Low-Level Programming

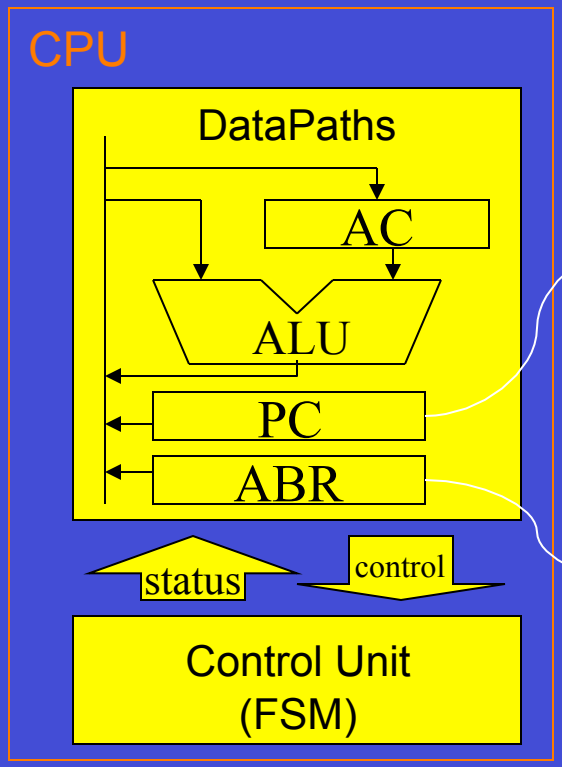
ICOM 4036
Lecture 4

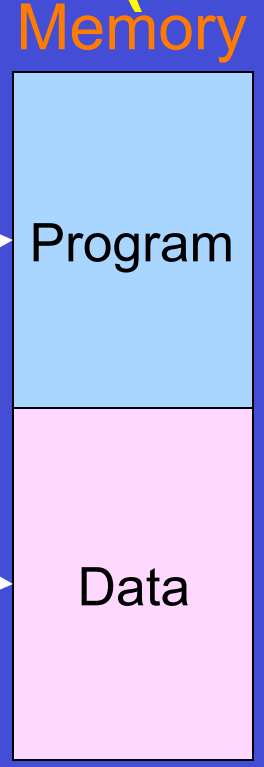
Prof. Bienvenido Velez

Practical Universal Computers

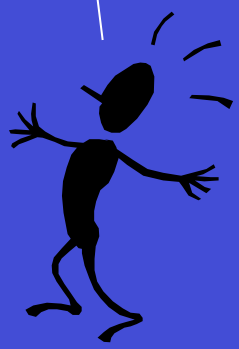
(John) Von Neumann Architecture (1945)


John von Neumann, 1950's





This looks just like a TM Tape



CPU is a universal TM

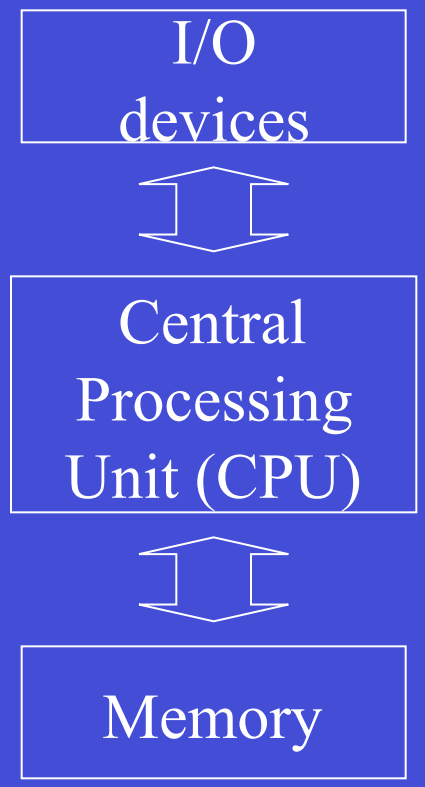
An interpreter of some programming language (PL)

Outline

- The Von Neumann Architecture
- From Voltages to Computers
- Low-level Programming
- Implementing HLL Abstractions
  - Control structures
  - Data Structures
  - Procedures and Functions

The (John) Von Neumann Architecture
(late 40's)


