exercise 1

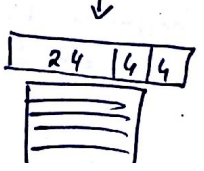
- What is the miss-rate of the following code segment for
  - 16 frame, direct mapped cache with 4-word cache blocks?
  - 16-frame, 2-way set-associative cache with 4-word cache blocks?
- Assume cold start (i.e. no valid data in the cache to begin with)
- Assume the following start addresses (bytes) for the three arrays
  - A : 0xC000
  - B : 0xC100
  - C : 0xC200
- Ignore writes for now

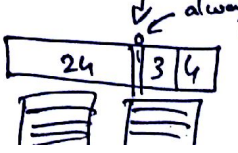
contiguous blocks.

each array 256 bytes

```
for(i=0;i<64;i++){
    C[i] = min( A[i] , B[i] );
}
for(i=0;i<64;i++){
    D[i] = max( A[i] , B[i] );
}
```

What if start addresses are:
A : C000
B : D110
C : E220
Instead?





always zero in this expt.

In Direct mapping here B kicks out A & so on. Merging / Fusing loops gives no adv. Original 2 loop also no adv. 100 % miss rate.

Worksheet 1

Cache 1: T=8 | I=4 | BO=4

Cache 2: T=9 | I=3 | BO=4

index

Address Stream	Address Stream 2	Cache 1	Cache 2
C000 (A[0])		Miss	Miss
C100 (B[0])		Miss	Miss
C004 (A[1])			
C104 (B[1])			

Cache-aware programming

- Instruction Sequencing
  - *Loop Interchange*: change nesting of loops to access data in order stored in memory
  - *Loop Fusion*: Combine 2 independent loops that have same looping and some variables overlap
  - *Blocking*: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down entire columns or rows
- Data Layout
  - *Merging Arrays*: Improve spatial locality by single array of compound elements vs. 2 separate arrays
  - *Nonlinear Array Layout*: Mapping 2 dimensional arrays to the linear address space
  - *Pointer-based Data Structures*: node-allocation
- Example walkthrough: Loop fusion, Blocking, Merging Arrays

Worksheet for example

	Base	Loop Fusion	Array Merging	Tiling
Direct Mapped				
2-way Set associative				

- Count read misses

Miss categorization

- 64+64 = 128 words
- Each miss brings in 4 words
  - Minimum of 32 (=128/4) misses
  - Cannot do better
- Exercise: Identify conflict and capacity misses

Walk through Exercise 1

- Count read misses for the following code segment for
  - 16 entry, direct mapped cache with 4-word cache blocks?
  - 16-entry, 2-way set-associative cache with 4-word cache blocks?
- Assume cold start (i.e. no valid data in the cache to begin with)
- Assume the following start addresses for the three arrays
  - A : 0xC000
  - B : 0xC100
  - C : 0xC200
  - D : 0xC300
- Ignore writes for now
- USE WORKSHEET

```
for(i=0;i<64;i++) {
    C[i] = min( A[i] , B[i] );
}
for(i=0;i<64;i++) {
    D[i] = max( A[i] , B[i] );
}
```


#1 : Loop Fusion

- Coverts distant reuse to near reuse
- Enhances temporal locality
- Code Transformation

```
for(i=0;i<64;i++) {
    C[i] = min( A[i] , B[i] );
}
for(i=0;i<64;i++) {
    D[i] = max( A[i], B[i] );
}
```

```
for(i=0;i<64;i++) {
    C[i] = min( A[i] , B[i] );
    D[i] = max( A[i], B[i] );
}
```

#2: Array merging

- Eliminates conflicts
  - Array of compound structure vs.
  - multiple arrays of simple data
- Enhances spatial and temporal locality
- Data layout transformation

```
for(i=0;i<64;i++) {
    C[i] = min( A[i] , B[i] );
}
for(i=0;i<64;i++) {
    D[i] = max( A[i], B[i] );
}
```

```
Struct merge {
   int A;
   int B;
};
Struct merge M[64];
for(i=0;i<64;i++) {
   C[i] = min( M[i].A , M[i].B);
}
for(i=0;i<64;i++) {
   D[i] = max( M[i].A, M[i].B);
}
```


#3: Blocking (Tiling)

- Exploits re-use across loops
  - Divide into pieces that fit in the cache vs.
  - Marching through whole array
- Capacity misses
- Code Transformation

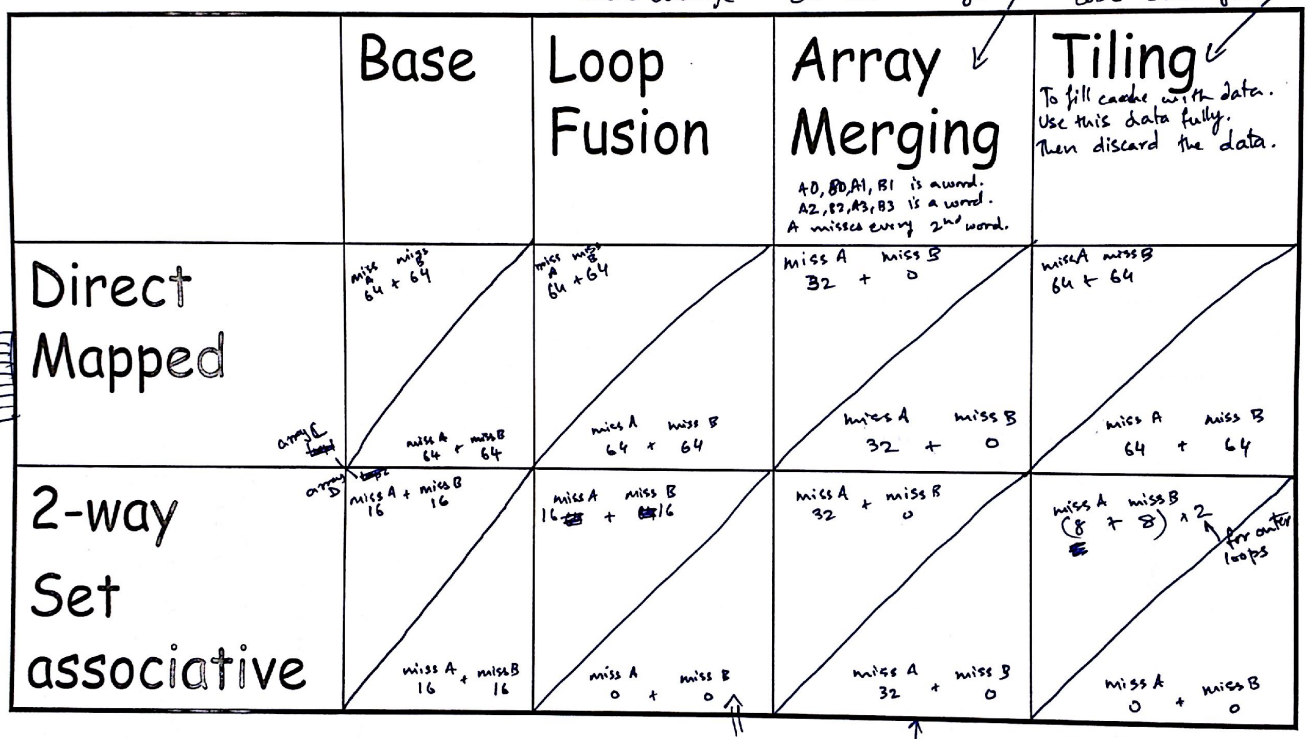
```
for(i=0;i<64;i++) {
    C[i] = min( A[i] , B[i] );
}
for(i=0;i<64;i++) {
    D[i] = max( A[i], B[i] );
}
```

```
for (j=0; j<2;j++)
{
 for(i=0;i<32;i++) {
    C[32*j + i] = min( A[32*j + i] , B[32*j + i] );
 }
 for(i=0;i<32;i++) {
    D[32*j + i] = max( A[32*j+ i], B[32*j + i] );
 }
}
```


State of the practice

- Cache friendly programming challenges
  - No global view of application
  - Different cache sizes
- Analyze programs after they're written
  - Find bad access patterns
  - Fix them
  - Lather, Rinse and Repeat

Worksheet for example



- Count read misses

ECE437: Introduction to Digital Computer Design

Mithuna Thottethodi
Chapter 7 (texbook + CD)

Spring 2016

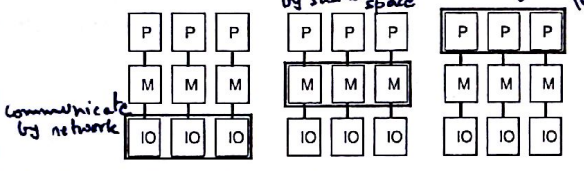
Outline

- Multiprocessors
  - Why?
  - Power changes everything
  - What about performance
    - Two processors of speed X instead of one processor of speed 2X
    - Challenges in harnessing parallelism
  - Is there a single vendor with uniprocessor products? (non-embedded)
- Key concepts
  - Programming models
  - Cache coherence
  - Synchronization
  - Consistency

Various Programming Views

communicate by shared addr space

communicate on processor level.



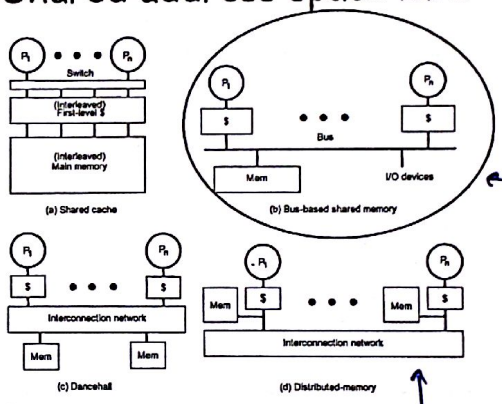
Join At:	I/O (Network)	Memory	Processor
Program With:	Message Passing	Shared Memory	(Dataflow/Systolic), Single-Instruction Multiple-Data (SIMD)

- Our focus: Shared Address Space
  - Most representative design point for multicores

==> Data Parallel

Shared address space MPs

hits are slower: switch ↑ critical path.



$ cache
P processor

← in our comps

bus vs general network.
↑ only 1 thing at a time
↑ 2+ ppl can operate on network at same time

v. high perf. better than dancehall :: No need to always go across interconnect network

AMD and Intel quad-core designs

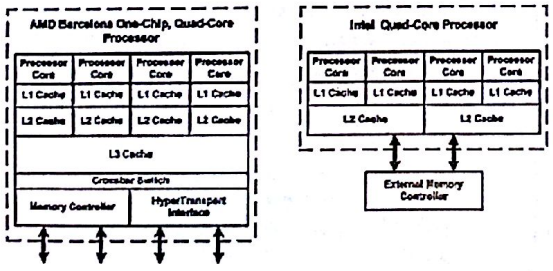


Figure © Embedded.com

Process vs. Thread

- "Heavyweight" processes
  - Different Thread of control
    - PC, registers, stack It has its own: values
  - Different Address space (page table, heap)
- "Lightweight" processes, a.k.a. "threads"
  - Different PC, register values, and stack allocation
    - "Context"

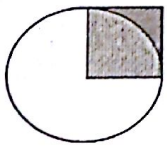
  Same address space (thus same page table, same heap)
- Shared regions across heavyweight processes possible.

How do threads communicate?

- Our focus:
  - Shared memory, bus-based systems
    - Shared address space
    - Memory does not have to physically shared
  - Communication via loads and stores

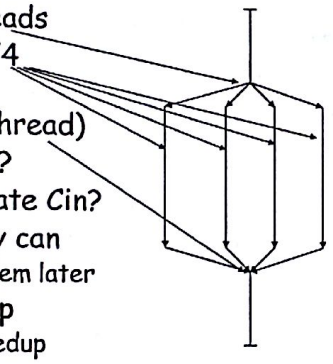
Writing Parallel Programs

- Example: Monte Carlo simulation - Embarrassingly parallel
- Determine value of Pi (π)
- Area of square = 1x1 = 1
- Area of $\frac{1}{4}$ circle = $(\pi r^2)/4 = \pi/4$
- Algorithm:
  - Pick a random point in the unit square
  - If inside quarter circle, Cin++
  - Repeat N times (large N)
- probability of randomly picked point being inside the circle
  - From first principles: $(\pi/4)/1 = \pi/4$
  - From experiment = Cin/N
    $\pi = 4*Cin/N$



Writing parallel programs

- Create "P" (P=4) threads
- Each thread does N/4 experiments
- "Join" (Back to one thread)
- What happens to Cin?
- Can each thread update Cin?
- For now, assume they can
  - synchronization problem later
- Almost linear speedup
  - P processors => P speedup



Merge-sort

- Basic Sequential (Recursive) algorithm
  - Recursively Mergesort top-half and bottom-half of data
    - Terminate at single element
  - Merge the two halves
- Parallel algorithm (Assume fixed two-way parallelism)
  - One thread does Sequential merge-sort of top-half
  - Another thread does the same with bottom half
  - Merge on one thread. (Must wait till both threads are done.)

Parallel merge

- Previous version did merging on one thread
- Possible to split merge work across two threads.
  - The first thread merges from the top till it has N/2 elements
  - The second thread merges from the bottom till it has N/2 elements.
- Done.

Common Parallelization Strategies

- Think of parallelism at the algorithm level
- Possibly different
  - Best parallel algorithm
  - Best parallelization of best sequential algorithm
- Think of partitioning
  - Tasks? (do webpage serving and data-base lookup for different requests in parallel.)
  - Data? (Do similar operations on different data in parallel.)

Programming models

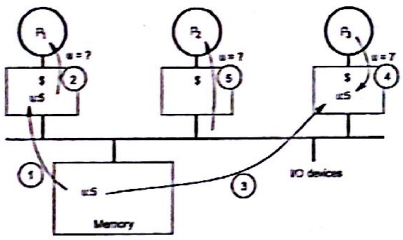
- Pthreads – Shared memory processors **
- MPI – clusters **
- OpenMP – Shared memory
  - Marking loops as parallel loops
- Higher level primitives
  - Streaming, MapReduce, Parallel Recursion (Cilk C)
  - Customized to architecture
    - CUDA for nVidia GPGPUs

Several Interesting Problems

- Does everything work intuitively***
  - No.
  - Reason about coherence/consistency
- Sequencing
  - Synchronization
- After the break: Cache Coherence

Example Cache Coherence Problem

(#) seq^n of events



- Is there a problem?
  - Replication + writes = coherence problem

P_2 wants u = 7 but memory has u = 5.
Also P_1 has u = 5, a stale value.

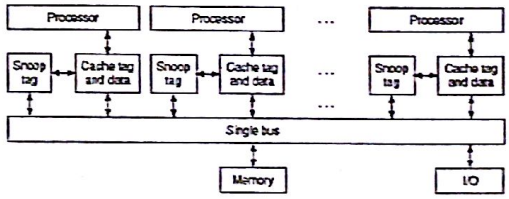
Coherence

- Fuzzy idea
  - Necessary for correctness
- Precise definition
- For any memory location X
  - Operations issued by any process occur in the order in which they issued
  - Value returned by read is the value written by last write
- Derivative properties
  - Write propagation: writes become visible to everyone (when?)
  - Write serialization: writes to a single location seen in same order by every processor

How to enforce coherence

- Correct actions:
  - On write, invalidate other copies
  - On write, update other copies
- Snooping:
  - Every node can "see" the bus
  - If all transactions can be observed
    - Each can maintain coherence

Snoopy coherence

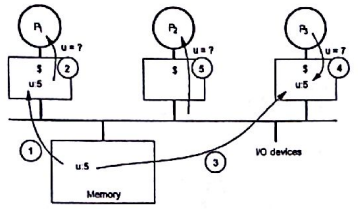


- Controller updates state of blocks in response to processor and snoop events and generates bus actions
- Often have duplicate cache tags

3/23:

ideally we want to get the most recent value.

Example Cache Coherence Problem



- Is there a problem?
  - Replication + writes = coherence problem

after step (4), there is a stale value in P1's cache & memory. Also, who gets to give (5) the value. Latest in $3, even memory is stale.

Coherence

- Fuzzy idea
  - Necessary for correctness
- Precise definition
- For any memory location X
  - Operations issued by any process occur in the order in which they issued
  - Value returned by read is the value written by last write
- Derivative properties
  - Write propagation: writes become visible to everyone (when?)
  - Write serialization: writes to a single location seen in same order by every processor

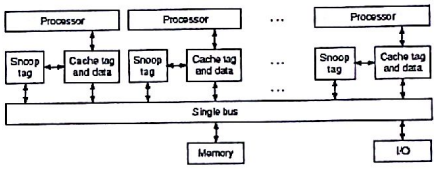
How to enforce coherence

- Correct actions:
  - On write, invalidate other copies
  - On write, update other copies
  (can do either)
- Snooping: (1 easy (maybe unnecessary))
  - Every node can "see" the bus
  - If all transactions can be observed
    - Each can maintain coherence

imp

Snoop tag: duplicate copy of all tags.

Snoopy coherence



- Controller updates state of blocks in response to processor and snoop events and generates bus actions
- Often have duplicate cache tags

Snoop controller sees the bus action and if it sees action, it updates itself.

When you snoop a write, you loose your read/write permissions.

At any time we can have max 1 writer and ∞ reader (given 0 writers).

Snoopy Design Choices

- Snoopy protocol
  - set of states
  - state-transition diagram
  - actions
- Basic Choices
  - write-through vs. write-back
  - invalidate vs. update

A 3-State Write-Back Invalidation Protocol

- 3-State Protocol (MSI)
  - Modified
    - one cache has valid/latest copy
    - memory is stale [RAM or L2 cache or L3 ...]
  - Shared
    - one or more caches have valid copy
  - Invalid

3 states for the blocks of data in cache.

- Must invalidate all other copies before entering modified state
- Requires bus transaction (order and invalidate)

All blocks initially in Invalid State.

M : have read & write permission.

S : have read permission.

I : no permissions.

MSI Processor and Bus Actions

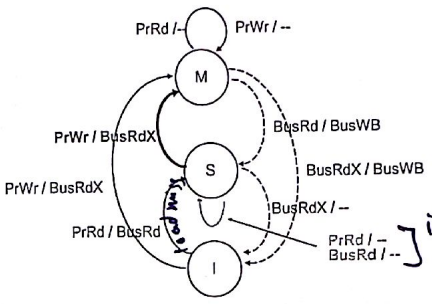
- Processor:
  - PrRd (load)
  - PrWr
  - Writeback on replacement of modified block
- Bus
  - Bus Read (BusRd) Read without intent to modify, data could come from memory or another cache
  - Bus Read-Exclusive (BusRdX) Read with intent to modify, must invalidate all other caches copies
  - Writeback (BusWB) cache controller puts contents on bus and memory is updated
  - Definition: cache-to-cache transfer occurs when another cache satisfies BusRd or BusRdX request
- Let's draw it!

memory inhibit is given to cache to cache transfer of data is done

ECE437, Spring 2016 (21)

on a tag level

MSI State Diagram



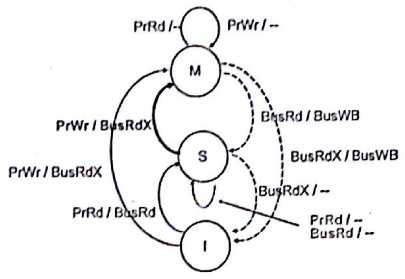
if I read or someone else reads no worries- Can stay in S.