John von Neumann, 1950's



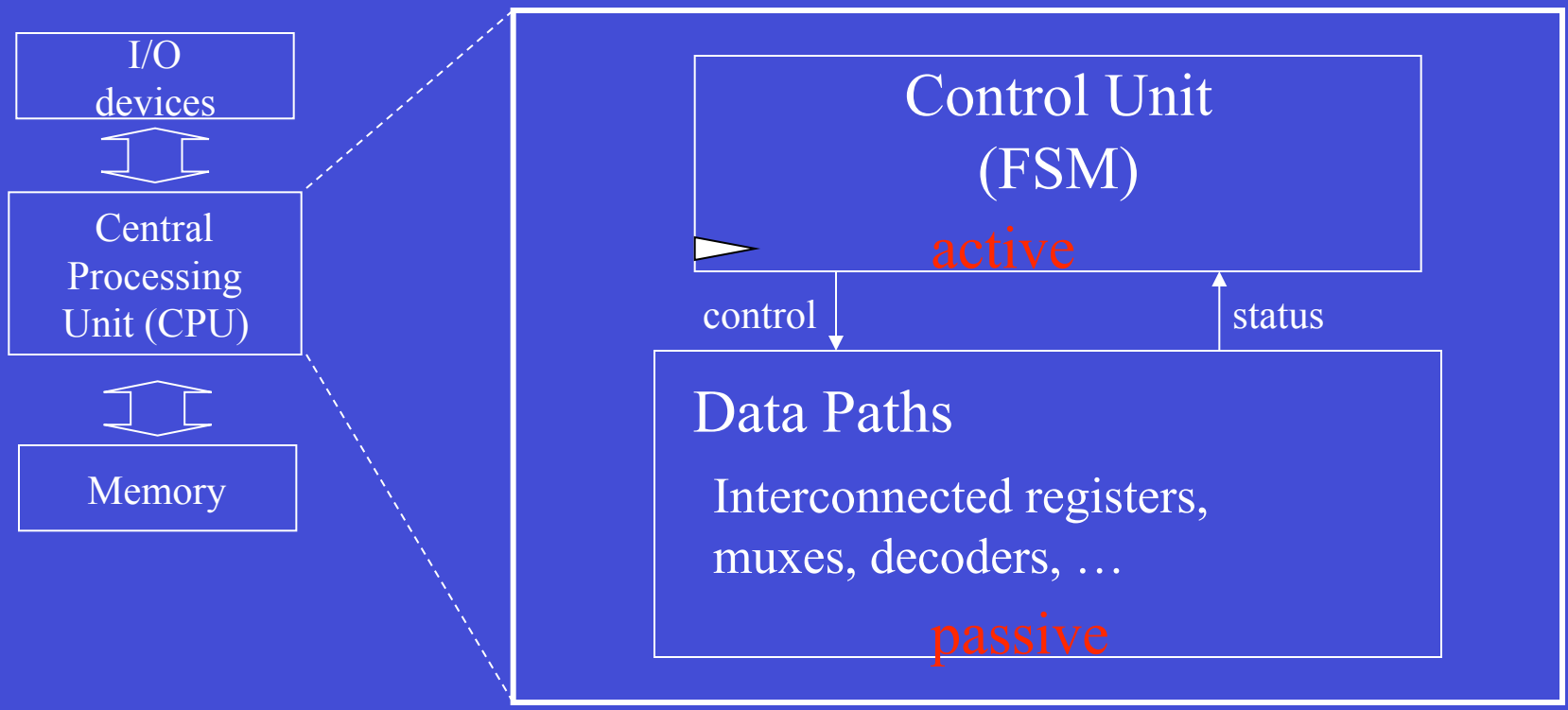
Allow communication with outside world

Interprets instructions

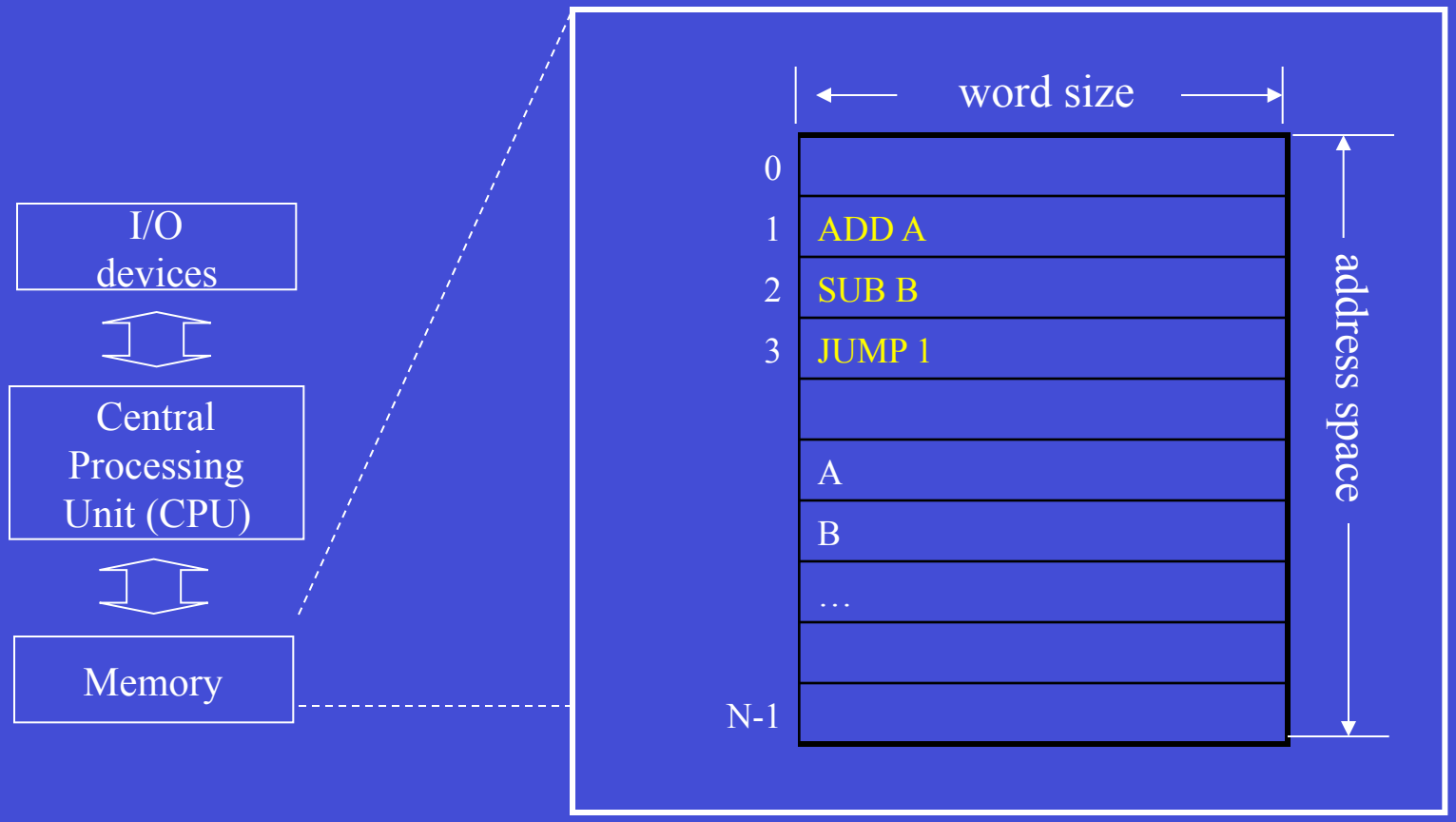
Stores both programs and data

After 60 years … most processors still look like this!

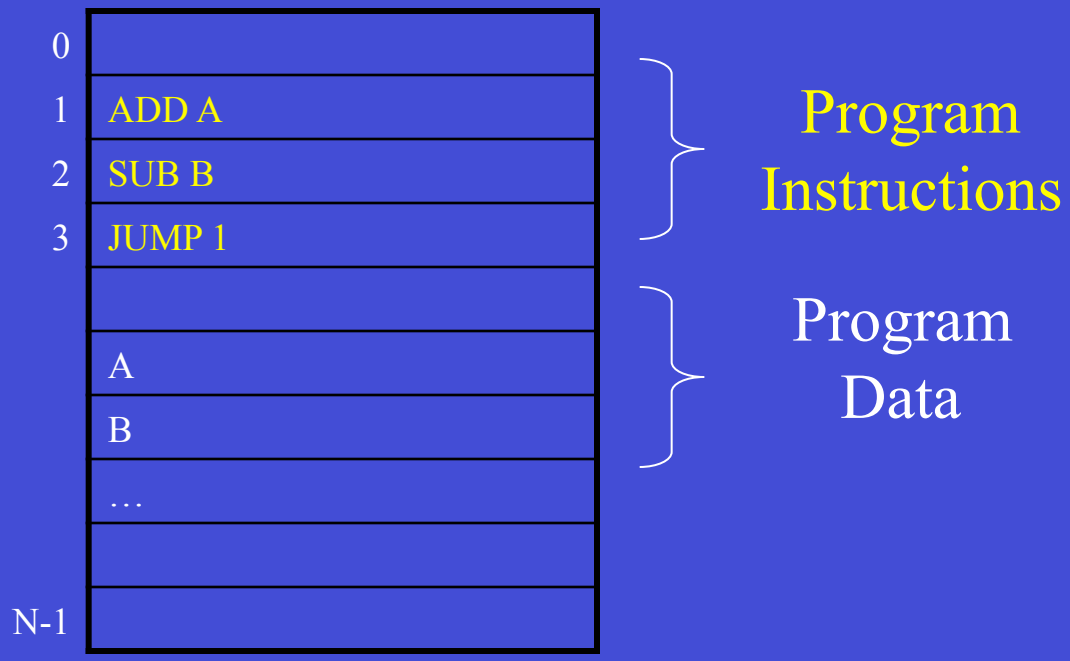
The von Neumann Architecture
Central Processing (CPU)