ECE437, Spring 2016 (22)

Update vs invalidate

1. P1 updates P2:10 & they all use value. great to update.
2. P1 updates 1 value 100 times. better to invalidate. why? Bus to implement update takes Energy.

MSI State Diagram



An example

Proc Action	P1 State	P2 state	P3 state	Bus Act	Data from
1. P1 read u	S	--	--	BusRd	Memory
2. P3 read u	S	--	S	BusRd	Memory
3. P3 write u	I	--	M	BusRdX	Memory (?)
4. P1 read u	S	--	S	BusRd	P3's cache
5. P2 read u	S	S	S	BusRd	Memory

- Single writer, multiple reader protocol
- Why Modified to Shared?
  - Why not to Invalid?
    - Give up as little as possible. Retain option to read.
  - Why throw away write permission?
    - Ownership
- What if not in any cache?
  - Read, Write produces 2 bus transactions!

Invalidate vs. Update

- Pattern 1: better for update

```
for i = 1 to k
  P1(write, x);      // one write before many reads
  P2--PN-1(read, x);
end for i
```

- Pattern 2: better for invalidate.

```
for i = 1 to k
  for j = 1 to m
    P1(write, x);  // many writes before one read
  end for j
  P2(read, x);
end for i
```


Invalidate vs. Update, cont.

- Pattern 1 (one write before reads)
  - N = 16, M = 10, K = 10
  - Update
    - Iteration 1: N regular cache misses (70 bytes)
    - Remaining iterations: update per iteration (14 bytes: control & data)
  - Total Update Traffic = 16*70 + 9*14 = 1246 bytes
    - book assumes 10 updates instead of 9…
  - Invalidate
    - Iteration 1: N regular cache misses (70 bytes)
    - Remaining: P1 generates upgrade (6), 15 others Read miss (70)
  - Total Invalidate Traffic = 16*70 + 9*6 + 15*9*70 = 10,624 bytes
- Pattern 2 (many writes before reads)
  - Update = 1400 bytes
  - Invalidate = 824 bytes

Only 1 with Read Snarfing → becomes 1 if snarfing

is an invalidate optimization. Don't need Update then

P1 writes so 2:N-1 in invalidate when, say P6 reads value, (random) The others SNARF & get the valid.

otherwise, 2:N-1 must all get a read.

SNARFING makes sense only if there are more than 2 processors.

Coherence: Performance Issues

- Access Pattern specific optimizations
- Update vs. Invalidate
- Cache Block size
  - Remember 3C classification
  - 4th C: "Coherence miss" You read, someone wrote & this loops
  - True vs. False Sharing
- Read Snarfing
- Non-Atomic Bus

In reality this happens. We have super simplified it.

Coherence vs. Consistency

- Intuition says loads should return latest value
  - what is latest?
- Coherence concerns only one memory location
- Consistency concerns apparent ordering for all locations
- A Memory System is Coherent if
  - can serialize all operations to that location such that,
  - operations performed by any processor appear in program order
    - program order = order defined by program text or assembly code
  - value returned a read is value written by last store to that location

says each read happen to pull latest written value.

Why Coherence != Consistency

/* initial A = B = flag = 0 */

P1	P2
A = 1;	while (flag == 0); /*spin*/
B = 1;	print A;
flag = 1;	print B;

Intuition says printed A = B = 1

Coherence doesn't say anything, why?

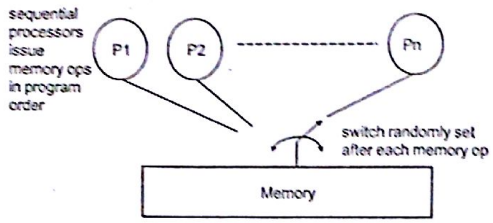
Consider coalescing write buffer

ECE437, Spring 2016

we expect 11 to be printed.

But coherence does NOT guarantee that assign happens in order.

The Memory Model



ECE437, Spring 2016

Sequential Consistency

- Register Allocation can break SC

a = f(x)	while (a == 0);
	b = g(a);
lw $arg1, x	lw $1, a (1)
jal f	loop: beqz $1, $0, loop
sw a, $retval (2)	lw $arg1, $1
	jal g (3)

- Use "volatile" declaration to prevent register allocation
  - `volatile int a;`

ECE437, Spring 2016

here, $1 gets 'a' and $1 never updates. So changing 'a' does not break loop.

To fix

problem line.

prevents 'a' in a register.

We want to keep reading 'a' from memory, not register.

This error may work 10 times, fail once then work again.

Sequential Consistency

- Programmers think in SC
  - i.e. preserve memory order
- Performance maximized when memory order is flexible
  - i.e., do not preserve memory order
- Architect's solution: Don't preserve order
  - Our problem is now the programmer's problem
  - Won't fly
- Can we get intuitive programming, correctness and fast performance?
- Several interesting approaches proposed
  - Parallel programming remains harder than sequential programming.
- Next: Say, we have SC
  - Can we co-ordinate activities of threads?

ECE437, Spring 2016

Why Synchronize?

lw $r1, ac-balance	lw $r1, ac-balance
add $r1, $r1, 100	add $r1, $r1, -50
sw $r1, ac-balance	sw $r1, ac-balance

- Two concurrent operations on the same bank account
  - Deposit $100 check
  - Withdraw $50 cash
- Correct result
  - ac-balance(new) = ac-balance(old)+50
- What can go wrong?

Interleaving of operations is bad. Must prevent this interleaving.

ECE437, Spring 2016

Synchronization

- Solution: Locks/Mutex
  - "critical section"
  - Only one person can have a lock
- Other synchronization primitives
- Barriers
  - Wait for everyone before proceeding
  - Can we just use a counter?
- Event notification
  - Producer consumer
  - Just use flags?
  - Consistency issues

```
LOCK ac-lock
lw $r1, ac-balance
add $r1, $r1, 100
sw $r1, ac-balance
UNLOCK ac-lock
```

```
LOCK ac-lock
lw $r1, ac-balance
add $r1, $r1, -50
sw $r1, ac-balance
UNLOCK ac-lock
```

ECE437, Spring 2016

Mar 30.

Why Synchronize?

Any interleaving will cause issues.

lw $r1, ac-balance	lw $r1, ac-balance
add $r1, $r1, 100	add $r1, $r1, -50
sw $r1, ac-balance	sw $r1, ac-balance

- Two concurrent operations on the same bank account
  - Deposit $100 check
  - Withdraw $50 cash
- Correct result
  - ac-balance(new) = ac-balance(old)+50
- What can go wrong?

Synchronization

- Solution: Locks/Mutex
  - "critical section"
  - Only one person can have a lock
- Other synchronization primitives
- Barriers
  - Wait for everyone before proceeding
  - Can we just use a counter?
- Event notification
  - Producer consumer
  - Just use flags?
  - Consistency issues

```
LOCK ac-lock
lw $r1, ac-balance
add $r1, $r1, 100
sw $r1, ac-balance
UNLOCK ac-lock
```

```
LOCK ac-lock
lw $r1, ac-balance
add $r1, $r1, -50
sw $r1, ac-balance
UNLOCK ac-lock
```


How Not To Implement Locks

many processors might read lock_variable = 0 and go ahead to lock it at the same time when only one should lock.

- LOCK
  while(lock_variable == 1);
  lock_variable = 1;
  0: unlocked 1: locked
- UNLOCK
  lock_variable = 0;
- Implementation requires Mutual Exclusion!
  - Can have two processes successfully acquire the lock

Problem solved if first to read automatically Locks the flag (i.e. writes to lock flag) → 3 Atomic operations

say P1 & P2 loads lock_variable. It is 0. So both lock the value & process code when only one guy should process it.

Atomic Read-Modify-Write Operations

intel ISA

- Test&Set(r,x)
  r = m[x]
  m[x] = 1
- Swap(r,x)
  r = m[x], m[x] = r
  - r is register
  - m[x] is memory location x
- Compare&Swap(r1,r2,x)
  if (r1 == m[x]) then
  r2 = m[x], m[x] = r2
- Fetch&Op(r,x,op)
  r = m[x], m[x] = op(m[x])

Performance of Test & Set

```
LOCK
  while (test&set(x) == 1);
UNLOCK
  x = 0;
```

- High contention (many processes want lock)
- Remember the CACHE!
- Each test&set is a read miss and a write miss
- Several Optimizations

BUT, testtset is not RISK type. The op. does 2 things together, un-RISK like.

Better Lock Implementations

- Two choices:
  - Don't execute test&set so much
  - Spin without generating bus traffic
- Test&Set with Backoff
  - Insert delay between test&set operations (not too long)
  - Exponential seems good (k*c^i)
  - Not fair
- Test-and-Test&Set
  - Spin (test) on local cached copy until it gets invalidated, then issue test&set
  - Intuition: No point in trying to set the location until we know that it's not set, which we can detect when it get invalidated...
  - Still contention after invalidate
  - Still not fair

test. If already locked, wait for rand(10). If locked, rand(100). If locked rand(1000) . . . .

chances that for same time, no ops happen.

load flag and check. If locked, you wait until UNLOCKED. At this point, your copy invalidates & so do a TEST & SET.

LL-SC based locks

- Different approach
  - Do not offer atomic Read-modify-write primitive
  - Offer mechanisms to detect non-atomicity
  - Load linked sets flag + address (link register)
    - Flag cleared if any other bus write observed
  - Store-conditional
    - Succeeds only if flag is set and address match

```
lock:   ll $s1, X       X is lock addr. $s1 gets value that says 1→locked 0→unlocked
        bne $s1, $0 lock  // held lock
        sc X,$s2          // $s2 contains 1
        beq $s2, $0 lock  // non atomic
        jr $31
unlock: sw $0, X
        jr $31
```

if $s1 ≠ 0 (it is locked) go back to lock to get value

Store 1 to lock operation (if) flag is still 1. If flag is 0, then $s2 gets 0.

ECE437, Spring 2018 (38)

LL puts X in register & sets a flag.
SC checks flag & ensures addr is same.

Synchronization: Performance Issues

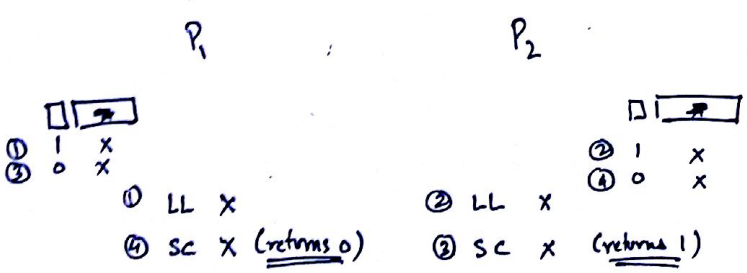
- Granularity
  - Lock single element vs. Lock whole array
  - Locking overhead vs. loss of concurrency
- Contention
  - TAS good at low contention
- Fairness
  - TAS offers no guarantees of fairness

ECE437, Spring 2018 (39)

Synchronization

- Deadlocks: Huge Correctness Issue
- Races
- May not be visible to the programmer
  - Hard to debug (Non-reproducible)
- "Locks don't compose with themselves"
  - Other synchronization mechanisms have other problems
- We live with this problem
  - Find a solution, become famous, rich.

ECE437, Spring 2018 (40)



Apr 1.

Synchronization.

Say we lock an array / hash table.
Then the other thread cannot operate.
This kills parallelism.

Now say we have a lock on bucket.
Need more memory but gives great concurrency. (Ilel programming).

TAS (test & set) do not gurantee fairness in operation order.
TAS has good contentions though.
↳ less requirement of locks for program operations.

contention: # of ppl fighting for resource.
good cont. → ↓ cont.

Deadlock.

Bank eg. Say there must be data ~~ta~~ transfer from A to B.
to send from A to B, need to lock both accounts.
Now, say P1 gets lock to A (A→B transfer)
& P2 gets lock to B (B→A transfer)
Now, they are stuck. DEADLOCK.

→ hard to debug.
→ Not reproducable.

To fix, we can have processor back off after 'n' fails.
• this causes LIVELOCK. (both back & come in),
To fix, we can have global lock. But this means single transaction at the ~~same~~ one time. Too inefficient.
To fix, let ID of smaller than ID of larger is locked.
So, in A→B & B→A, both try lock A first, then B.
In this case no issue happens (fixed).

Outline

- Basic arithmetic (Ch 3.1-3.5)
  - Representing numbers
  - 2's Complement, unsigned
  - Addition and subtraction
  - Add/Sub ALU
    - full adder, ripple carry, subtraction, together
  - Logical operations
    - and, or, xor, nor, shifts - barrel shifter
  - Carry lookahead, overflow
  - Integer multiplication, division
  - floating point representation/arithmetic

Unsigned Integers

- Recall:
  - n bits give rise to 2^n combinations
  - let us call a string of 32 bits as "$b_{31}\ b_{30} \ldots b_3\ b_2\ b_1\ b_0$"
- $f(b_{31} \ldots b_0) = b_{31} \times 2^{31} + \ldots + b_1 \times 2 + b_0 \times 2^0$
- Treat as normal binary number
  - e.g., 0 . . .011010101
  - $= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
  - $= 128 + 64 + 16 + 4 + 1 = 213$
- max f (111 . . . 11) = $2^{32} - 1 =$ 4, 294, 967, 295
- min f(000 . . . 00) = 0
- range [0, 2^{32}-1] => # values $(2^{32} - 1) - 0 + 1 = 2^{32}$

Numbers

- Bits are just bits (no inherent meaning)
  - — conventions define relationship between bits and numbers
- Binary numbers (base 2)
  - 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  - decimal: 0...2^n-1
- Of course it gets more complicated:
  - numbers are finite (overflow)
  - fractions and real numbers ***
  - negative numbers
  - e.g., no MIPS subi instruction; addi can add a negative number)
- How do we represent negative numbers?
  - i.e., which bit patterns will represent which numbers?

Number Representation

Sign Magnitude:	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Balance, number of zeros, ease of arithmetic

(xor. of numbers is 0. Can't have 2 types of zeroes.

Signed Integers

- 2's complement
- $f(b_{31}\ b_{30} \ldots b_1\ b_0) = -b_{31} \times 2^{31} + \ldots + b_1 \times 2 + b_0 \times 2^0$
  - max f(0111 . . . 11) = $2^{31} - 1 = 2147483647$
  - min f(100 . . . 00) = $-2^{31} = -2147483648$ (asymmetric)
- range [-2^{31}, 2^{31}-1] => #values $(2^{31}-1 - -2^{31} + 1) = 2^{32}$
- E.g., -6
- 000 . . . 0110 --> 111 . . 1001 + 1 --> 111 . . .1010

Two's Complement Operations

- Negating a two's complement number: invert all bits and add 1
  - remember: "negate" and "invert" are quite different!
- Converting n bit numbers into numbers with more than n bits:
  - MIPS 16 bit immediate gets converted to 32 bits for arithmetic
  - copy the most significant bit (the sign bit) into the other bits

    0010 -> 0000 0010

    1010 -> 1111 1010
  - "sign extension"
    - (remember ORI vs. LW, zero extend vs sign extend)

Negation in 2's complement

- Negation: Invert all bits, add 1
  - Why?
- If the (k+1) bit 2's complement representation of a number N is $\langle b_k\ b_{k-1} \ldots b_1\ b_0 \rangle$
  $N_k = -b_k \times 2^k + b_{(k-1)} \times 2^{(k-1)} + \ldots + 2^1 \times b_1 + 2^0 \times b_0$
- Show that the negation procedure is sound.

Negation in 2's complement

- Key trick:
  - Complement of bit b can be written as (1-b)
- $N_k = -b_k \times 2^k + b_{(k-1)} \times 2^{(k-1)} + \ldots + 2^1 \times b_1 + 2^0 \times b_0$
- Inversion gives us
  $-(1-b_k) \times 2^k + (1-b_{(k-1)}) \times 2^{(k-1)} + \ldots + 2^1 \times (1-b_1) + 2^0 \times (1-b_0)$
- Separating the red and blue terms and adding
  $-2^k + 2^{(k-1)} + \ldots + 2^1 + 2^0 + 1 - N_k$
- Blue terms plus 1 goes to zero. Q.E.D.

Sign extension

- Consider representation of -2:

3bit (decimal)	2-bit (decimal)
011 (+3)	
010 (+2)	
001 (+1)	01 (+1)
000 (0)	00 (0)
111 (-1)	11 (-1)
110 (-2)	10 (-2)
101 (-3)	
100 (-4)	

Mathematical basis

- Inductive proof
  - if the (k+1)-bit 2's complement representation of a number N is $\langle b_k\ b_{k-1} \ldots b_1\ b_0 \rangle$
    - $N_k = -b_k \times 2^k + b_{(k-1)} \times 2^{(k-1)} + \ldots + 2^1 \times b_1 + 2^0 \times b_0$
  - Then the (k+2)-bit 2's complement representation $\langle b_k\ b_k\ b_{k-1} \ldots b_1\ b_0 \rangle$ also represents N
    - $N = -b_k \times 2^{k+1} + b_k \times 2^k + b_{(k-1)} \times 2^{(k-1)} + \ldots + 2^1 \times b_1 + 2^0 \times b_0$
    - $= (-2 \times b_k + b_k) \times 2^k + b_{(k-1)} \times 2^{(k-1)} + \ldots + 2^1 \times b_1 + 2^0 \times b_0$
    - $= -b_k \times 2^k + b_{(k-1)} \times 2^{(k-1)} + \ldots + 2^1 \times b_1 + 2^0 \times b_0$

Addition and Subtraction

- Similar to decimal (carry/borrow twos instead of tens)
- Identical operation for signed and unsigned
  - E.g. Unsigned vs signed

0011	3	3
1010	10	-6
1101	13	-3

Interesting cases

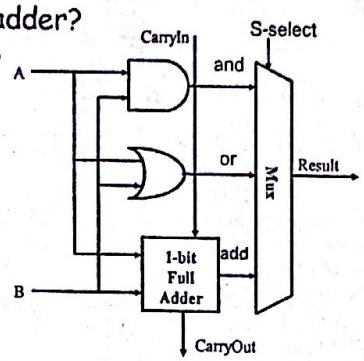
- Show computation in 4-bit 2's complement representation
  4+4

  (-4) + (-4)

- Overflow: later

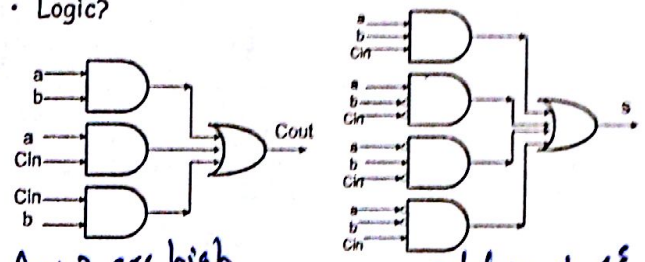
ALU bit-slice

- Bit-wise operation
  - and, or, add
  - Full adder?
  - Sub?



Full adder

- Three inputs and two outputs
- Cout, s = F(a,b,Cin)
  - Cout : only if atleast two inputs are set
  - S : only if exactly one input or all three inputs are set
- Logic?



Any 2 are high ⇒ Cout = 1

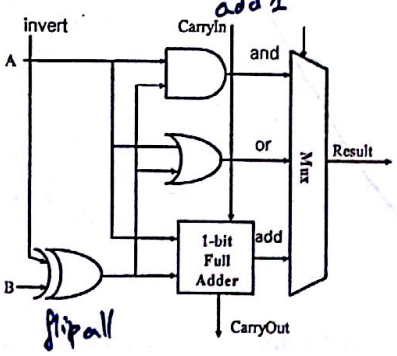
odd no of 1's give output of 1.

Subtract

- A - B = A + (- B)
  - form two complement by invert and add one

add 1

For Subtraction



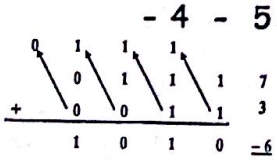
Exercise

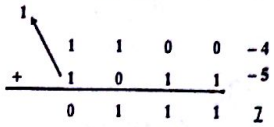
- How do I convert 237 to binary?
- What is the 2's complement representation (3-bit) of:
  -3:
  +5:
- If a number X is represented as $b_3\ b_2\ b_1\ b_0$ if 4 bit 2's complement arithmetic, what is the 8-bit 2's complement representation of X?
- Show the computation in 4-bit 2's complement arithmetic:
  5 - (-3)

Overflow

Decimal	Binary	Decimal	2's Complement
0	0000	0	0000
1	0001	-1	1111
2	0010	-2	1110
3	0011	-3	1101
4	0100	-4	1100
5	0101	-5	1011
6	0110	-6	1010
7	0111	-7	1001
		-8	1000

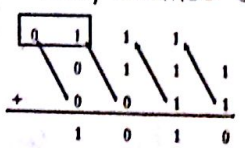
- Examples: 7 + 3 = 10 but ...
  - 4 - 5 = - 9 but ...



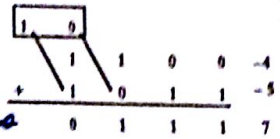


Overflow

- Overflow: the result is too large (or too small) to represent properly
  - Example: - 8 < = 4-bit binary number <= 7
- When adding operands with different signs, overflow cannot occur!
- Overflow occurs when adding:
  - 2 positive numbers and the sum is negative
  - 2 negative numbers and the sum is positive
- On your own: Prove you can detect overflow by:
  - Carry into MSB ⊕ Carry out of MSB

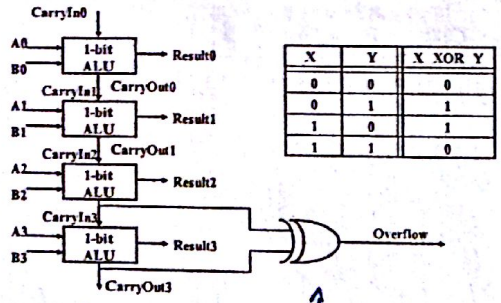


XOR for overflow



Overflow detection

- Carry into MSB ⊕ Carry out of MSB
  - For N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]



X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

for overflow.

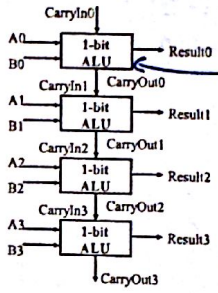
Negative, Zero

- Required for conditional branches
- Zero
  - How?
  - NOR all 32 bits
  - Avoid 33rd bit (carry out)
- Negative may be required on overflow
  - If (a<b) jump : jump taken if a-b is negative
- Tempting to consider MSB
  - E.g. if (-5 < 4) branch
  - Branch should be taken, but (-5-4) computation results in overflow... so MSB is 0
  - E.g. if (7 < -3) branch
  - Branch should not be taken but (7- (-3)) results in overflow... so MSB is 1.
  - Negative = ??

Outline

- Advanced addition techniques
  - The problem with ripple carry — ↑ linear added delays
  - Blocked ripple carry - ripple in chunks/blocks to avoid too much delay.
  - Carry Look-ahead adder
  - Carry select adder
  - Barrel shifter

Ripple-carry adder



has 2 gates of delay for 1 bit.

In 32 bit adder, there would be horrible gate delays.

Problem : Slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
  - Delay = 32 x Critical-path(Fast adder) + XOR
- Is there more than one way to do addition?
  - Two extremes: ripple carry and sum-of-products
  - Flatten expressions to two levels

Can you see the ripple? How could you get rid of it?

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$
$c_2 = b_1c_1 + a_1c_1 + a_1b_1$ $\quad c_2$ = <7 min-terms>
$c_3 = b_2c_2 + a_2c_2 + a_2b_2$ $\quad c_3$ = <15 min-terms>
$c_4 = b_3c_3 + a_3c_3 + a_3b_3$ $\quad c_4$ = <31 min-terms>

when we replace c_{n-1} with string.

Not feasible! Why? Exponential fanin

Now, we use more gates but or gates need more & more inputs.

Blocked Ripple Carry

- Flatten Logic in blocks of "k" say 4
  - But not the naïve version
  - Block of 4 requires 31-input OR gate
- Ripple carry from one block to the next
- Delay through N-bit addition
  - (N/4) * 2 + 1 (XOR)

k factor improvement

- Reduction by a constant factor
  - Still linear in number of inputs
  - Can do better: Logarithmic delay

No bits / block size × critical path + XOR delay

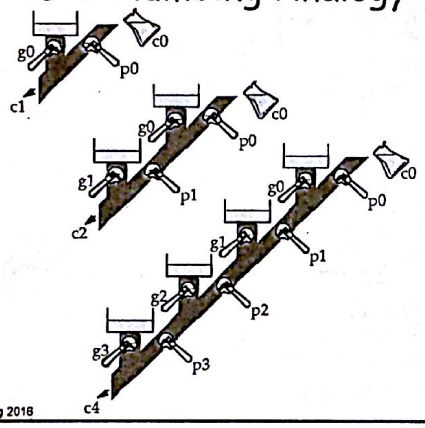
Carry Lookahead Adder

- Reformulate addition
  - Facilitates block computation
  - Facilitates hierarchical, parallel computation
  - Key concepts: Generate and Propagate
- An approach in-between our two extremes
  - Ripple carry
  - Flattened 2-level

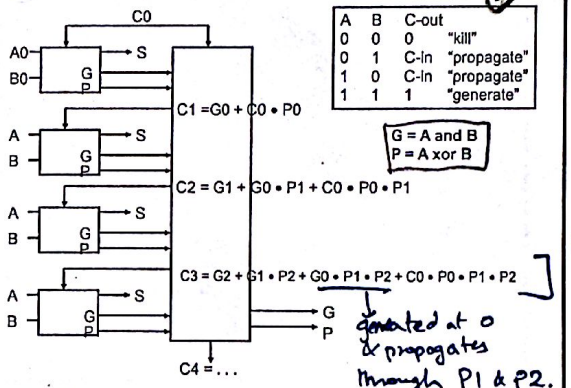
Carry look-ahead

- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry?
    - $g_i = a_i . b_i$
  - When would we propagate the carry?
    - $p_i = a_i \oplus b_i$
- Did we get rid of the ripple?

CLA: Plumbing Analogy



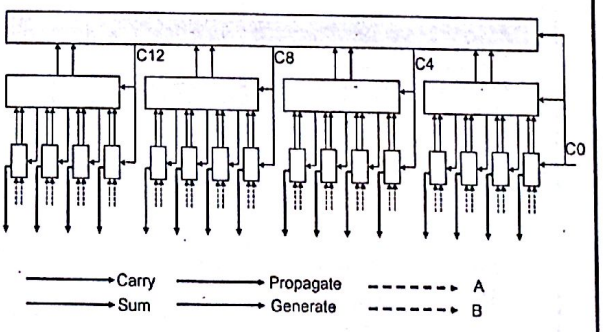
Carry-lookahead adder



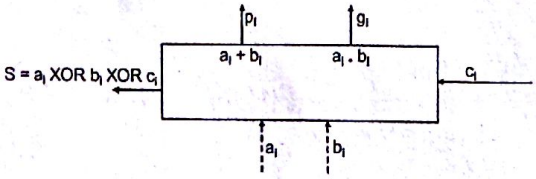
Carry-Lookahead Adder

- Waitaminute!
  - Nothing has changed
  - Fanin problems if you flatten!
    - Not really, Linear fanin, not exponential
  - Ripple problem if you don't!
- Enables divide-and-conquer
- Figure out Generate and Propagate for k-bits together
- Compute hierarchically

2-level 16-bit CLA



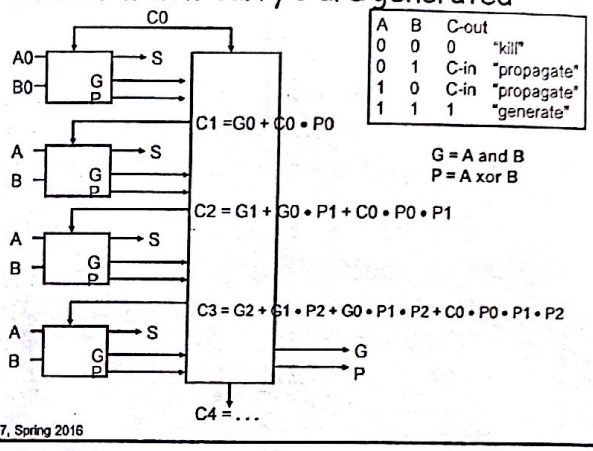
Leaf Node



- Zero level gi, and pi generation
  - One gate delay
- Part of Full adder (only sum bit)
  - Two gate delays (ignoring inverters)

Intermediate nodes

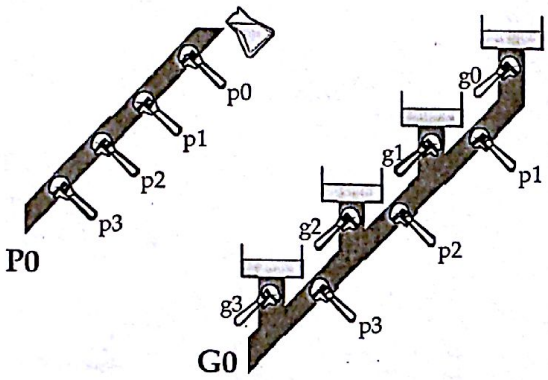
- We know how carry's are generated



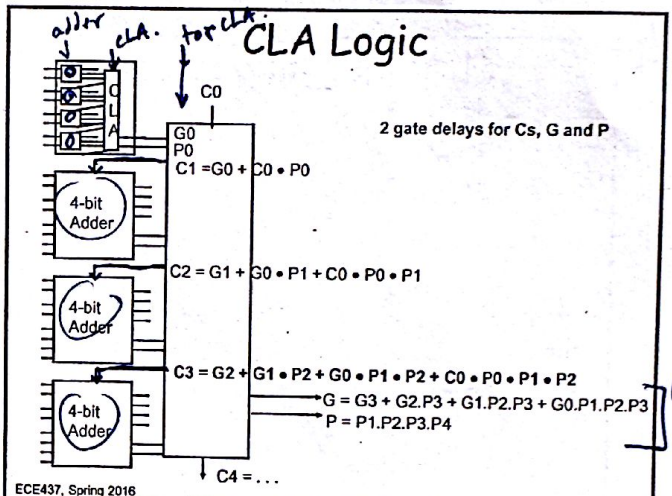
A	B	C-out	
0	0	0	"kill"
0	1	C-in	"propagate"
1	0	C-in	"propagate"
1	1	1	"generate"

G = A and B
P = A xor B

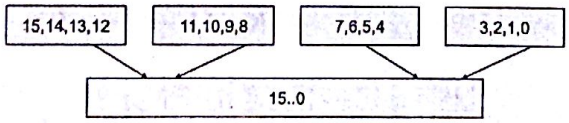
Block level signals



CLA Logic

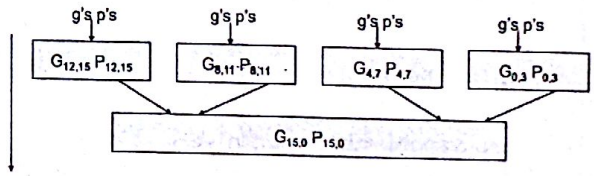


CLA: Delay Analysis



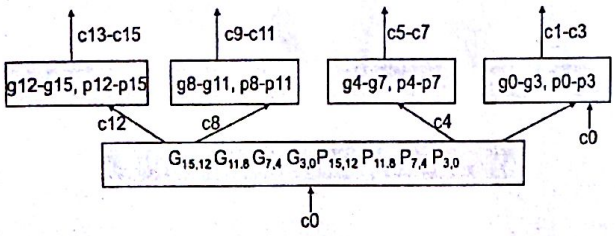
- Height of tree = $O(\log_4(n))$
- 16 bit addition : $k*\log_4(16) = k*2$

Computing G's and Ps



- Parallel algorithm in hardware
  - g's and p's : 1
  - First level Gs and Ps : 1 + 2 = 3
  - Second level Gs and Ps : 1 + 2 + 2 = 5
    - (not needed for 16 bit adder example)

Computing C's



- Different tree.
- Note propagation order
- Worth spending some time to think about the intricacies

Delays

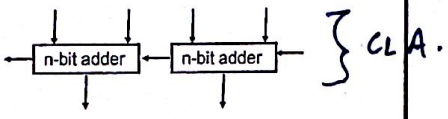
- c1—c3 : 1 + 2
- c4 : 1 + 2 + 2 **
- c5-c7 : 1 + 2 +2 + 2 = 7
- c8 : 1 + 2 + 2
- c9-c11 : 1 + 2 + 2 + 2 = 7
- c12: 1 + 2 + 2
- c13-c15 : 1 + 2 + 2 + 2 = 7

- What about sum-bits?

** Can be 1 + 2 if computed in first level CLA block. No overall improvement by doing this

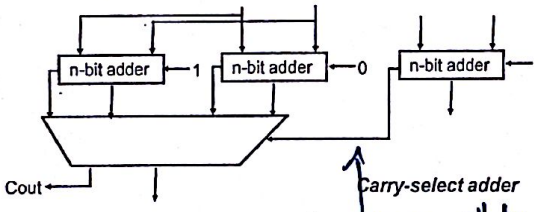
Carry-selection: Guess

CP(2n) = 2*CP(n)



} CLA.

CP(2n) = CP(n) + CP(mux)



carry will be 0 or 1. Pick from mux.

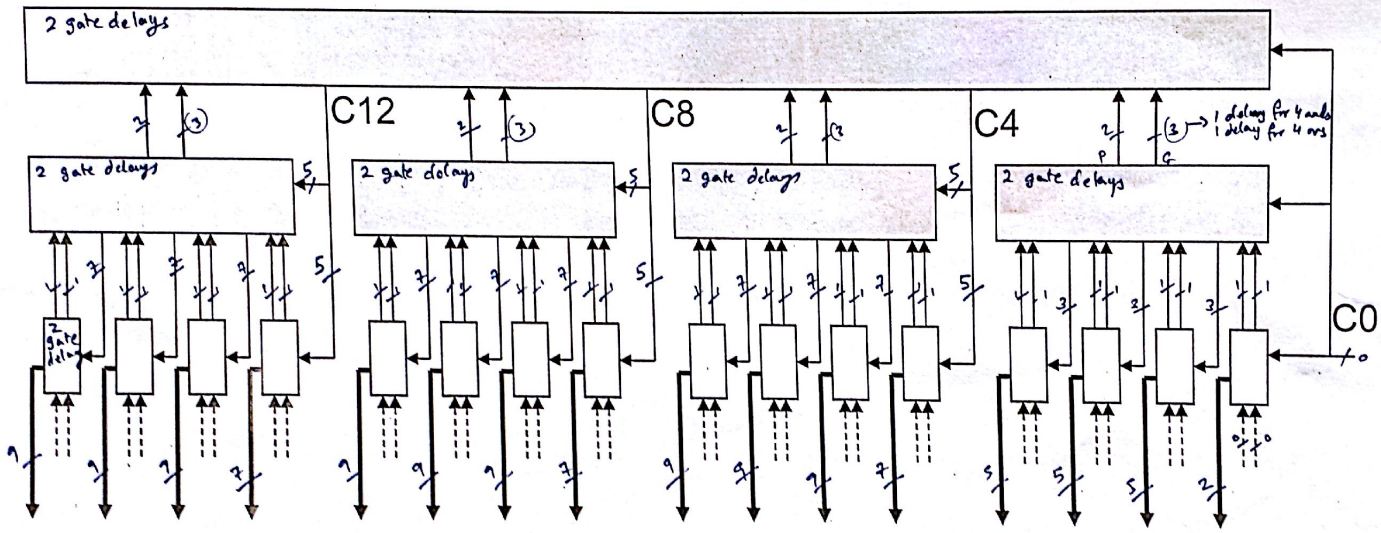
costs hardware & energy.

→ GATE DELAY CALCULATION:

2-level 16-bit CLA

Assume we have 4 input gate that has same delay as 2 input gate.

For 32-bit, just add another level. [takes only extra 2 gate delays]



Sum ready in 9 gate delays.

Carry — Propagate — A

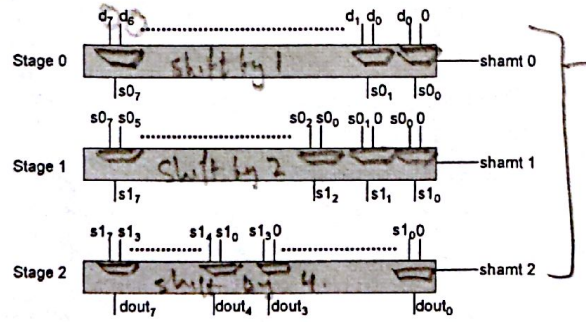
Sum — Generate — B

At high bit nos, CLA needs high no of wires that slow it a bit.

Shift

- E.g., Shift left logical for d<7:0> and shamt<2:0>
  - Using 2-1 muxes called Mux(select, in0, in1)
  - stage0<7:0> = Mux(shamt<0>,d<7:0>, d<6:0> || 0)
  - stage1<7:0> = Mux(shamt<1>, stage0<7:0>, stage0<5:0> || 00)
  - dout<7:0> = Mux(shamt<2>, stage1<7:0>, stage1<3:0> || 0000)
- Other operations
  - Right shift
  - Arithmetic shifts
  - Rotate

Barrel Shifter



3 bits allowed 0-7 shifts.

Logarithmic efficiency

32 bit no has 5 shift bits.
↳ $\log_2 32$.

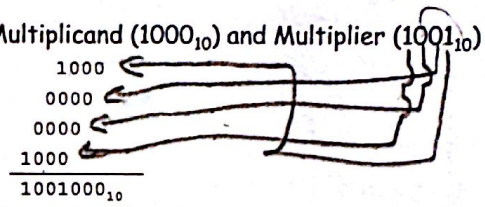
8 bit no has 3 shift bits
↳ $\log_3 8$.