The (John) Von Neumann Architecture
The Memory Unit

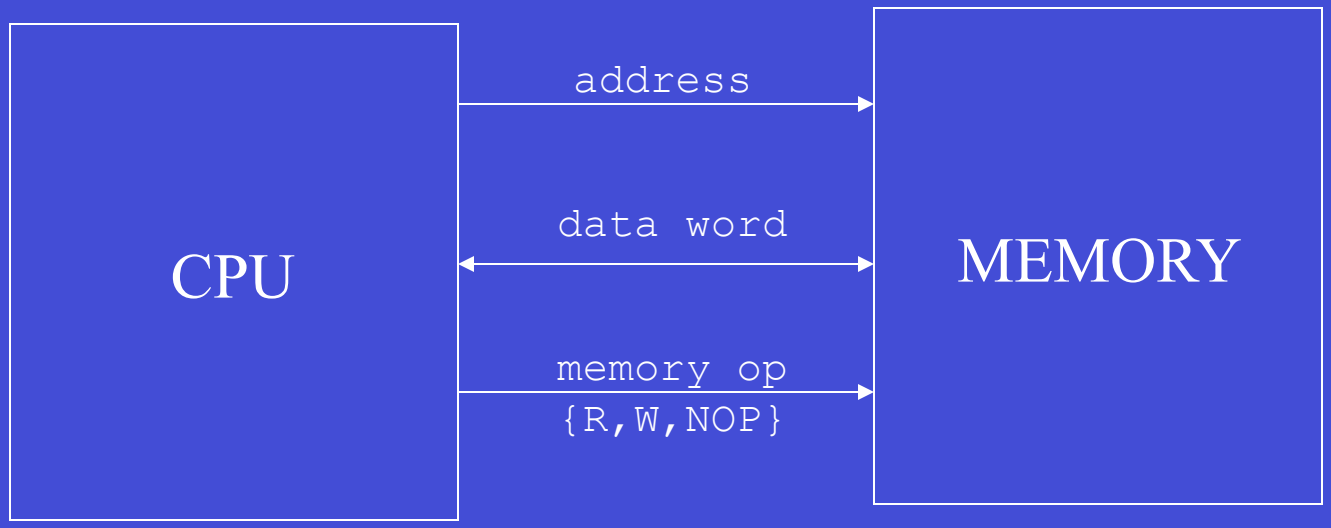


The (John) Von Neumann Architecture Stored Program Concept



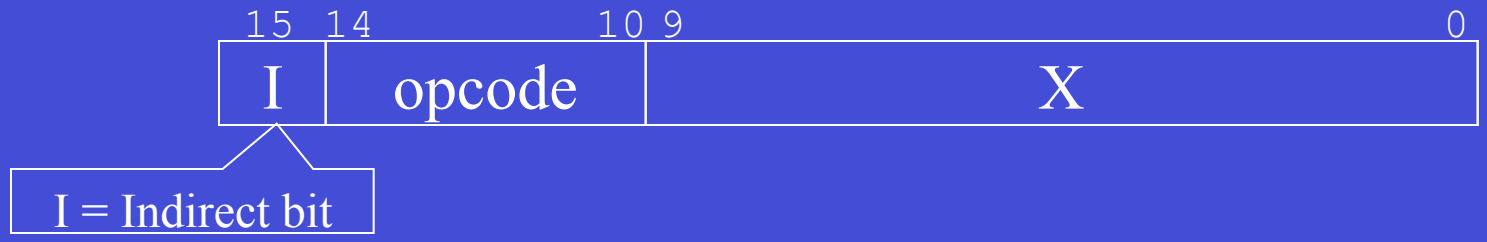
- Programs and their data coexist in memory
- Processor, under program control, keeps track of what needs to be interpreted as instructions and what as data.

Easy I
Memory Interface



Easy I
A Simple Accumulator Processor
Instruction Set Architecture (ISA)

Instruction Format (16 bits)



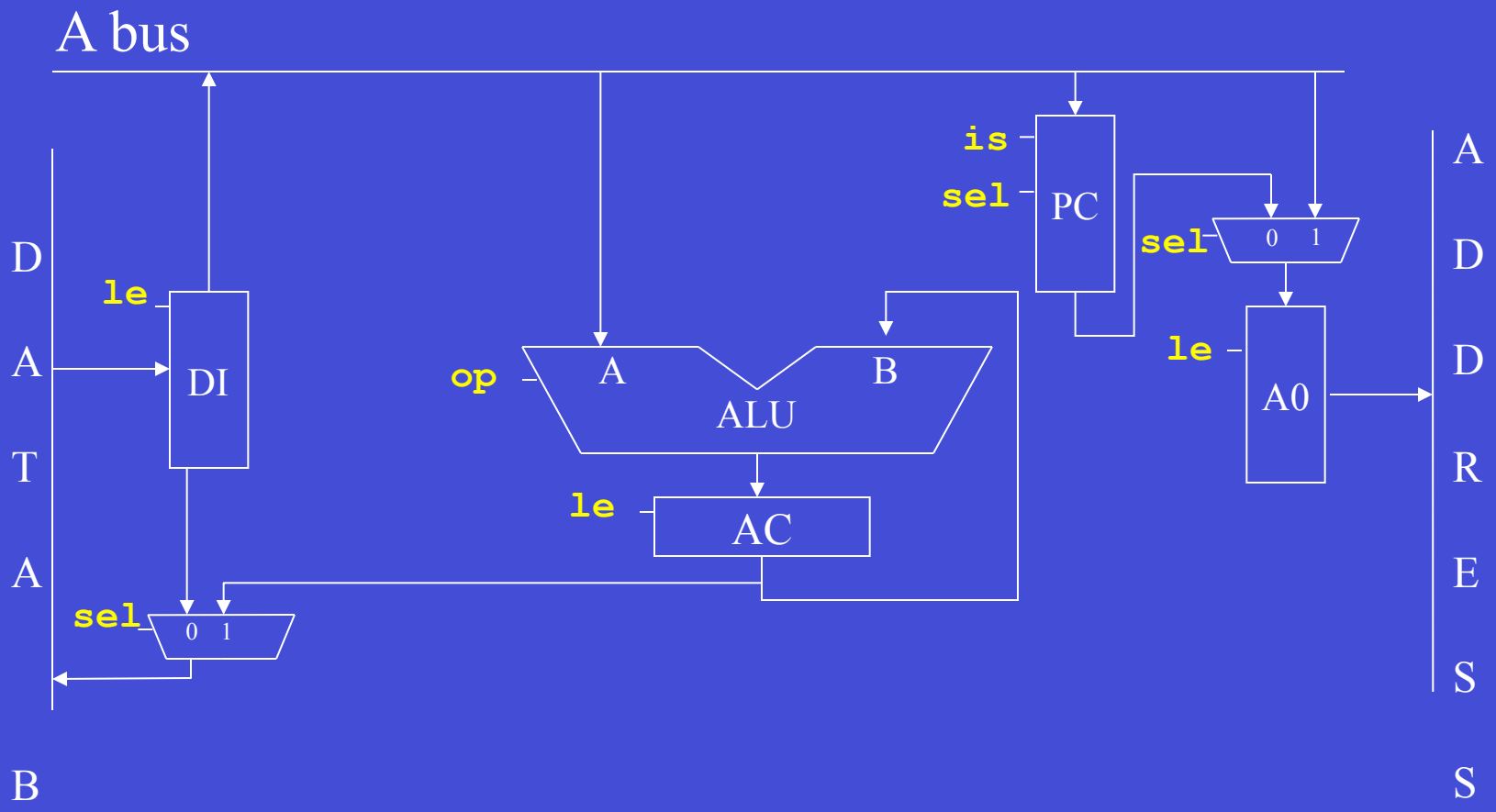
Easy I
A Simple Accumulator Processor
Instruction Set Architecture (ISA)

Instruction Set

Name	Opcode	Action I=0	Action I=1
Comp	00 000	AC ← not AC	AC <- not AC
ShR	00 001	AC ← AC / 2	AC ← AC / 2
BrN	00 010	AC < 0 ⇒ PC ← X	AC < 0 ⇒ PC ← MEM[X]
Jump	00 011	PC ← X	PC ← MEM[X]
Store	00 100	MEM[X] ← AC	MEM[MEM[X]] ← AC
Load	00 101	AC ← MEM[X]	AC ← MEM[MEM[X]]
And	00 110	AC ← AC and X	AC ← AC and MEM[X]
Add	00 111	AC ← AC + X	AC ← AC + MEM[X]

Easy all right … but universal it is!

Easy I
Data Paths (with control points)

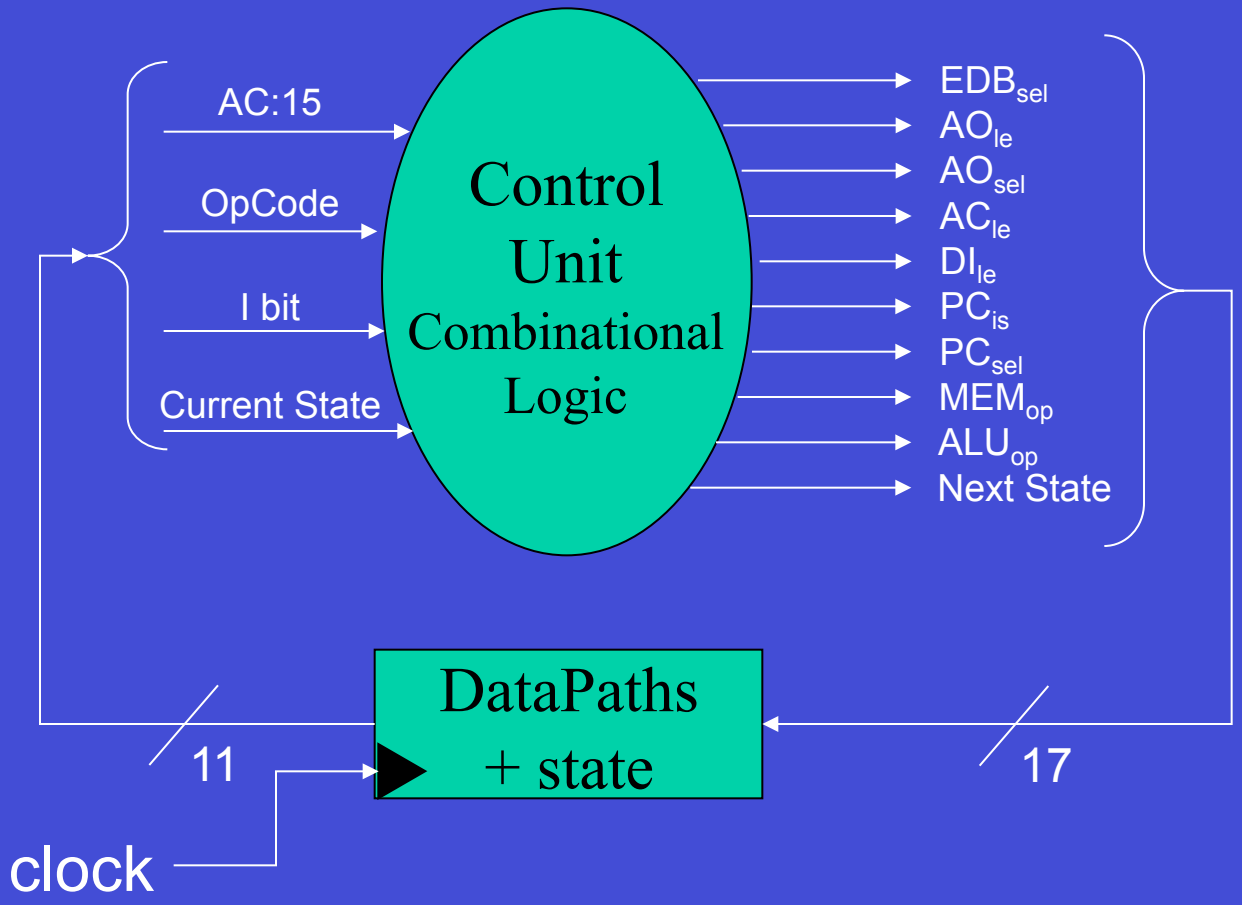


Easy I
A Simple Accumulator Processor
Instruction Set Architecture (ISA)

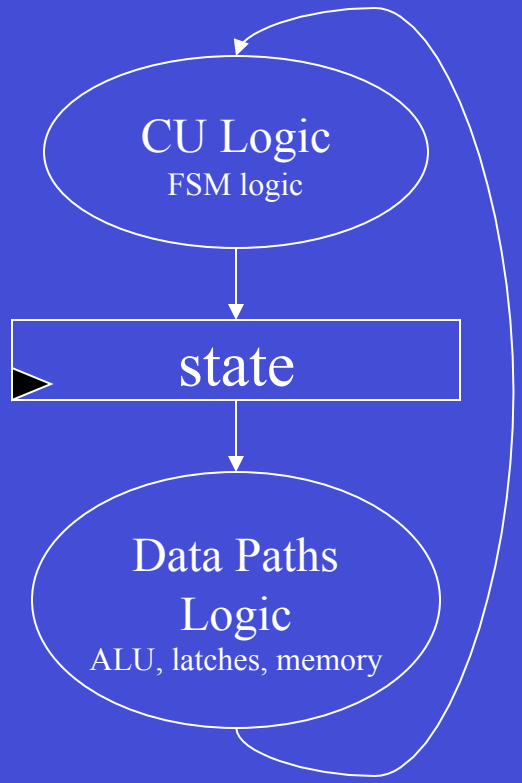
Some Immediate Observations on the Easy I ISA

- Accumulator (AC) is implicit operand to many instructions. No need to use instruction bits to specify one of the operands. More bits left for address and opcodes.
- Although simple, Easy I is universal. (given enough memory). Can you see this?
- Immediate bit specifies level of indirection for the location of the operand. I = 1: operand in X field (immediate). I=1 operand in memory location X (indirect).