Extensions

- Design a barrel shifter unit that can do
  - Right shift
  - Left shift
- Design a shifter/rotator combo that can do
  - Right rotate
  - Left rotate
  - In addition to shifts

Grade School Multiplication

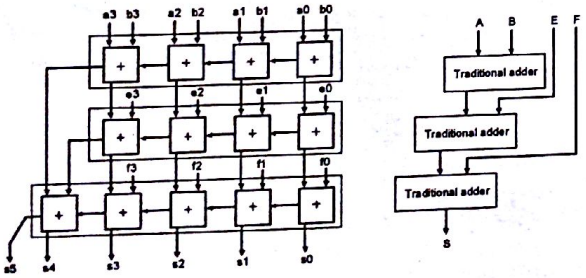
- Multiplicand (1000_{10}) and Multiplier (1001_{10})



- Two approaches
  - Purely combinational
  - Sequential

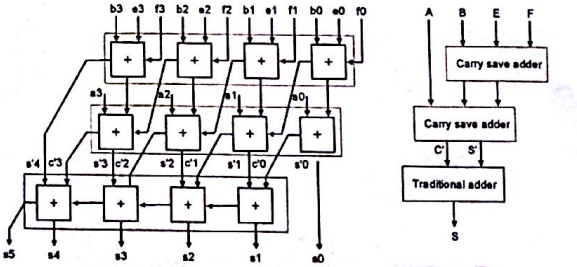
good to add many nos.

Adding more than two operands



- Add four numbers (A,B,E,F)
- "Traditional adder" : ripple carry
- Basic block: Full adder
- N-1 adders for N numbers

Carry-save adders



- Same basic block
- Adds three numbers per stage (using Cins) and output 2 numbers (Cout and S)
- Number of stages reduced to ~2/3N for N-numbers

Say we had 9 no.
11^{th} can do 3 trees.
[Wallace tree]

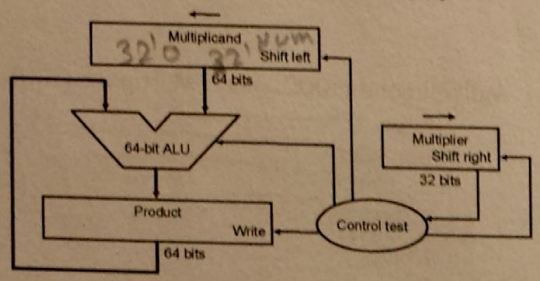
Purely combinational

- Uses carry-save adders (in a tree organization)
- Partial products
  - AND multiplicand with multiplier bits

Grade School Multiplication

- Sequential approach
- Reuse hardware
  - Partial products generated, accumulated into product each cycle

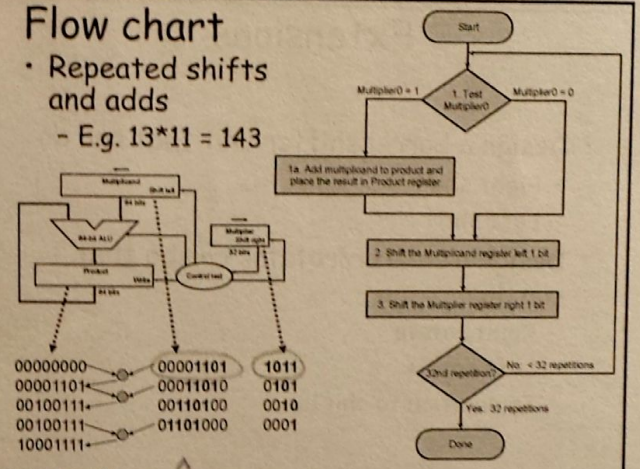
Grade-School Version



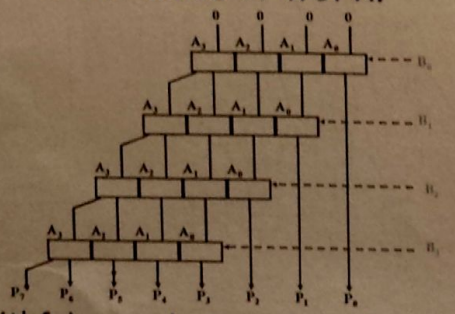
- Naïve implementation
  - 2x 64-bit registers, 1x 32 bit register
  - 1x 64-bit ALU

Flow chart

- Repeated shifts and adds
  - E.g. 13*11 = 143

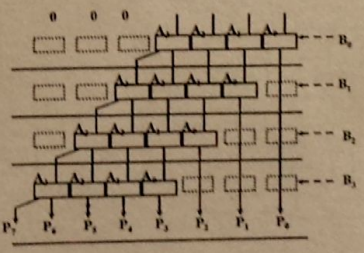


How does it work?



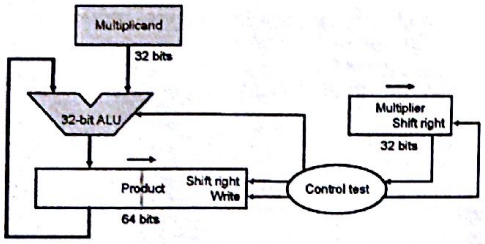
- Faithful reproduction of grade-school algorithm

Version #1



- Additional zeroes (padding)

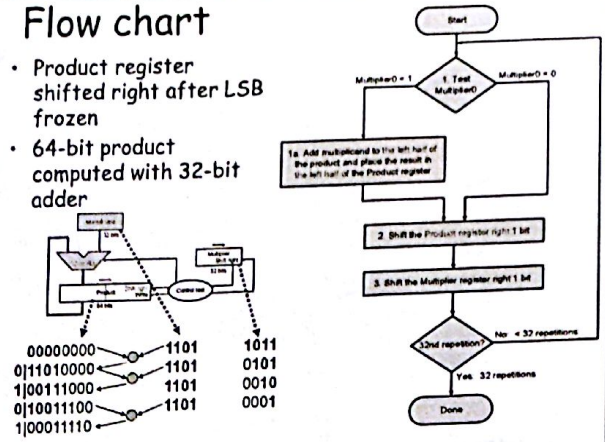
Optimization # 1



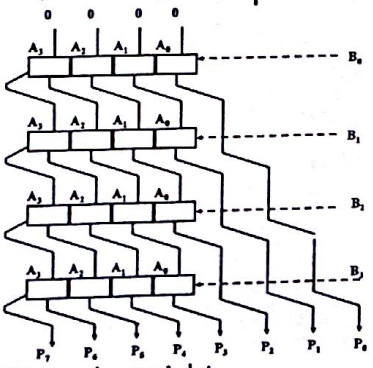
- Insight: LSB bits are frozen after each iteration
- Reduced 64-bit ALU to 32-bit ALU

Flow chart

- Product register shifted right after LSB frozen
- 64-bit product computed with 32-bit adder

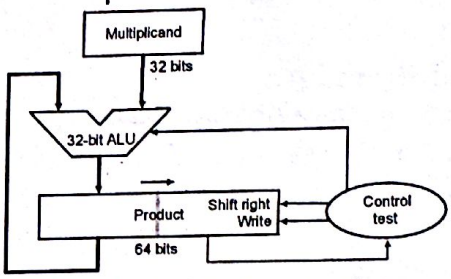


Flow of computation



- Shift product right

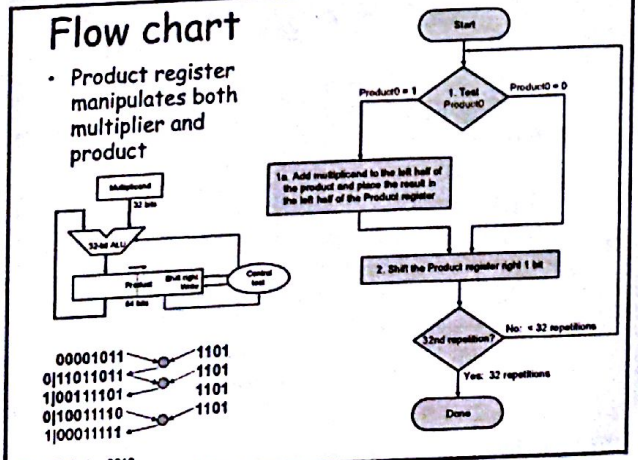
Optimization # 2



- Insight: bits of multiplier are consumed
  - 1x 32-bit register
  - 1x 64-bit register
  - 32-bit ALU

Flow chart

- Product register manipulates both multiplier and product



Signed Multiplication

- { <+,+>, <-,-> } : +
- { <+,->, <-,+> } : -

- If multiplier negative,
  - Multiplier = - Multiplier
  - Negative ++
- If multiplicand negative,
  - Multiplicand = -Multiplicand
  - Negative ++
- Product = Multiplicand * Multiplier
- If (Negative) sign = '-' , else sign = '+'

Booth's Algorithm

- Signed/Unsigned integer multiplication
  - Multiple bits at a time
    - We won't go into this topic
  - Like CLA for addition
    - sophisticated
- Intuition : exploit run-lengths

ECE437, Spring 2016

Booth's Algorithm Intuition

- How would you compute the decimal product:
  - 5345 * 999 ?
  - 5345 * (1000 - 1)
  - 5345000 - 5345 = 5339655
- What about
  - 5345 * 9990
  - 5345 * (10000 - 10)
  - 53450000 - 53450 = 53396550
- Now, what about
  - 5345 * 9900990
  - 5345 * (1000000 - 100000 + 1000 - 10)
  - 52915500000 + 5291550 = 52920791550
- Works only when multiplier has only '9's in the decimal system

ECE437, Spring 2016

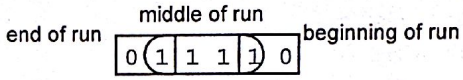
look at pairs of bits & operate based on their values.

Application to Binary

- "1 is the new 9"
- 9 = 10-1 (10 is base of decimal system)
- 1 = 2-1 (2 is the base of binary system)
- For example
  - 0010 * 0011
  - 0010 * (0100 - 0001)

ECE437, Spring 2016

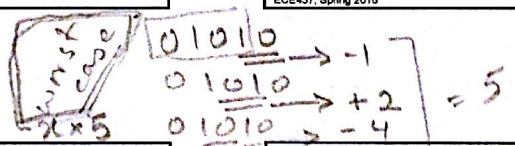
Run of '1's identification



Current bit	Bit to right	Explanation	Example	Action
1	0	Start of run of '1's	11001110	sub
1	1	Middle of run of '1's	11001110	nop
0	1	End of run of '1's	11001110	add
0	0	Middle of run of '0's	11001110	nop

ECE437, Spring 2016

0 padded at end.



Works for Signed numbers

- Consider 2's complement number 10110
  - Normal 2's complement view
    - 10110 = -16 + 0 + 4 + 2 + 0 = -10
  - Booth encoding view
    - 10110 = -16 + 8 - 0 - 2 + 0 = -10
    - 10110 :: 1$\bar{1}$0$\bar{1}$0 in Booth encoding
      - 1011<u>0</u>(0)
      - 101<u>10</u>(0)
      - 10<u>11</u>0(0)
      - 1<u>01</u>10(0)
      - <u>10</u>110(0)

ECE437, Spring 2016

bit pair 5 shows math op each cycle

Booth's Algorithm

Example: -4*6

- 4 bit 2's complement 1100 * 0110
- Extend multiplicand width
  - -4 = 1111 1100
- Booth encoding of Multiplier
  - 6 = 1$\bar{1}$0 (corresponds to +8 -2)

1111 1100	0	0000 0000
1111 1000	$\bar{1}$	0000 1000
1111 0000	0	0000 0000
1110 0000	1	1110 0000
		1110 1000

ECE437, Spring 2016

-4 x 6

		Partial prod
0110(0)	nop	0000 0000
0110(0)	sub	0000 1000
0110(0)	nop	0000 1000
0110(0)	add	1110 1000

1111 1100
1 1111 1000 shift 1
11 1111 0000 shift 2
111 1110 0000 shift 3

[sub -4 x 2]

[add -4 x 8]

The Mathematical Basis

- Recall the table
  - Multiplier = $a_{j-1} - a_j$
- 3-bit numbers
  - A (a_2,a_1,a_0)
  - B (b_2,b_1,b_0)
- Compute B x A
  - $B * \{ (a_1 - a_2).2^2 + (a_0 - a_1).2^1 + (0 - a_0).2^0 \}$
  - $B * \{ -a_2.(2^2) + a_1.(2^2-2^1) + a_0.(2^1-2^0) \}$
  - $B * \{ -a_2.(2^2) + a_1.(2^1) + a_0.(2^0) \}$
  - B * A !!

Current bit	Bit to right	Multiplier
1	0	-1
1	1	0
0	1	+1
0	0	0

Summary

- Integer multiplication:
  - Grade school version with minor optimizations
- Sophisticated optimizations
  - Booth's algorithms
    - More involved
    - Scope for handling multiple bits at a time
  - Remember CLA vs. ripple-carry (grade-school version)

Recap

- Booth's Multiply Algorithm
  - Identify and exploit runs of '1's
  - Remember analogy of multiplication by 999
    - 0110 normal encoding (4+2 = 6)
    - $10\bar{1}0$ Booth encoding (8-2 = 6)

Outline

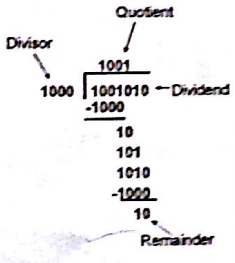
- Integer Division
  - Restoring/Non-restoring
- Floating Point Arithmetic

Integer Division

- Remember progressive optimization of multiplication hardware
  - Similar Story
- Start with naïve hardware
  - Faithful implementation of grade-school method (long division)
  - Optimize

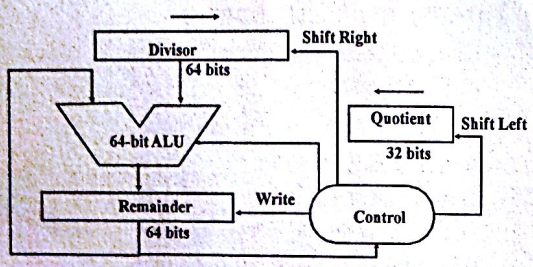
Grade School Division

- Consider decimal division
  - Subtract largest possible multiple
  - Shift down one digit
  - "Lather, Rinse and repeat"
  - Binary
    - Exactly two choices: 0 or 1



Dividend = Quotient * Divisor + Remainder
|Dividend| = |Quotient| + |Divisor|

Version #1

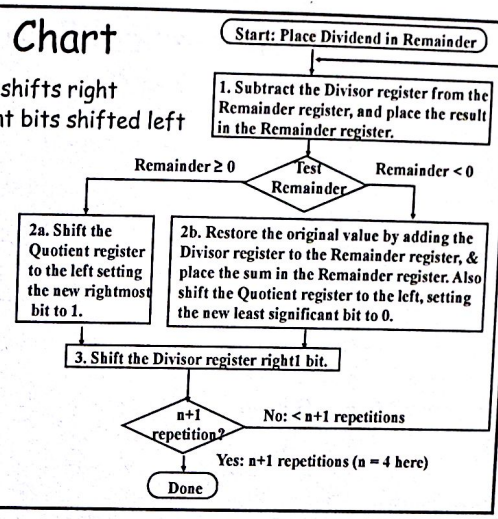


- Dividend initially in Remainder reg
- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg

ECE437, Spring 2016

Flow Chart

- Divisor shifts right
- Quotient bits shifted left



ECE437, Spring 2016

Example

- n+1 steps for n-bit quotient
- 7 / 2 = (3,1)
- Use 8(4) bit 2's complement representation
- Register size/ALU size optimizations possible

Remainder	Divisor	Quotient
0000 0111	0010 0000	0000
1110 0111		
0000 0111	0001 0000	0000
1111 0111		
0000 0111	0000 1000	0000
1111 1111		
0000 0111	0000 0100	0000
0000 0011	0000 0010	0001
0000 0001	0000 0001	0011

ECE437, Spring 2016

Recap

- Booth's Multiply Algorithm
  - Identify and exploit runs of '1's
  - Remember analogy of multiplication by 999
    - 0110 normal encoding (4+2 = 6)
    - $10\bar{1}0$ Booth encoding (8-2 = 6)

Outline

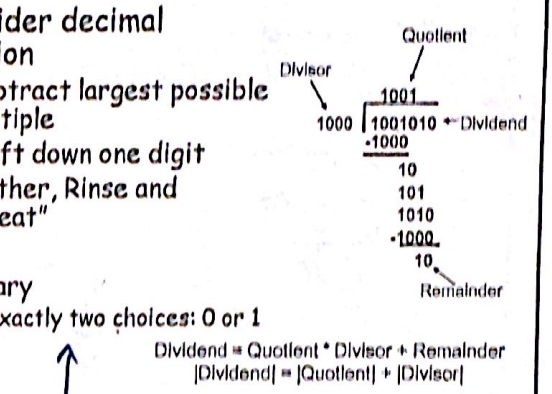
- Integer Division
  - Restoring/Non-restoring
- Floating Point Arithmetic

Integer Division

- Remember progressive optimization of multiplication hardware
  - Similar Story
- Start with naïve hardware
  - Faithful implementation of grade-school method (long division)
  - Optimize

Grade School Division

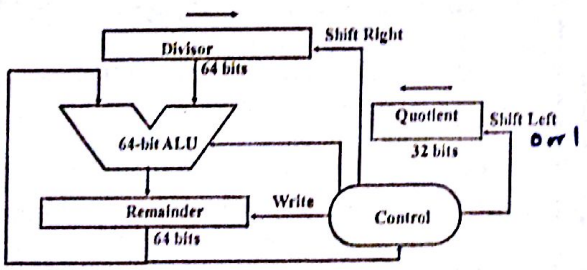
- Consider decimal division
  - Subtract largest possible multiple
  - Shift down one digit
  - "Lather, Rinse and repeat"
  - Binary
    - Exactly two choices: 0 or 1



Dividend = Quotient * Divisor + Remainder
|Dividend| = |Quotient| * |Divisor|

hardware doesn't know tables. but we have either 0 or 1. So we guess 1 and see if subtraction results in +ve no. If not, we go up a level & use 0.

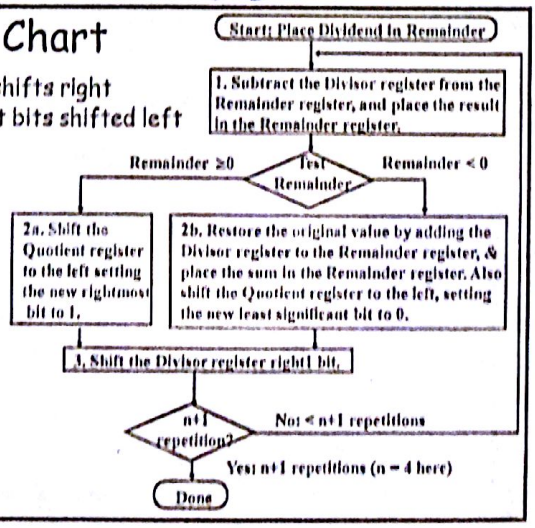
Version #1



- Dividend initially in Remainder reg
- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg

Flow Chart

- Divisor shifts right
- Quotient bits shifted left



Example

- n+1 steps for n-bit quotient
- 7 / 2 = (3,1)
- Use 8(4) bit 2's complement representation
- Register size/ALU size optimizations possible

Remainder (7)	Divisor (2)	Quotient
0000 0111	0010 0000	0000
1110 0111		
0000 0111	0001 0000	0000
1111 0111		
0000 0111	0000 1000	0000
1111 1111		
0000 0111	0000 0100	0000
0000 0011	0000 0010	0001
0000 0001	0000 0001	0011

rem. quotient

run only n+1 times.

always do remainder - Divisor.
If -ve, restore.
shift divisor Right
shift Quotient Left.
If +ve. shift divisor right.
get 1 in quotient on left shift.

not in exam.

Non-restoring division

- Restoring Division
  - Subtract largest possible multiple
    - Try '1'
    - If too large (i.e. remainder negative), restore
- Alternative
  - Non-restoring division
  - Introduce $\bar{1}$ (Like Booth's)

Non-Restoring Division

- With +ve divisor
  - Subtract if positive remainder
  - Add if negative remainder
- Vice versa with -ve divisor

	Remainder	Divisor	Quotient
-	0000 0111	0010 0000	00000
+	1110 0111	0001 0000	00001
+	1111 0111	0000 1000	00011
+	1111 1111	0000 0100	00111
-	0000 0011	0000 0010	01111
	0000 0001	0000 0001	11111

Quotient = 0011 => $1 \times 2^1 + 1 \times 2^0$
Quotient = $\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}$ => $1 \times 2^4 - 1x2^3 - 1x2^2 - 1x2^1 + 1x2^0$

Conversion to standard form

- Non-restoring division uses two unique symbols
  - Can use "0" for $\bar{1}$
- Issues
  - Perfect division before all bits are computed
  - Re-conversion to standard form

Outline

- Integer Division
  - Restoring and Non-restoring
- Floating Point Arithmetic

Floating Point

- Integer arithmetic till now
- How to represent real numbers:
  - Uncountably infinite
  - Realistically, can operate with limited number of significant digits (precision)
  - Remember exponent separately
- Remember scientific norms
  - Planck's constant 6.626068×10^{-34} m² kg/s
  - Avogadro's constant 6.022×10^{23}
  - Normalization:
    - Not 0.6022×10^{24}, Not 60.22×10^{22}
    - MSD in [1,9] range except for 0.0

mantissa ↓ $a e^{b}$ ← exponent

FP in hardware

- Binary instead of decimal
- MSB = 1 (equivalent to [1,9] in decimal)
  - $(-1)^s \times f \times 2^e$
  - s: Sign stored separately
  - f: is of the form 1.F
  - e: is the (signed integer) exponent
- Optimizations:
  - Base not stored (implicit 2)
  - MSB not stored (always 1)

f is 1.0110... ↓ This is stored, ∵ we know that 1 will always be there.

ECE437, Spring 2016

still need to have a way to represent zeroes.

If all are zero, it has value 0. that +1 is forgotten then.

Binary Fractions Recap

- How do you represent $3.125_{(10)}$ in binary?
  - $11.001_{(2)}$
- For integer conversion to binary
  - Repeated division by 2 till quotient goes to zero
- For fraction conversion to binary
  - Repeated multiplication by 2 till fraction goes to zero
- Recurring binary fraction
  - $0.6_{(10)} = 0.1001\ 1001\ 1001\ \ldots._{(2)}$

0.125 * 2 = 0.25
0.25 * 2 = 0.5
0.5 * 2 = 1.0 ↓ 0 0 1

0.6 * 2 = 1.2
0.2 * 2 = 0.4
0.4 * 2 = 0.8
0.8 * 2 = 1.6
0.6 * 2 = 1.2

1.1001...

ECE437, Spring 2016

make 0.6 to binary.
Do by multip.
0.6 × 2 = (1).2
0.2 × 2 = (0).4
0.4 × 2 = 0.8
0.8 × 2 = (1).6
0.6 × 2 = ...

IEEE 754

float : 32 bits — E (8 bit)

double : 64 bits — E (11 bit)

- Floating point representation standard
- Floating points stored as a packed triple
  - <S, E, F>
  - S : 1 bit
    - 0 → positive
    - 1 → negative
  - E : Biased representation for signed numbers
    - 8 bit (single precision—float) (Bias = +127)
    - 11 bit (double precision—double) (Bias = +1023)
  - F : (remember the 1 in 1.F is not stored)
    - 23 bit (single precision—float)
    - 52 bit (double precision—double)

ECE437, Spring 2016

Exceptions

- Specified in great detail in the document IEEE 754-1985
  - 20 pages

S E F not covered.

S	E	F	Number
0	0	0	0
0	Max	0	+inf
1	Max	0	- inf
X	Max	!= 0	NaN ← not a no. $= x/0$ $= \log(-x)$
X	0	!= 0	Denorm

ECE437, Spring 2016

2 reserved in exp. for special cases. Hence bias = 127 & not 128 [8 bit exp] (For balance).

denormalized. No, too small to represent.

Examples

- 3.125 in double precision
  - 11.001
  - $(-1)^0 \times 1.1001 \times 2^1$ → to shift by 1 for 11.
  - Exponent: 1 with bias of 1023 = 1024
  - 0100 0000 0000 1001 0000 0000 ...
  - 0x4009 0000 0000 0000
- 0.6 in single precision
  - 0.100110011001...
  - $(-1)^0 \times 1.001100110011... \times 2^{-1}$
  - Exponent: -1 with bias of 127 = 126
  - 0011 1111 0001 1001 1001...
  - 0x3F19 9999

suggests recurring value.

ECE437, Spring 2016

FP Representation Worksheet

- Consider two floating point numbers (represented in single precision, IEEE 754 format). A = 0xBFC00000 and B = 0x41600000. If C = A x B, what is the hexadecimal representation of C in IEEE 754 format.

$A = \underbrace{1011\ 1111\ 1100}_{127} = (-1)^{\times}\ 1.5 \times 2^{127-127}$
$= -1.5.$

$B = 0x416$
$= 0100\ 0001\ 0110 = (+1) \times 2^3 \times 1.75 = 14$

$C = -21 = -10101.000$
$= -1.0101 \times 2^4$

$s = 1$
$f = 0101_2$
$e = 131_d = 1000\ 0011_2$

$= 1100\ 0001\ 1010\ 1000\ 0000\ 0000\ 0000\ 0000$

C = 0xC1A8 0000...

ECE437, Spring 2016

Solution

float : 32 bits — E (8 bit)

- A = 0xBFC00000
  = 1011 1111 1100 0000 0000 0000 0000 0000
  = -1 * 1.5 * 2^0
- B = 0x41600000
  = 0100 0001 0110 0000 0000 0000 0000 0000
  = +1 * 1.75 * 2^3
- C = -21 = -1 * 1.0101 (binary)* 2^4
  = 1 100 0001 1010 1000 0000 0000 0000 0000
  = 0xC1A80000

Biased Exponent

- Why?
  - 2's complement arithmetic was elegant
  - Larger numbers appear to have larger magnitude with biased representation
    - Negative numbers appear large in 2's complement
  - Comparing two FP numbers is like comparing two integers with biased exponent

↳ comparing is common. Bias helps do that.

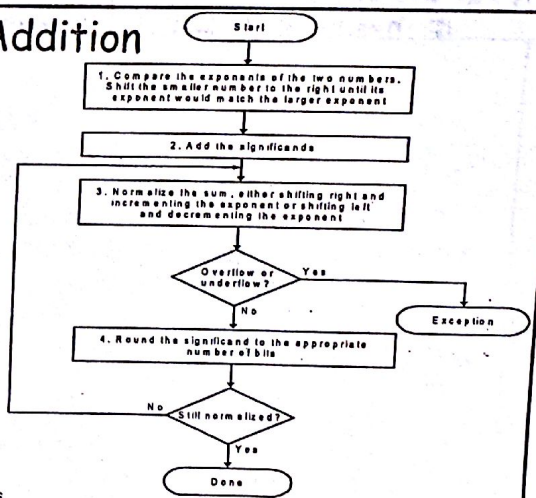
2's complement meant that the magnitude must be extracted. This is bad.

Floating Point Addition

- FP addition
  - Align decimal point → 9.997 x 10^2 + 0.0046371 x 10^2
  - Add → 10.0016371 x 10^2
  - Normalize → 1.00016371 x 10^3
- May involve rounding
  - The LSB may be shifted right beyond the limited precision

9.997 x 10^2 + 4.6371 x 10^{-1}

FP Addition



Floating Point Addition

- FP addition
  - Align decimal point
  - Add
  - Normalize

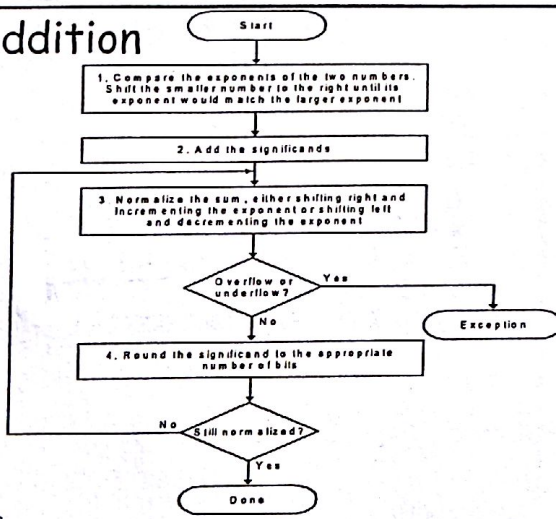
$$\begin{array}{r} 9.997 \times 10^{2} \\ +\ 4.6371 \times 10^{-1} \end{array}$$

$$\begin{array}{r} 9.997 \times 10^{2} \\ +\ 0.0046371 \times 10^{2} \\ \hline 10.0016371 \times 10^{2} \end{array}$$

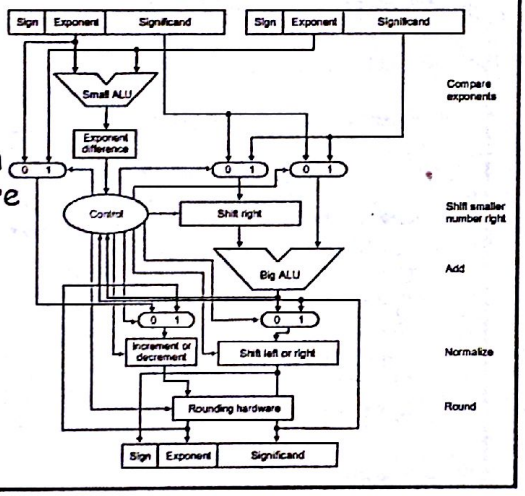
$$1.00016371 \times 10^{3}$$

- May involve rounding
  - The LSB may be shifted right beyond the limited precision

FP Addition



- FP Addition Hardware



FP Subtraction

- At most one shift for normalization after add
  - Applies when two operands are of same sign
  - Also the case when subtracting two operands of different signs
- Subtraction
  - Subtract two numbers of similar sign
  - Add two numbers of dissimilar sign

Example

- $1.00010 \times 10^{13} - 1.00001 \times 10^{13}$
- 0.00001×10^{13}
  - Not normalized
- 1.0×10^{8}
  - Normalize involves multiple digit shifts

FP multiplication

- Example
  - $A = 3.0 \times 10^{1}$
  - $B = 5.0 \times 10^{2}$
  - $A \times B = (3.0 \times 5.0) \times 10^{(1+2)} = 15.0 \times 10^{3}$
  - $A \times B = 1.5 \times 10^{4}$
- Multiply mantissas
- Add exponents (no overflow/underflow)
- Normalize (and round)
- Set sign (easy, exor sign bits)

Adding exponents

- Biased representation
- Adding biased numbers.
- Raw number

$e_1 = 12, e_2 = 13 ; e_* = (e_1+e_2) = 25$

$E_1 = 12 + 127 = 139, E_2 = 13 + 127 = 140$

$E_* = e_*+127 = E_1 - 127 + E_2 - 127 + 127$

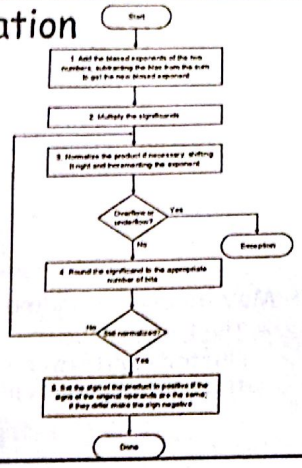
$E_* = E_1 + E_2 - 127$

$-127 = -(01111111) = (1000\ 0000) + 1$

```
1000 1011
1000 1100
1000 0001
1001 1000
```

re-biac.

FP Multiplication



FP Multiplication

- Mantissa multiplication
  - Two non-negative integers
  - Remember combinational approach
    - Carry save adders in tree (Wallace tree)

for multiplication

FP division

- Exponent computation
  - $E_/ = E_1 - E_2 + 127$ → re-biac. → their bias cancel each other
- Raw number

$e_1 = 12, e_2 = 13 ; e/ = (e_1-e_2) = -1$

$E_1 = 12 + 127 = 139, E_2 = 13 + 127 = 140$

$E_/ = e_/+127 = (E_1 - 127) - (E_2 - 127) + 127$

$E_/ = E_1 - (E_2 - 127)$

$-E_2 = 1's\ C\ (E_2) + 1$

$127 = (01111111)$

$E_/ = E_1 + 1'sC\ (E_2) + 1000\ 0000$

```
1000 1011
1000 1100
```

```
1000 1011   E1
0111 0011   E2 bar
1000 0000   -> 127+1
```

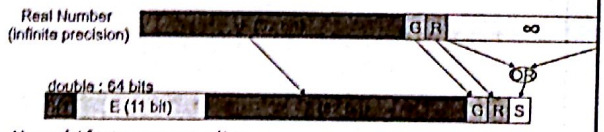
```
0111 1110
```

$\bar{E_2}+1$ is $-E_2$ in 2's comp.

Rounding

- Decimal conventions
  - [5+,9] -> up
  - 5 -> round to even (4.5 ~ 4, 5.5 ~ 6)
  - [1,5-] -> down
  - 0 -> uncharged
- Binary rounding
  - xxx.1...1... -> up
  - xxx.10000 -> round to even
  - xxx.0xx1x -> down
  - xxx.00000 -> uncharged

Guard, Round and Sticky bits



- May shift to renormalize
  - Save one bit to right of LSB (Guard bit)
  - This is a significant bit in the normalized representation
  - Round-bit
    - one additional to the right of guard bit
    - Used only for rounding, never becomes significant bit
  - Sticky bit
    - Logical OR of all bits right of round bit

Round	Sticky	Action
1	1	Up
1	0	Even
0	1	Down
0	0	Unchanged

Rounding

- IEEE 754
  - Error bounds
  - No more than $\frac{1}{2}$ "units of the last place" (ULP)] rules of rounding
  - Errors accumulate
    - Billions of FP ops in a single application
- Effects of limited precision
  - Commutativity, associativity etc. may no longer apply
  - A = (3.1415 + 6.022x10^{23}) - 6.022x10^{23}
  - B = 3.1415 + (6.022x10^{23} - 6.022x10^{23})
  - Is A==B?

No.

In A, π must align to give Avogadro No.

On aligning, π pretty much becomes 0.

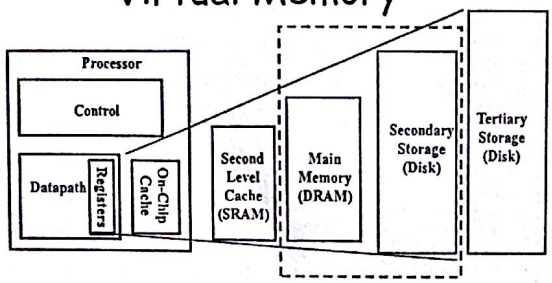
we have $6.022e23 - 6.022e23 = 0$.

In B, we get π as expected.

Order is imp.

This will happen ∵ of low precision.

Virtual Memory



- Data movement between Disk and Main memory
- We know how layers of hierarchy interact
  - Cache and main memory
- Can we apply all the same techniques?
  - Similarities and differences

VM : The real reason

- Application's view of memory
  - Large: > 4GB (32b address)
  - Exclusive: Only program in memory
- System view
  - Smaller: 512MB-2GB
  - Multiple programs share memory
  - Run in a protected manner (memory is private **)
  - Address range is fixed (starts at 0x0)
  - Do not bring in entire program
    - Bring in relevant parts as needed
- VM reconciles these conflicting "views"
  - Not just the classical benefits of hierarchy
  - Illusion of
    - speed of expensive level
    - capacity and cost of cheaper level

Virtual Memory

- Similarities:
  - Mapping a larger address space to a smaller space
  - Used for data movement between layers of the memory hierarchy
- Differences:
  - Miss-penalty : 10-100 cycles vs. ~1 million cycle
  - Block size : 16-64 bytes vs. 4 KB - 8 KB
  - Full associativity (But no associative search!)
  - Software handling

Fundamental

cache disk

↑ spacial locality

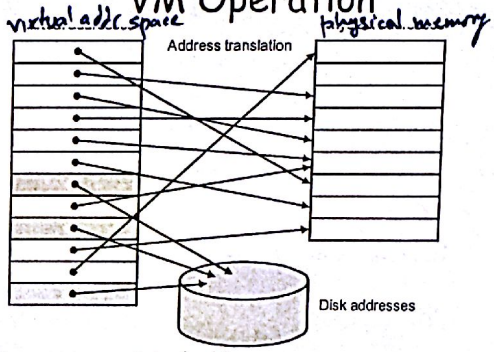
to reduce miss rate

no conflict misses

Cannot do old type of search ∵ of tooo many comparisons.

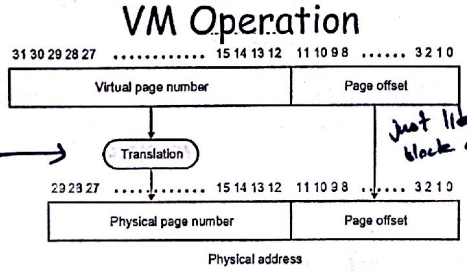
Blocks in VM are called pages. Same concept though.

VM Operation



- Programs use virtual address
- The data/code resides elsewhere (physical address)

VM Operation



- Assume 4KB pages
- 32-bit VA and 30-bit PA
- System responsible for translation
- E.g.
  - lw $r2, 0xffffc004
  - 0xffffc -> 0x20000 # VPN-> PPN translation
  - PA = 0x20000004

instead of tag comparison

just like block offset for D$ at end

∴ fully associate, no need of index bits.