Easy I - Control Unit

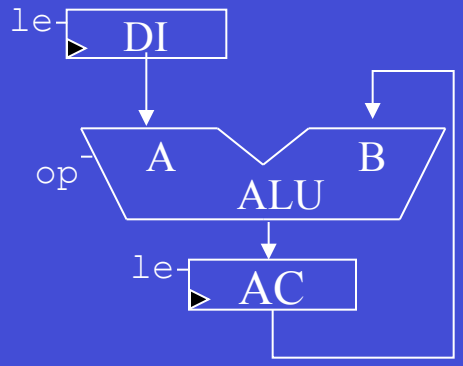


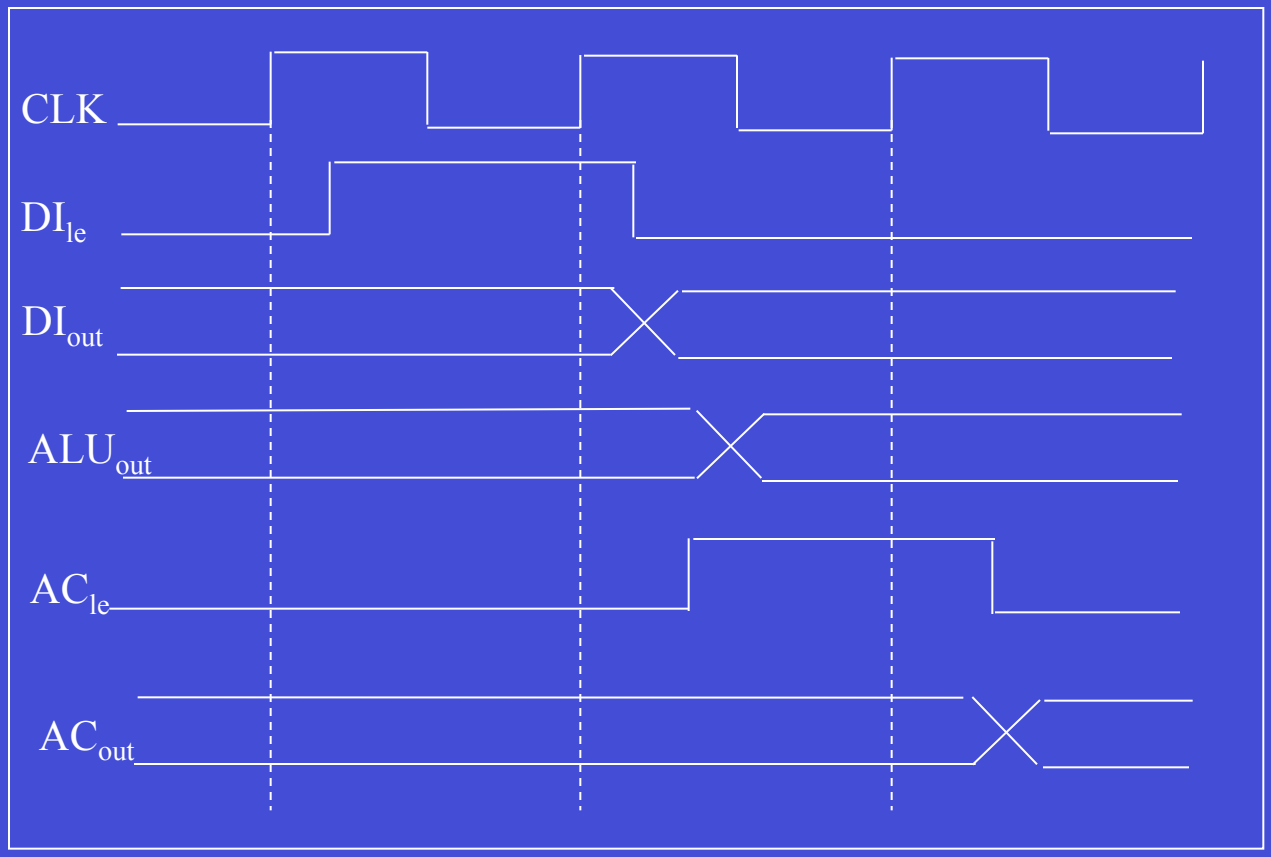
What makes a CPU cycle?



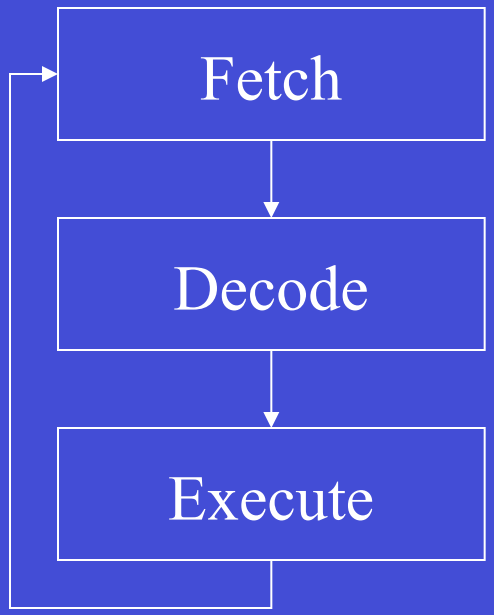
Cycle time must accommodate signal propagation

Easy I – Timing Example
ALU Operation





Easy I
Control Unit
(Level 0 Flowcharts)