VM Terminology

- Blocksize ~ 512B - 8KB+
  - Block : Page
- Miss : Page-fault ← naming convention.
  - Fetch from disk
- Derivative properties:
  - Fundamental constraint: high access latency

Back to 4 questions

- Q1. Where does a block go?
  - Fully associative
    - Block offset and tag
    - NO INDEX
  - Why?
    - AMAT = hit time + miss-rate*miss-penalty
    - Miss-penalty = ~1 million cycles
    - Have to minimize miss-rate

Q2. Block Identification

- Q2. Block identification
  - Fully associative search??
    - 30-bit physical address (1GB)
    - 4 KB pages
    - Number of frames = $2^{30}/2^{12} = 2^{18}$
    - 256 K frames
    - Compare 256 K frames in parallel??!!
  - Reframe question:
    - Old: Is this VA in any given frame? => Parallel search
    - New: Where is this VA? => Table lookup
      - Page table

Exercise: Virtual Memory Design

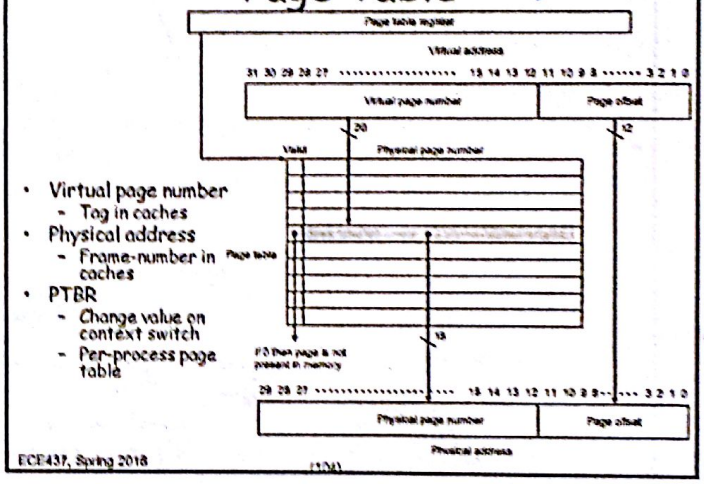
- Page size = 8KB
- Virtual Memory
  - 32b = 2^{32} = 4GB
- Physical memory = 512KB
- What is the size of the per-process page table?

lookup table / phone book.

- What happens on a context switch?

points to page tables of diff program

Page Table



- Virtual page number
  - Tag in caches
- Physical address
  - Frame-number in caches
- PTBR
  - Change value on context switch
  - Per-process page table

Page table

- Where does table reside?
  - Main memory
  - 100% overhead? → 1 to lookup addr. 1 to get value.
    - Each memory reference now generates two memory references
    - lw $r2,0xffff0004
      - Access page table entry for 0xffff0, get PPN
      - Access PPN:004
  - We want to minimize main memory access!
    - Page table entries can be cached like ordinary data
    - Tricky ***!!
    - Special cache for Page table entries

pid & lookup addr. in it.

Page Table Entries

- What does a page table entry contain
  - Physical page number (18 bits)
  - Access control (read/write permissions)
  - Valid bit (1 bit)
  - Misc
    - Use bit for replacement
    - Dirty bit for write-back
  - ~ 4 bytes (1 word)

Size of Page Table

- What is the size of the page table for a system with
  - 32 bit addresses
  - 1 GB physical memory
  - 4KB pagesize
- Virtual pages = $2^{32}/2^{12} = 2^{20}$
- Physical pages = $2^{30}/2^{12} = 2^{18}$
- PT size (per process) = (Entry size) * (# entries)
  = 4 bytes * 2^{20} = 2^{22} bytes = 4 MB
- # of processes? ~60 on my WinXP Pro machine
  - 240 MB for page tables?
  - "Big government" : 25% consumed for administration!!
  - Techniques to reduce overhead
    - Gradual growing
    - Inverted PT : entries per physical page

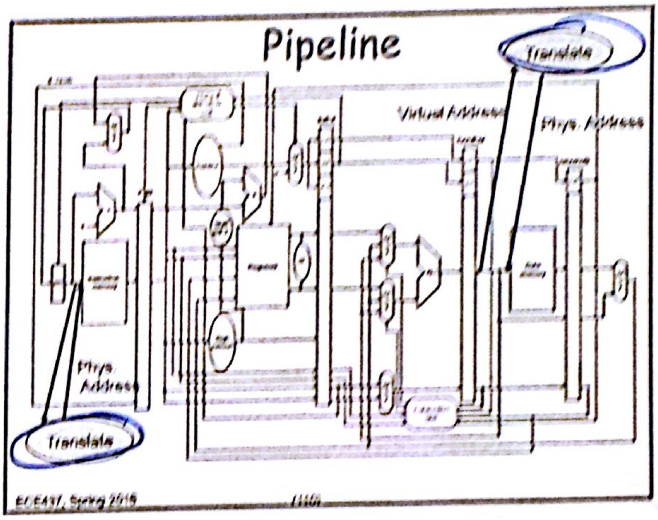
Replacement

- Q3. Block replacement
  - LRU and/or LRU approximation (FIFO with reference/use bits)
  - Sophisticated mechanisms possible (handling in software)
- Page-fault : Exception
  - Save instruction that causes fault
  - OS service the fault, i.e., brings in the relevant page from disk (VA → disk address??)
  - OS knows service is slow; schedule other program
  - When disk access is complete
  - Restart at offending instruction

Write handling

- Q4. What happens on a write
  - Write-through or write-back?

Pipeline



Page-fault throws exception.
Then software exception handles it.
↳ Here the disk work is going on.
Since it takes ~1 mil cycles, the OS gets someone else in & you are taking time.
↳ When ready, you come back & retry the same instruction.

Page table

- Where does table reside?
  - Main memory
  - 100% overhead?
    - Each memory reference now generates two memory references
    - lw $r2,0xffff0004
      - Access page table entry for 0xffff0, get PPN
      - Access PPN:004
  - We want to minimize main memory access!
    - Page table entries can be cached like ordinary data
    - Tricky ***!!
    - Special cache for Page table entries

Page Table Entries

- What does a page table entry contain
  - Physical page number (18 bits)
  - Access control (read/write permissions)
  - Valid bit (1 bit)
  - Misc
    - Use bit for replacement ← LRU/NRU types. To determine usage timing for replacement.
    - Dirty bit for write-back
  - ~ 4 bytes (1 word)

Size of Page Table

- What is the size of the page table for a system with
  - 32 bit addresses ✓
  - 1 GB physical memory ✓ 2^{30} B
  - 4KB pagesize ✓ 2^{12} B
  - Virtual pages = $2^{32}/2^{12} = 2^{20}$
  - Physical pages = $2^{30}/2^{12} = 2^{18}$
  - PT size (per process) = (Entry size) * (# entries)
    = 4 bytes * 2^{20} = 2^{22} bytes = 4 MB
- # of processes? ~60 on my WinXP Pro machine 4x60
  - 240 MB for page tables?
  - "Big government" : 25% consumed for administration!!
  - Techniques to reduce overhead
    - Gradual growing
    - Inverted PT : entries per physical page

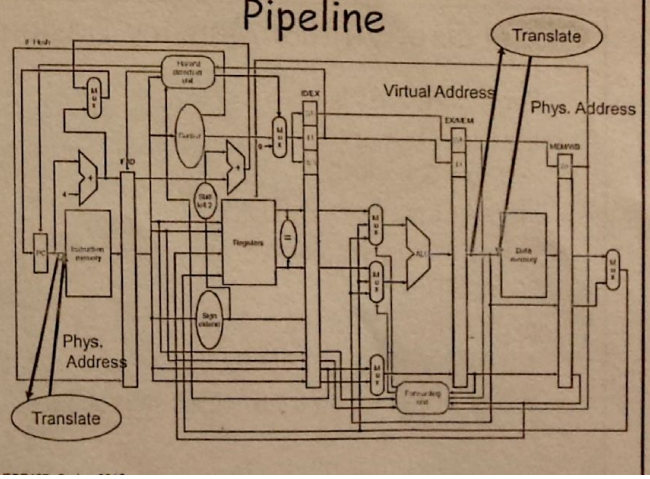
Replacement

- Q3. Block replacement
  - LRU and/or LRU approximation (NRU with reference/use bits)
  - Sophisticated mechanisms possible (handling in software)
- Page-fault : Exception
  - Save instruction that causes fault
  - OS service the fault, i.e., brings in the relevant page from disk (VA-> disk address??)
  - OS knows service is slow; schedule other program
  - When disk access is complete
  - Restart at offending instruction

Write handling

- Q4. What happens on a write
  - Write-through or write-back?

Pipeline



Memory Access Critical path

- Why use physical addresses to access caches?
  - Use virtual addresses
  - Faster : no translation
  - Block may have a different (still unique) tag and index
    - Who cares where the block resides in the cache?
- Synonyms
  - Two virtual pages map to same physical page
  - Should not be replicated in cache

ECE437, Spring 2016

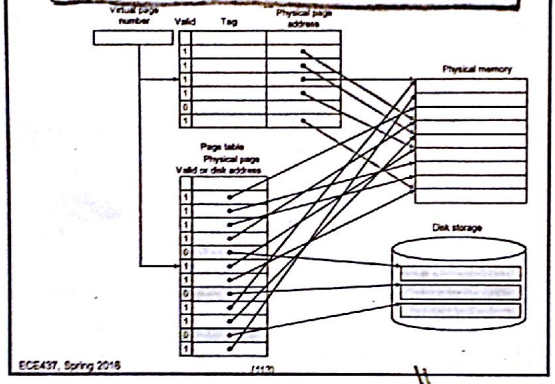
Faster Translation

- 100% overhead with VM system
- Eliminate memory access for translation
  - Caching
  - Translation lookaside buffer (TLB)
    - Also DTB in some literature
    - A cache of translations
  - 64-128 entries
    - Covers 256 KB ~512 KB
  - Organization
    - 64 entry fully associative
    - 256 entry, 16-way set associative
    - May be multi-level

ECE437, Spring 2016

TLB

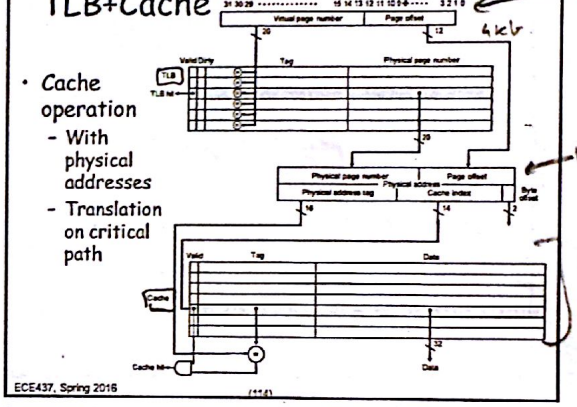
Translation Lookaside Buffer



ECE437, Spring 2016

TLB+Cache

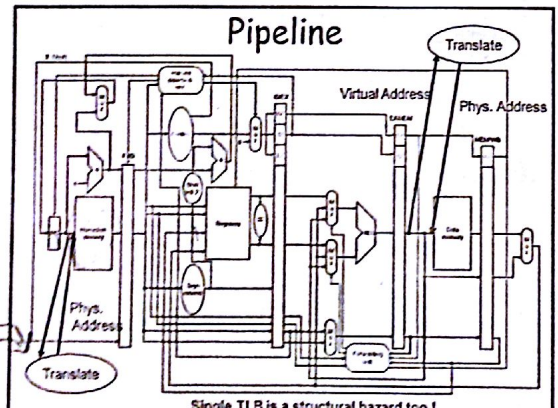
- Cache operation
  - With physical addresses
  - Translation on critical path



ECE437, Spring 2016

There is iTLB & dTLB to avoid structural hazard.

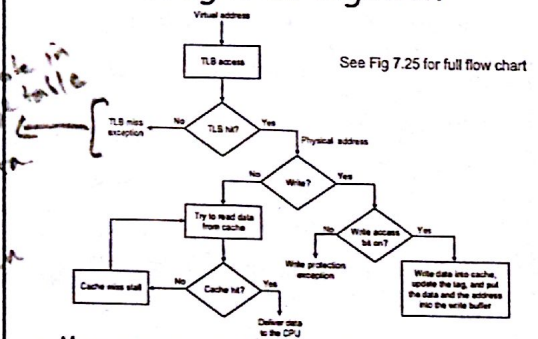
Pipeline



Single TLB is a structural hazard too !

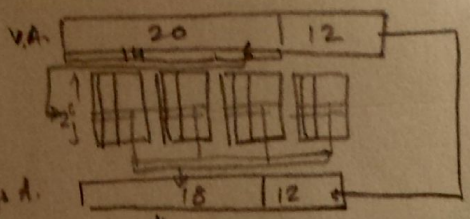
ECE437, Spring 2016

Putting it all together



- Memory access flowchart

ECE437, Spring 2016

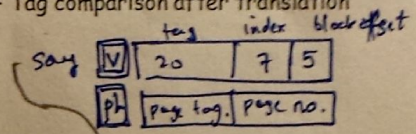


Full Hierarchy design

- 32b VA, 30 bit PA (i.e. 1GB Phys. Mem)
  - TLB
    - 256 entry, 4-way associative
  - Cache
    - 32KB, direct mapped, 32B blocks

Memory Access Critical Path

- Twist in the tale
  - Virtual-index
  - Physical tags
  - Indexing and translation proceeds in parallel
  - Tag comparison after translation



Summary

- 4Q on VM
  - Placement : fully associative
  - Identification : Page Table lookup
  - Replacement : LRU / LRU-approx
  - Writes : Writeback
- 4Q on TLB
  - Placement: small and fully associative, larger and set-associative
  - Identification: Associative search (CAM)
  - Replacement : random**
  - Writes : ?? Writes to TLB??

VM Miscellanea

- TLB: cache of VA-> translations
- On a context switch:
  - Change contents of PTBR for appropriate page table
  - What do we do with TLB contents?
    - Flush all entries
      - Simple, but inefficient
    - Associate Process ID with address
      - Flush required only when process IDs are reused

VM Miscellanea

- Memory efficiency of page tables
  - Limit register
    - Only region between PTBR and Limit is valid
    - Grow as needed
  - Segmented
    - Two page tables and two limit registers (Stack and Heap)
  - Inverted Page table
    - Hashing to map VA to a number within PA range
      - Hash 32-bit VA to 28 bit PA
      - Lookup complications
      - Collision
  - Multilevel page tables
  - Paging page tables

Real Stuff

Characteristic	Nehalem	Athlon 64x2
VA	64 bit	48 bit
PA	40 bit	40 bit
Page size	4KB, 2/4MB	4KB, 2/4 MB
TLB	Split I&D 4-way assoc Pseudo-LRU I-128+7, D-64+32 TLB miss H/W	Split, two level I&D L1 fully assoc (24I+40D) L2 4-way assoc (256I+256D) LRU/round-robin-LRU TLB miss H/W

Real Stuff

Characteristic	Nehalem	Athlon 64x2
L1 Cache	Split I & D	Split I & D
Size	32K + 32K	64K + 64K
Associativity	I 4-way; D 8-way	2-way
Hit time	3 cycles	3 cycles
Replacement	Approx LRU	LRU
Block	64 bytes	64 bytes
Write	Write-back	Write-back

- Pentium D had write-through L1

ECE437, Spring 2016 (123)

Real Stuff

Characteristic	Nehalem	Athlon 64x2
L2 Cache	Unified	Split I & D
Size	256KB (Private)	1MB+1MB
Associativity	8-way	16-way
Replacement	Approx LRU	Approx LRU
Block	64 bytes	64 bytes
Write	Write-back	Write-back
L3 cache	Unified (shared) 8MB	None (Phenom has it.)

ECE437, Spring 2016 (124)

eg.

Say Process B & Process Y have 8 bit of virtual address
Physical memory has 4 bit of address.

P0 asks for addr 0x10.
This value comes into RAM at say address 0x4.
In P0's page table, tag 0x10 has value of 0x4.

Page size comes into RAM (eg. 4kb data is read).

Now P1 asks for addr 0x08.
This value goes to 0xC in RAM.
P1's page table has 0xC at location 0x08.

Super pages
programmer can define larger page sizes.

P0 asks for addr 0x00.
RAM gets a page at addr 0x8.
P0's page table has 0x8 at location 0x00.

P1 asks for addr 0x14.
RAM gets a page at addr 0x0.
P1's page table has 0x0 at location 0x14.

P0 asks for 0x10
Physical addr of 0x4 given.
Data read & life goes on.

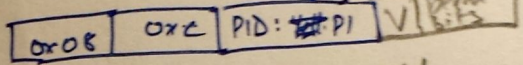
P1 asks for 0x04.
RAM is full. PAGE FAULT.
OS decides to get page at 0x4.
dirty bit maintained → Old page that was evicted is written to harddisk if it was modified.
The P0' page table has valid bit invalidated for RAM 0x4.
New page comes in.
P1's page table has 0x4 at location 0x04.

Cache has physical addr. So; it cannot be reached since lookup won't get location.

When doing this, the cache with that address has to be invalidated. cannot make shared physical ... lookup won't ... have that

Continuing but now with TLBs

P1 asks v.addr 0x08.
misses in TLB, get



Now physical addr 0xc is read from page table.

P0 asks v.addr 0x0C.
misses in TLB. Looks in page table [not there].
Say 0x0 [value of 0x14 of P1 is removes]. The Page table of P1 must be cleared.
P addr. caches too if need be.
Now 0x0C's value got into memory.
P0's page table gets 0x0 at 0x0C.

And so, life goes on...

Can avoid tag by clearing whenever Processor changes.

TLB also have cache like hierarchy goes down into L1 TLB (i & d).

Apr 18.

PA is constrained by TLB.

Inverted Page table
Use hash table to lookup physical addr for virtual addr.
Collision = open-addr forwarding
Now size matches PA (saves space)
Big improvement. But, to allow good hashing performance, space is 2-4x PA size.
In 64 bit architecture this is a good deal.

ECE437: Introduction to Digital Computer Design

Chapter 6

Spring 2016

Motivation

- I/O needed for
  - To/from users
  - To/from computers
  - To/from non-volatile storage media
- I/O Performance matters
- Total time = CPU + I/O - overlap
  - 10 + 4 - 4 = 10; 1x
  - 5 + 4 - [0,4] = [9,5]; [1.1x, 2x]
  - 1 + 4 - [0,1] = [5,4]; [2x, 2.5x]

slow improvement in I/O.

at best overlap all
at worst, see no overlap.

I/O Performance

- What is performance?
  - With CPU : Iron Law
  - With I/O
    - Applications have different I/O needs
- Supercomputers read and write 1G of data
  - High data throughput
- Transactions processing does many independent, small I/O ops.
  - Fast I/O throughput (# of I/Os per sec)
- File systems
  - Fast response time, locality

I/O Device Characteristics

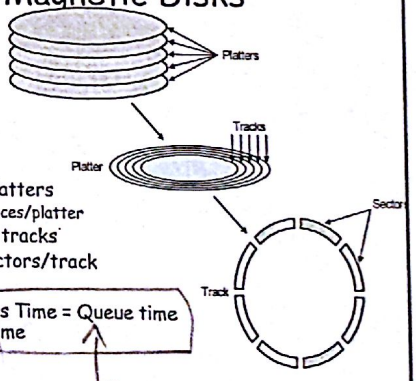
See Figure 6.2

Device	I or O?	Partner	Data Rate Mb/s
Mouse	I	Human	0.0038
Graphics Display	O	Human	800-8000
Modem	I/O	Machine	0.016-0.064
LAN	I/O	Machine	100-1000
Tape	Storage	Machine	32
Disk	Storage	Machine	240-2560

Disk Trends

- Disk Trends
  - $/MB decreasing
    - 2010: $125/2TB = 6.3c/GB
    - 2008: $149/1.5TB = 10c/GB
    - 2006: $92/250 GB = 36c/GB
    - 2004 : 50c/GB (was 10000c/GB in 1997)
    - OEM prices lower
  - Disk diameter 14" -> 1.8" -> 1" **
  - Seek time down ~5-10ms
  - Rotation speed unchanged
    - ~7200 rpm for consumer
    - ~15K rpm for high end disks
  - Xfer rates up
  - Move from parallel buses to Serial bus (Serial ATA)

Magnetic Disks

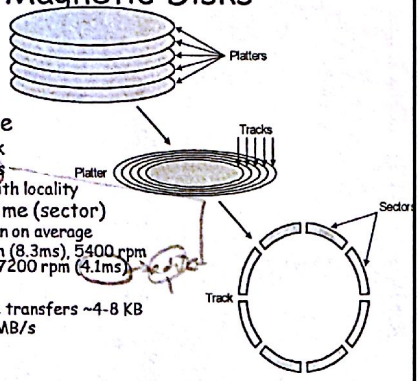


- Stack of platters
  - Two surfaces/platter
- Concentric tracks
- Several sectors/track
- Disk Access Time = Queue time + service time

pretty constant

can improve

Magnetic Disks



- Service time
  - Seek track
    - ~10-20ms
    - Better with locality
  - Rotation time (sector)
    - ½ rotation on average
    - 3600 rpm (8.3ms), 5400 rpm (5.6ms), 7200 rpm (4.1ms)
  - Xfer
    - Page size transfers ~4-8 KB
    - Several MB/s

Disk Access Time

- Disks R' us™ has a GenX disk with the following parameters
  - Average seek time = 10ms
  - Rotation speed = 5400 rpm
  - Xfer rate = 160MB/s
  - Disk-block size = 4KB
  - Controller overhead = 3ms ← Q time [time spent in disk's controller]
- They introduce a GenY disk technology with rotation speed of 7200 rpm which they advertise as being 50% faster
- If average queue time is 0 ms, what is the true speedup of GenY over GenX when reading a randomly chosen disk-block.

In Disk world
1K = 1000

Disk Access Time

- GenX access time =
  - Queuetime + controller overhead + seektime+ rotational latency for ½ rotation + Xfer time
  - $0 + 3 + 10 + (0.5*60/5400 \times 10^3)ms + (4K/160M \times 10^3)$ ms
  - 0+ 3 + 10 + 5.6 + 0.025
  - 18.625ms
- GenY access time
  - $0 + 3 + 10 + (0.5*60/7200 \times 10^3)ms + (4K/160M \times 10^3)$ ms
  - 0 + 3 + 10 + 4.1 + 0.25
  - 17.125 ms
- Speedup = 18.625/17.125 = 1.087 = 8.7%

transfer rate
Note how this does not improve access time.
Obj continuous read/write cares about Xfer rate

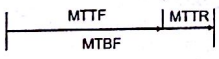
Exercise

- A disk has the following parameters
  - Access size : 8KB
  - Bandwidth: 80MB/s
  - Controller overhead : 0.1 ms
  - Seek time : 6ms
  - Rotation speed: 7200 rpm
- What is the average access time per transfer?
- If the disk controller has a 8MB cache that results in a 20% hit rate, what is the average access time? (Hits eliminate seek time and rotational latency)

Try.

I/O {Depend/Reli/Avail}-ability

failure



- MTTF Mean time to Failure
- MTTR Mean time to repair
- MTBF Mean time between Failure
  - MTTR + MTTF
- Reliability metric: MTTF
- Availability metric: MTTF/MTBF

Redundant Array of Inexpensive Disks

→ MTTF for 1 disk.

- What if we want to store 100 disks
- MTTF : 5 yr/100 = 18 days
  - RAID 0 = No redundancy (Striping)
  - RAID 1 = mirror = full data redundancy = 100% overhead
  - RAID 3 = bit interleaved parity = small overhead
    - Dedicated parity disk
  - RAID 4 = block interleaved parity = small overhead + small writes
  - RAID 5 = distributed block wise parity = small overhead + small writes
- Other raid levels in book

[Redundant array of inexpensive disks]

Apr 20.

3

Research impact
- top prof / conference.
- SOP now what interested
NOT what done in UG.

Redundant Array of Inexpensive Disks

- What if we want to store 100 disks
- MTTF : 5 yr/100 = 18 days
  - RAID 0 = No redundancy (Striping)
  - RAID 1 = mirror = full data redundancy = 100% overhead
  - RAID 3 = bit interleaved parity = small overhead
    - Dedicated parity disk
  - RAID 4 = block interleaved parity = small overhead + small writes
  - RAID 5 = distributed block wise parity = small overhead + small writes
- Other raid levels in book

Full copy of data

from 4 disks this raid holds parity bits.

improvement to raid 5.

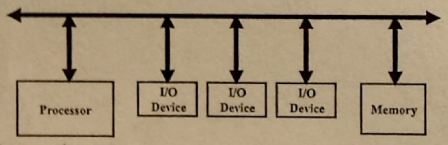
Recap

- Disks
  - Components of latency
  - Organization
  - RAID
  - Availability/Dependability/Reliability
- Buses
  - Types
    - One-bus/two-bus/three-bus architectures
  - Transaction timing
    - Synchronous vs. asynchronous

Connecting I/O to the CPU

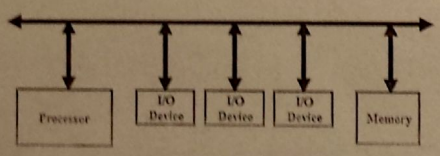
- Many I/O devices with vastly different datarates
- I/O device economics
  - Need standards
    - Large number of peripheral device producers
    - Multiple business entities involved favors longer lasting standards
- Let's examine several connection strategies

Advantages of Buses



- Versatility:
  - New devices can be added easily
  - Peripherals can be moved between computer systems that use the same bus standard
- Low Cost:
  - A single set of wires is shared in multiple ways
- Manage complexity by partitioning the design

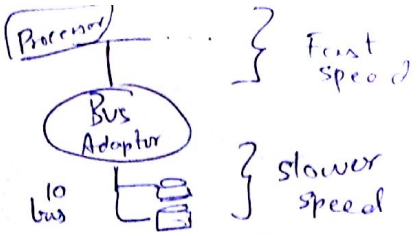
Disadvantages of Buses



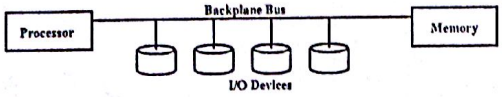
- It creates a communication bottleneck
  - The bandwidth of that bus can limit the maximum I/O throughput
- The maximum bus speed is largely limited by:
  - The length of the bus
  - The number of devices on the bus
  - The need to support a range of devices with:
    - Widely varying latencies
    - Widely varying data transfer rates

Types of buses

- Processor-Memory Bus (design specific/proprietary)
  - Short and high speed
  - Only need to match the memory system
    - Maximize memory-to-processor bandwidth
  - Connects directly to the processor
  - Optimized for cache block transfers
- I/O Bus (industry standard)
  - Usually is lengthy and slower
  - Need to match a wide range of I/O devices
  - Connects to the processor-memory bus or backplane bus
- Backplane Bus (standard or proprietary)
  - Backplane: an interconnection structure within the chassis
  - Allow processors, memory, and I/O devices to coexist
  - Cost advantage: one bus for all components

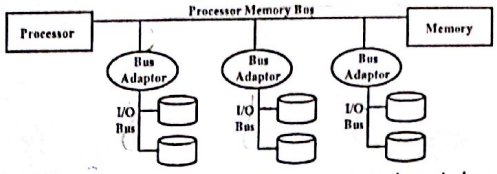


One Bus (Backplane) Architecture



- A single bus (the backplane bus) is used for:
  - Processor to memory communication
  - Communication between I/O devices and memory
- Advantages: Simple and low cost
- Disadvantages: slow and the bus can become a major bottleneck
- Example: IBM PC - AT

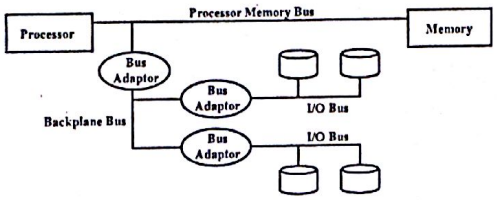
Two Bus System



- I/O buses tap into the processor-memory bus via bus adaptors:
  - Processor-memory bus: mainly for processor-memory traffic
  - I/O buses: provide expansion slots for I/O devices
- Apple Macintosh-II
  - NuBus: Processor, memory, and a few selected I/O devices
  - SCCI Bus: the rest of the I/O devices

Now fewer things hanging off the bus.

Three Bus System



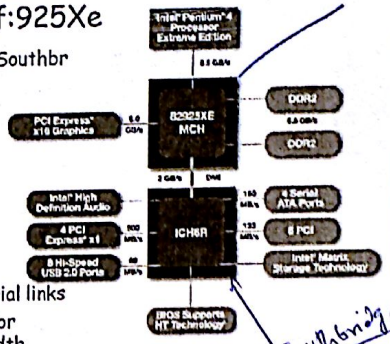
- A small number of backplane buses tap into the processor-memory bus
  - Processor-memory bus is used for processor memory traffic
  - I/O buses are connected to the backplane bus
- Advantage: loading on the processor bus is greatly reduced

AGP: accelerated graphics port.

Northbridge high speed bus.

Real Stuff:925Xe

- Northbridge/Southbridge
- NB for high bandwidth
- SB for lower bandwidth

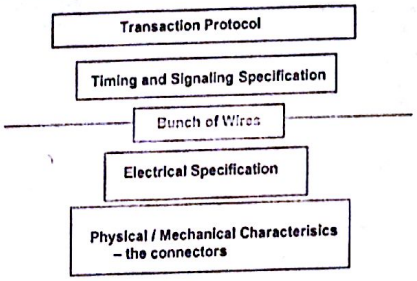


- PCI-e uses serial links
- Ganged links for higher bandwidth

Graphic from Intel.com

Southbridge low speed bus.

Bus Definition



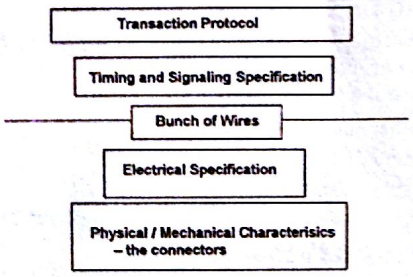
Synchronous vs. Asynchronous

- Synchronous Bus:
  - Includes a clock in the control lines
  - A fixed protocol for communication that is relative to the clock
  - Advantage: involves very little logic and can run very fast
  - Disadvantages:
    - Every device on the bus must run at the same clock rate
    - To avoid clock skew, they cannot be long if they are fast
- Asynchronous Bus:
  - It is not clocked
  - It can accommodate a wide range of devices
  - It can be lengthened without worrying about clock skew
  - It requires a handshaking protocol

Apr. 22:

2.

Bus Definition



ECE437, Spring 2016

Synchronous vs. Asynchronous

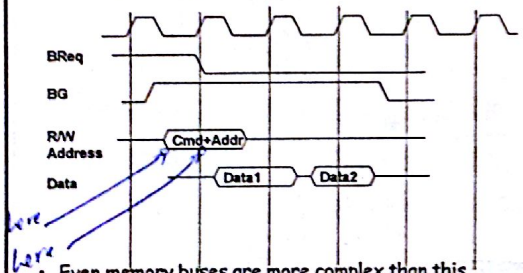
- Synchronous Bus:
  - Includes a clock in the control lines
  - A fixed protocol for communication that is relative to the clock
  - Advantage: involves very little logic and can run very fast
  - Disadvantages:
    - Every device on the bus must run at the same clock rate
    - To avoid clock skew, they cannot be long if they are fast
- Asynchronous Bus:
  - It is not clocked
  - It can accommodate a wide range of devices
  - It can be lengthened without worrying about clock skew
  - It requires a handshaking protocol

→ minimizes handshakes but not rule it out (e.g. wait signal)

ECE437, Spring 2016

operation happens on each rising edge.

Simple Synchronous Protocol

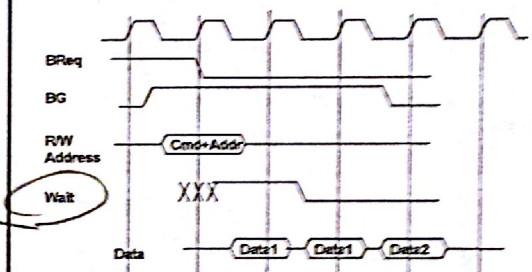


ready here / here / read

- Even memory buses are more complex than this
  - memory may take time to respond
  - need to control data rate

ECE437, Spring 2016

Typical Synchronous Protocol

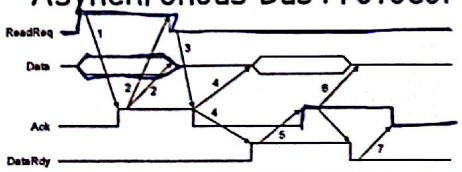


- Device indicates when it is prepared for data xfer
- Actual transfer goes at bus rate

ECE437, Spring 2016

↑ to read / memory

Asynchronous Bus Protocol



Proc asserts READREQ, <address>	
	Mem asserts ACK after reading <address>
Proc deasserts READREQ, <address> and waits	
	Mem deasserts ACK
	Mem asserts DATA-RDY, <data>
Proc asserts ACK after reading <data>	
	Mem deasserts DATARDY, <data>
Proc deasserts ACK	

ECE437, Spring 2016

→ Need a lot of requests / ready flags & acknowledgements to the same.

glue: processor block: memory

I/O Interfacing: Outline

- How does the CPU initiate I/O?
- How does I/O device notify CPU when I/O op is complete?
- Who (on the CPU) performs I/O ops?

ECE437, Spring 2016

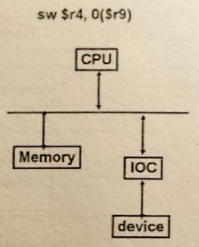
I/O Interfacing

- I/O operation needs to be initiated
  - Special opcodes → used to be done
  - Memory-mapped I/O → this is standard today
- I/O completion must be known
  - Polling
  - Interrupt-driven

communicate via lw & sw on common memory.

I/O commands

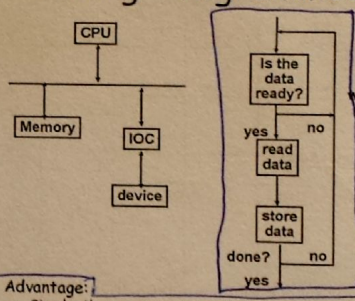
- Memory-mapped I/O
  - Special addresses not for memory ($r9 contains special address for IOC)
  - Commands written (sw) as data ($r4)
- I/O commands
  - Special opcodes
  - Send over I/O Bus



I/O Device Notifying the OS

- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- This can be accomplished in two different ways:
  - Polling:
    - The I/O device put information in a status register
    - The OS periodically check the status register
  - I/O Interrupt:
    - Whenever an I/O device needs attention from the processor, it interrupts the processor from what it is currently doing.

Polling: Programmed I/O

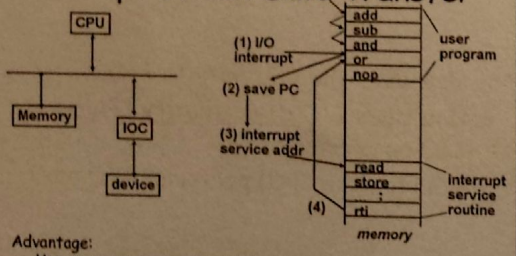


busy wait loop not an efficient way to use the CPU unless the device is very fast!

but checks for I/O completion can be dispersed among computation intensive code

- Advantage:
  - Simple: the processor is totally in control and does all the work
- Disadvantage:
  - Polling overhead can consume a lot of CPU time

Interrupt Driven Data Transfer



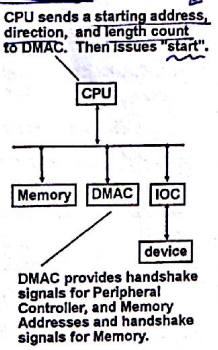
- Advantage:
  - User program progress is only halted during actual transfer
- Disadvantage: special hardware is needed to:
  - Cause an interrupt (I/O device)
  - Detect an interrupt (processor)
  - Save the proper states to resume after the interrupt (processor)

I/O Interrupts

- An I/O interrupt is just like the exceptions except:
  - An I/O interrupt is asynchronous
  - Further information needs to be conveyed
- An I/O interrupt is asynchronous with respect to instruction execution:
  - I/O interrupt is not associated with any instruction
  - I/O interrupt does not prevent any instruction from completion
    - You can pick your own convenient point to take an interrupt
- I/O interrupt is more complicated than exception:
  - Needs to convey the identity of the device generating the interrupt
  - Interrupt requests can have different urgencies:
    - Interrupt request needs to be prioritized

Direct Memory Access (DMA)

- Direct Memory Access (DMA):
  - External to the CPU
  - Act as a master on the bus
  - Transfer blocks of data to/from memory without CPU intervention



master → initiates bus operations.

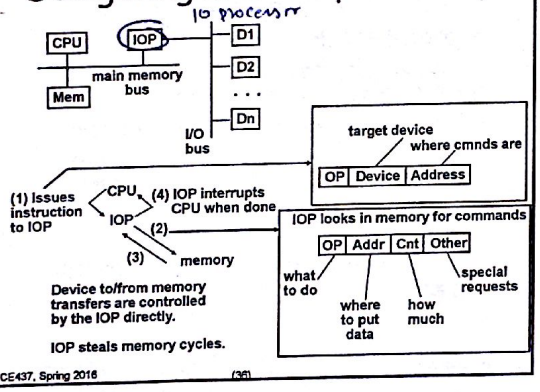
but this is virtual addr. Need to go to TLB [or interrupt CPU for TLB lookup] or read from page table to get physical addr.

DMA interfacing

- DMA : virtual addresses or physical addresses?
- Challenge: crossing page boundaries
  - Virtual addresses
    - Translation provided by OS
  - Physical addresses
    - One page per transfer
    - OS chains the physical addresses
- No page faults in between
  - "Pin" pages

if it was physical addr, had to get complete block. Also, a page might be moved out from mem. To prevent this, the page table value must be "pinned" [unremovable].

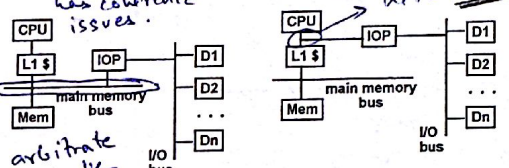
Delegating I/O Responsibility



CPU sets instr. in memory & tells IOP to go to run the IO Processor. at that instr. set. The IOP works & tells CPU that it is done.

The Coherence Problem

has coherence issues.



here need to arbitrate. ↓ hit time.

need to arbitrate for IOP. It can be like a processor but not done so as it has no cache & usually does big writes.

- Where is the cache w.r.t. I/O device (and/or I/O processor)?
  - In front of cache : Slows down CPU
  - Behind Cache : Cache coherence
- Applies to any I/O device that writes autonomously to memory
  - DMA, other
  - Multiprocessors also
- Coherence is a problem when both the following are involved:
  - Multiple writers AND
  - Replication

Arbiter can allow bulk read/write permission.