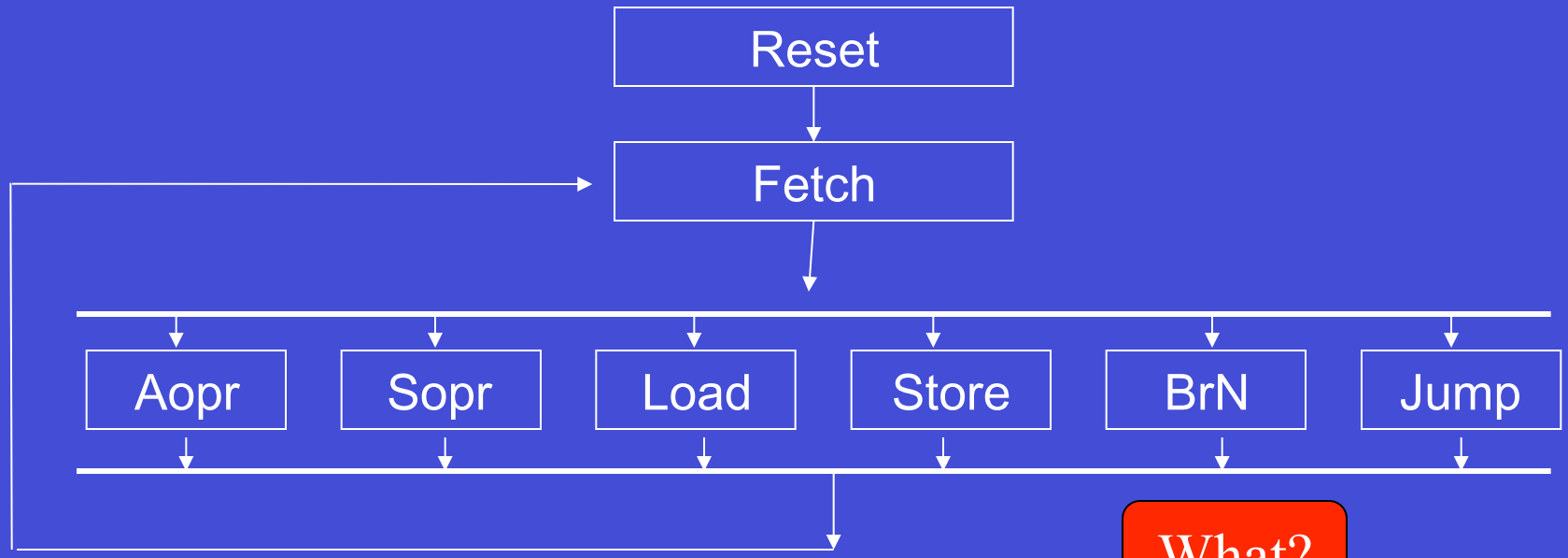
Read next instruction

Determine what it does and prepare to do it. Fetch operands.

Do it!

We will ignore indirect bit (assuming I = 0) for now

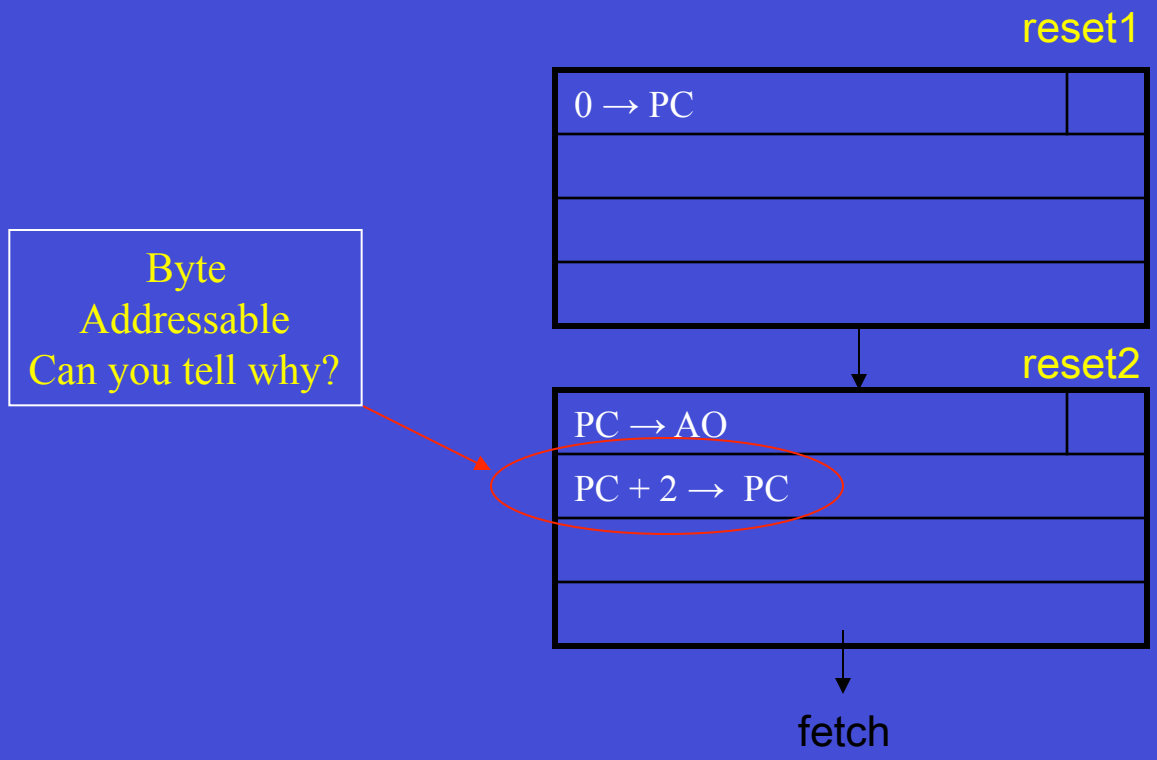
Easy I
Control Unit
(Level 1 Flowcharts)