CPU involvement

- Low offloading to high offloading
  - Polling
    - CPU does everything including detecting events and subsequent I/O copying to/from memory
  - Interrupts
    - Someone else notifies CPU when processing needed, CPU does the needful
  - DMA
    - CPU describes what needs to be done, DMAC does the rest
  - IOP
    - CPU tells IOP where to find instructions on what needs to be done

max work in CPU

min work in CPU

OS Responsibilities

- The operating system acts as the interface between:
  - The I/O hardware and the program that requests I/O
  - Why?
    - Why not let user programs directly read/write to I/O devices?
- Three characteristics of the I/O systems:
  - The I/O system is shared by multiple program using the processor
  - I/O systems often use interrupts (external generated exceptions) to communicate information about I/O operations.
    - Interrupts must be handled by the OS because they cause a transfer to supervisor mode
  - The low-level control of an I/O device is complex:
    - Managing a set of concurrent events
    - The requirements for correct device control are very detailed

OS Responsibilities

- Provide protection to shared I/O resources
  - Guarantees that a user's program can only access the portions of an I/O device to which the user has rights
- Provides abstraction for accessing devices:
  - Supply routines that handle low-level device operation
- Handles the interrupts generated by I/O devices
- Provide equitable access to the shared I/O resources
  - All user programs must have equal access to the I/O resources
- Schedule accesses in order to enhance system throughput

life cycle of IO devices

Interfacing: Summary

- Multiprogramming
  - Program invokes syscall (i.e., invokes OS)
  - OS checks protection [cannot run root w/o password]
  - OS runs device drivers
  - Suspends current process and switches process
  - I/O interrupt fielded by OS
  - OS completes I/O and wakes up suspended process (i.e. make it runnable)
  - Run next ready process

Outline

- Disks
- Buses
  - Performance vs. Simplicity tradeoff
  - Timing
- Interfacing
  - OS vs. user mode, virtualized interface
- Networks
- Video
- Optical drives

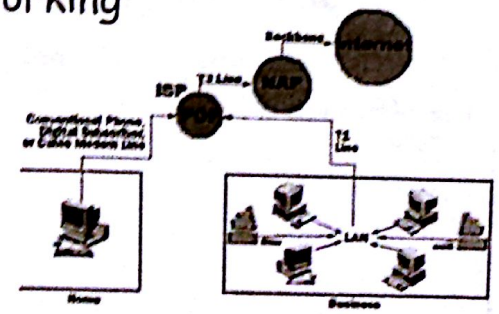
Networking I/O Issues

- Protocol Stack
  - Application: sends a HTML file
  - HTML file sent over TCP/IP
  - TCP breaks up html file into IP packets
  - Sends IP packets (and ensures in-order delivery, retransmitting if necessary)
  - IP packets may further be broken up into Ethernet/802.11b frames
- Checksums/ error handling at several layer

Networking I/O issues

- I/O Issues
- Partitioning this stack
  - What gets done on the CPU?
  - What gets done on other hardware?
- TCP offload engines (TOE) ← hardware to check TCP packets.
  - "smart NIC" handles more of TCP stack functionality
  - Relevant at GigE+ speeds
- Other functionality may also be offloaded
  - Firewall/VPN/filtering/IDS

Networking



- WAN
  - Backbone
- LAN
  - Ethernet
  - WLAN (802.11b)

Ethernet

- One shared medium
  - Distributed arbitration by collision detection
  - Officially : CSMA/CD : carrier-sense multiple access with collision detection
  - Length limits (why?)
  - Shared medium is bottleneck
- "The rumors of my demise have been greatly exaggerated." -- Mark Twain
- But wait... what survived is not the same
- Evolving
  - 100M Bit ethernet, Gigabit ethernet, 10Gb Ethernet
  - Now a switched network standard
    - Not shared bus, point-to-point network

ECE437, Spring 2016 (46)

NOT bus network. No arbiter

carrier sense : every one can read off wire.
multiple access : every one can write on wire.
collision detect : as you send, you listen. If someone else sends too, you can detect and act based on that.

Increase of collisions, exponential back off
wait- rand(10)
if still collides, rand(100)
⋮

compatibe

802.11b/g/n

- Also distributed arbitration
- CSMA/CA : carrier-sense multiple access with collision avoidance
- To minimize collision - Combination of
  - Handshakes (RTS/CTS) and
  - Backoff try to send → dr. to send
- Shared medium is the 802.11b/g/n spectrum (near 2.4GHz)
- Cannot detect collision
  - Limitation of RF - turn off receiver when transmitting
- Destination has to send an acknowledgement
  - No Ack -> collision and/or RF interference

ECE437, Spring 2016 (47)

Cannot detect collision:
ie. cannot read & write together.

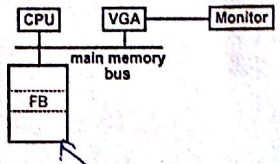
So, collision avoidance

send RTS.
If you get CTS, then write/send message.
else, wait & try later.

Apr. 25.

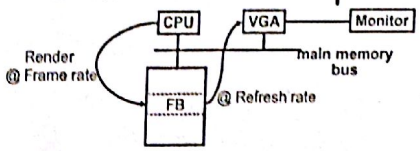
back when IBM PC & Bill gates

Video



- Frame-buffer
  - Memory representation of monitor contents
  - Color of each pixel
    - E.g., 8 bits/ color = 24 bits/pixel
    - # of colors = 256 x 256 x 256 = 16,777,216 = 16M colors

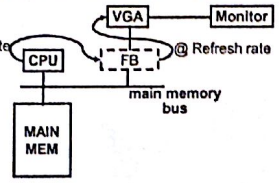
"Dumb" Video Adapter



- CPU responsible for all data conversion from internal symbolic data-structures to final pixel representation
  - Rendering
  - Frame rate
- Video adapter grabs data from frame buffer
  - Refresh rate
- Bandwidth:
  - 24 bit color, 1280x1024 pixels, 75Hz refresh rate
  - 3 x 1280 x 1024 x 75 = 554 MB/s
- On memory bus
  - Structural hazard ← CPU & VGA fight for memory
  - CPU and Video adapter competing for memory bandwidth

Optimization #1

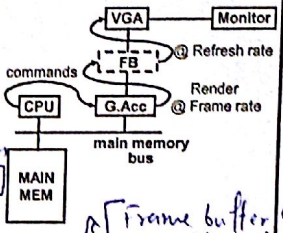
VGA now not on memory bus.



- VRAM (Video RAM)
  - Eliminate structural hazard
  - Dual ported
    - Normal DRAM port ← for CPU
    - Serial port ← for VGA to read serially.
      - Raster scan → read in serial (row) order
      - Natural because DRAM read in rows
      - Latch an entire row in a shift-register

Optimization #2

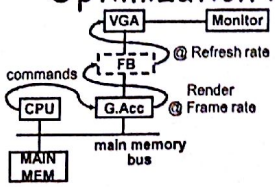
- Improve Frame rate
  - Smart video accelerators
    - Graphics co-processor
    - Offload rendering (at least significant fractions of it)
    - Not just less work, more power-efficient too
- Fringe benefit
  - Massive reduction in bandwidth ↓ main memory bus
  - CPU not writing frame buffer
  - CPU issues graphics commands through API
    - API's such as OpenGL, Direct3D etc.



Frame buffer (FB) completely hidden from CPU.

CPU now calls api (eg OpenGL) and the G. Accelerator converts to 2D & saves to the FB.

Optimization #3



- How OpenGL/Direct3D is implemented on the Graphics Co-processor:
- Interface vs. Implementation
  - API specifies interface
  - Software implementation
    - Runs on Graphics processor (GPU)
  - Hardware implementations
    - Support for clipping, rendering, transformations etc

Other Accelerators

- Accelerators
  - Per application
    - openGL accelerators for 3D rendering
    - Mpeg decoders for dvd (movie) rendering
  - Broader applicability
    - GPGPUs → allow use of graphics hardware (really, SIMD parallelism) for other computation
      - Subsumes graphics accelerators
      - New programming interfaces
    - FPGA
      - Virtual processors with parametric pipeline width
      - Compile C to hardware

not graphic only.

synthesize code to hardware & run here at hardware speed.

What is Performance for us?

- For computer architects
  - CPU execution time = time spent running a program
- Because people like faster to be bigger to match intuition
  - performance = 1/X time
  - where X = response, CPU execution, etc.
- Elapsed time = CPU execution time + I/O wait
- We will concentrate mostly on CPU execution time

Improve Performance

- Improve (a) response time or (b) throughput?
  - faster CPU
    - both (a) and (b)
  - Add more CPUs
    - (b) but (a) may be improved due to reduced queueing

Give an example of this phenomenon

Performance Comparison

- Machine A is n times faster than machine B iff
  - perf(A)/perf(B) = time(B)/time(A) = n
- Machine A is x% faster than machine B iff
  - perf(A)/perf(B) = time(B)/time(A) = 1 + x/100
- E.g., A 10s, B 15s
  - 15/10 = 1.5 => A is 1.5 times faster than B
  - 15/10 = 1 + 50/100 => A is 50% faster than B

Breaking down Performance

- A program is broken into instructions
  - H/W is aware of instructions, not programs
- At lower level, H/W breaks instructions into cycles
  - lower level state machines change state every cycle
- E.g., 4 GHz Opteron runs 4 B cycles/sec, 1 cycle = 0.25 ns

Iron law

Time/program = instrs/program x cycles/instr x sec/cycle

- sec/cycle (a.k.a. cycle time, clock time) - 'heartbeat' of computer
  - mostly determined by technology and CPU organization
- cycles/instr (a.k.a. CPI)
  - mostly determined by ISA and CPU organization
  - overlap among instructions makes this smaller
- instr/program (a.k.a. instruction count)
  - instrs executed NOT static code
  - mostly determined by program, compiler, ISA

Our Goal

- Minimize time which is the product, NOT isolated terms
- Common error to miss terms while devising optimizations
  - E.g., ISA change to decrease instruction count
  - BUT leads to CPU organization which makes clock slower

Iron Law Example

- Machine A: clock 1 ns, CPI 2.0, for a program
- Machine B: clock 2 ns, CPI 1.2, for same program
- Which is faster and how much?
- Time/program = instrs/program x cycles/instr x sec/cycle
  - Time(A): N x 2.0 x 1 = 2N
  - Time(B): N x 1.2 x 2 = 2.4N
- Compare: Time(B)/Time(A) = 2.4N/2N = 1.2
- So, Machine A is 20% faster than Machine B for this program

Iron Law Example

- Keep clock of A at 1 ns and clock of B at 2 ns
- For equal performance, if CPI of B is 1.2, what is A's CPI?
  - Time(B)/Time(A) = 1 = (N x 2 x 1.2)/(N x 1 x CPI(A))
  - CPI(A) = 2.4

Iron Law Example

- Let CPI of A = 2.0 and CPI of B = 1.2
- For equal performance, if clock of B is 2 ns, what is A's clock?

Iron Law Example

- Let CPI of A = 2.0 and CPI of B = 1.2
- For equal performance, if clock of B is 2 ns, what is A's clock?
  - Time(B)/Time(A) = 1 = (N x 2.0 x clock(A))/(N x 1.2 x 2)
  - clock(A) = 1.2 ns

Other Metrics

- MIPS and MFLOPS
- MIPS
  = instruction count/(execution time x 10^6)
  = clock rate/(CPI x 10^6) (How?)
- BUT MIPS has problems

$$MIPS = \frac{I}{\#clk \times Clk\,time \times 10^6} = \frac{1}{CPI \times Clk\,time \times 10^6}$$

not considers instruction count directly

Problems with MIPS

- E.g., without FP H/W, an FP op may take 50 single-cycle instrs
- with FP H/W only one 2-cycle instr
- Thus adding FP H/W
  - CPI increases (why?) The FP op goes from 50/50 to 2/1
  - but instrs/prog decreases more (why?) each of the FP op reduces from 50 to 1, factor of 50
  - total execution time decreases
- For MIPS
  - instrs/prog ignored
- MIPS gets worse!

Problems with MIPS

- Ignore program
- Usually used to quote peak performance
  - ideal conditions => guarantee not to exceed!!
- When is MIPS ok?
  - same compiler and same ISA
  - e.g., same binary running on Pentium Pro and Pentium
  - why? instrs/prog is constant and may be ignored

Other Metrics

- PFLOPS = FP ops in program/(execution time x 10^12)
- Assuming FP ops independent of compiler and ISA
  - Assumption not true
  - may not have divide instruction in ISA
  - optimizing compilers can remove some insts
- Relative MIPS and normalized MFLOPS
  - adds to confusion!

Rules

- Use ONLY Time
  - Beware when reading, especially if details are omitted
  - Beware of Peak

Outline

- Time and performance
- Iron law
- MIPS and MFLOPS
- Which programs
- How to average
- Amdahl's law

Which Programs

- Execution time of what
- Best case - you run the same set of programs everyday
  - port them and time the whole "workload"
- In reality, use benchmarks
  - programs chosen to measure performance
  - predict performance of actual workload (hopefully)
  - saves effort and money
  - representative? honest?

Benchmarks: SPEC2006

- SPEC: System Performance Evaluation Cooperative
- Latest is SPEC2006, before SPEC89, SPEC92, SPEC95, SPEC 2000
- 12 integer and 18 floating point programs
  - normalize run time with a Gold Standard processor (*SPEC ratio*)
  - GM of the normalized times (why GM?)

Recap Exercise

- 60% instructions run in 1 ns and 40% instructions need 1.45 ns
  - Machine A: Cycletime = 1.45 ns
    - All instructions execute in 1 cycle
  - Machine B: Cycle time = 1 ns
    - 60% instructions execute in 1 cycle
    - 40% instructions execute in 2 cycles
  - Which is the better architecture? And by what factor/percentage?

How to average

- Example

	Machine A	Machine B
Program 1	1s	10s
Program 2	1000s	100s
Total	1001s	110s

- One answer: total execution time, then how much faster than A is B? 1001/110 = 9.1

assuming P1 & P2 are equally weighted.

How to average

- Another: arithmetic mean (same result: B 9.1 times faster than A)
- Arithmetic mean of times: $\left\{\sum_{i=1}^{n} time_i\right\}/n$ for n programs
- AM(A) = 1001/2 = 500.5
- AM(B) = 110/2 = 55
- 500.5/55 = 9.1
- Valid only if programs run equally often, else use "weight" factors
- Weighted arithmetic mean: $\left\{\sum_{i=1}^{n} weight_i \times time_i\right\}/n$

Other Averages

- E.g., 30 mph for first 10 miles
- 90 mph for next 10 miles. Average speed?
- Average speed = (30+90)/2 =60mph? WRONG
- Average speed = total distance / total time
  = (20 / (10/30+10/90))
  = 45 mph
- What if it was 10 hours at each speed?
  - instead of 10 miles

rate

for rates

Harmonic Mean

- Harmonic mean of rates = $\dfrac{1}{\left\{\sum_{i=1}^{n} \frac{1}{rate_i}\right\}/n}$
  - Use HM if forced to start and end with rates
- Trick to do arithmetic mean of times but using rates and not times

Practice

- Machine A runs 10M instructions at 15 MIPS and the next 10M instructions at 45 MIPS
  - Average MIPS = ?? harmonic mean
- Machine A runs for 10 seconds at 15 MIPS and the next 10 seconds at 45 MIPS
  - Average MIPS = ?? normal mean

eg.

$\frac{1}{rate}$ = time

time can average.
the avg. time is reciprocated for avg. rate.

SPEC normalized run time with a Gold Standard processor (SPEC Ratio)
Due to this, must make geometric mean.

Dealing with Ratios

- Absolute times

	Machine A	Machine B
Program 1	1s	10s
Program 2	1000s	100s

- Now consider ratios (w.r.t. A)

	Machine A	Machine B
Program 1	1	10
Program 2	1	0.1

- Averages: A = 1, B = 5.05

Dealing with Ratios

- Absolute times

	Machine A	Machine B
Program 1	1s	10s
Program 2	1000s	100s

- Now consider ratios (w.r.t. B)

	Machine A	Machine B
Program 1	0.1	1
Program 2	10	1

- Averages: A = 5.05, B = 1
- Both cannot be true

Geometric Mean

- Don't use arithmetic mean on ratios (normalized numbers)
- Use geometric mean for ratios
  - geometric mean of ratios = $\sqrt[n]{\prod_{i=1}^{n} ratio_i}$
  - Use GM if forced to use ratios
- Independent of reference machine (math property)
- In the example, GM for machine A is 1, for machine B is also 1
- normalized with respect to either machine

But..

- Geometric mean of ratios is not proportional to total time
- AM in example says machine B is 9.1 times faster
- GM says they are equal
- If we took total execution time, A and B are equal only if
  - program 1 is run 100 times more often than program 2
- Generally, GM will mispredict for three or more machines

Previous Midterm Question

- Machine A runs 9 times faster than machine B when running program-P,
- Machine B runs 4 times faster than machine A when running program Q.
- Which machine is faster (and by what factor) when averaged across both programs?

	A	B
P	1	9
Q	1	1/4
GM	1	1.5

∴ times.
A better than B by: 1.5/1 (factor)

Summary for Averages

- Use AM for times
- Use HM if forced to use rates
- Use GM if forced to use ratios
- Better yet, use unnormalized numbers to compute time

when a feature is made faster by a factor (F) and others are same.

major pitfall

Amdahl's Law

- Why does the common case matter the most?
- Let an optimization speed f fraction of time by a factor of s
- assuming that old time = T, what is the speedup?
  - f is the "affected" fraction of T
  - (1-f) is the unaffected fraction

$$\text{Speedup} = \frac{time_{old}}{time_{new}} = \frac{unaffected_{old} + affected_{old}}{unaffected_{new} + affected_{new}}$$

$$= \frac{(1-f)\times T + f\times T}{(1-f)\times T + \frac{f}{s}\times T}$$

$\frac{1}{(1-f)+\frac{f}{s}}$

Amdahl's Law Example

ratio amount of improvement

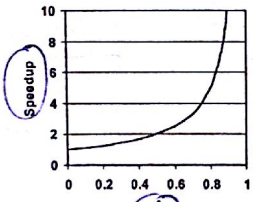
- Your boss asks you to improve processor performance
- Two options: What should you do?
  - improve the ALU used 95% of time, by 10%
  - improve the square-root unit used 5%, by a factor of 10

f	s	Speedup
95%	1.10	1.094
5%	10	1.047
5%	∞	1.052

Amdahl's Law: Limit

- Make common case fast because:

$$\lim_{s\to\infty}\left(\frac{1}{1-f+f/s}\right) = \frac{1}{1-f}$$



for small 'f', speedup is small even if ∞ factor scale up.

Amdahl's Law

- "Make common case fast"
  - Heuristic, not commandment
  - Use for intuition, verify with numbers
- 60% can be improved by a factor of 2
  - Speedup = 1/(0.4+0.6/2) = 1/0.7
- 40% can be improved by a factor of 8
  - Speedup = 1/(0.6+0.4/8) = 1/0.65
- Second option is better
  - Less common case, but higher speedup compensates

Summary

- Time and performance:
  - Machine A n times faster than Machine B
  - iff Time(B)/Time(A) = n
- Iron Law: Time/prog
  - Instr count x CPI x Cycle time
- Other Metrics: MIPS and MFLOPS
  - Beware of peak and omitted details
- Benchmarks: SPEC2006 (int + FP)
- Summarize performance:
  - AM for time, HM for rate, GM for ratio
- Amdahl's Law: Speedup = $\left(\frac{1}{1-f+f/s}\right)$ common case fast