Level 1: Each box may take several CPU cycles to execute

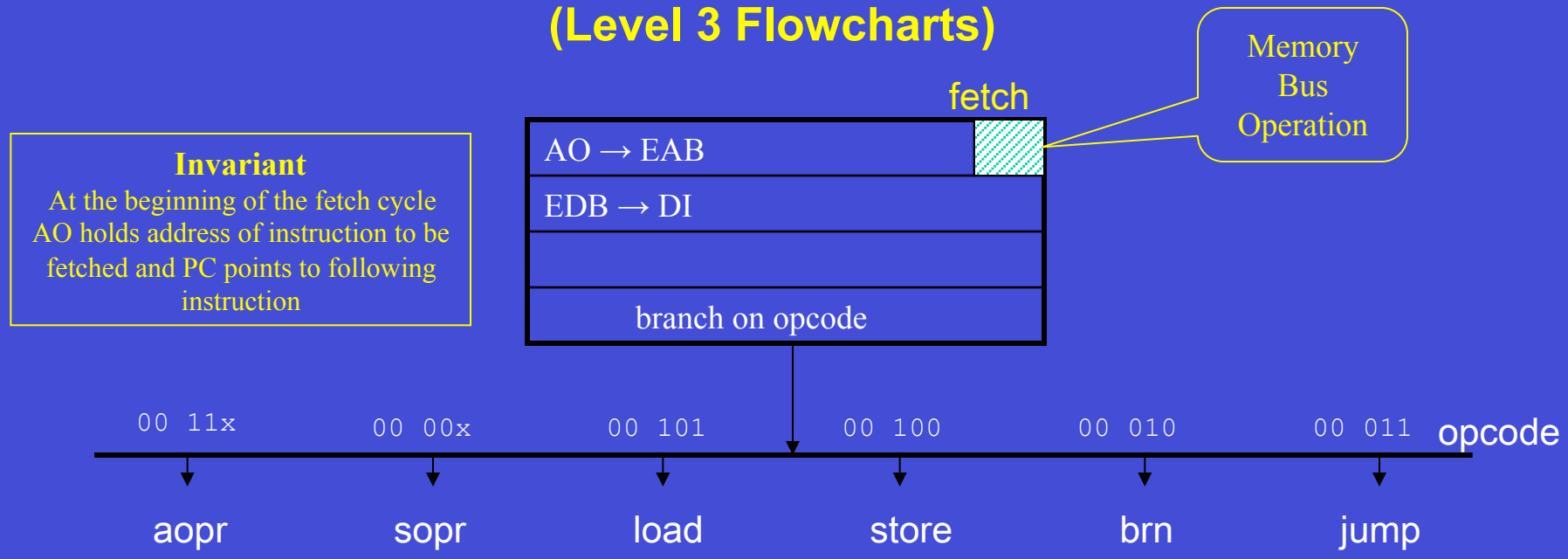
RESET

Easy I
Control Unit
(Level 2 Flowcharts)



Each box may take only one CPU cycle to execute

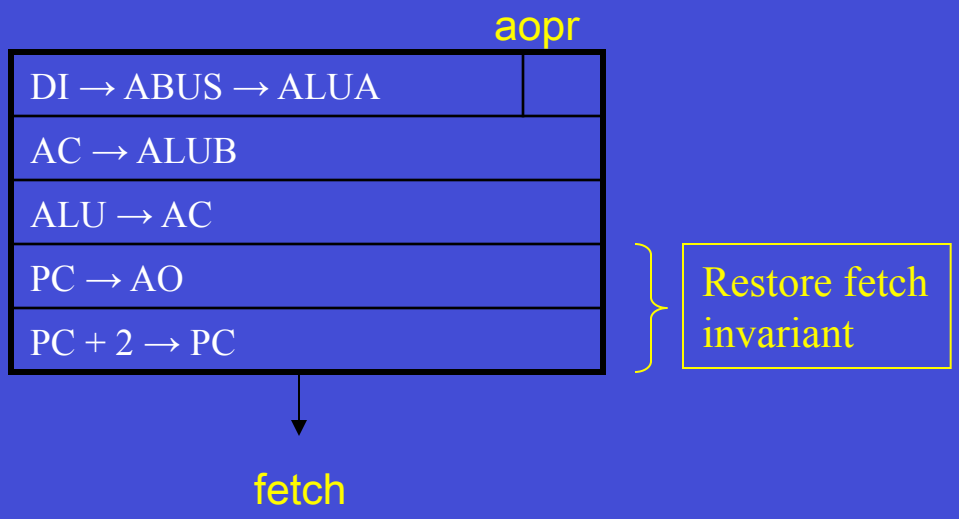
Easy I
Control Unit
(Level 3 Flowcharts)



Opcode must be an input to CU's sequential circuit

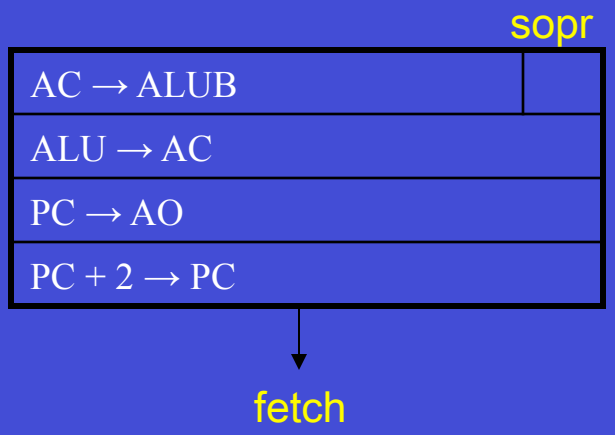
Easy I
Control Unit
(Level 2 Flowcharts)

AOpr



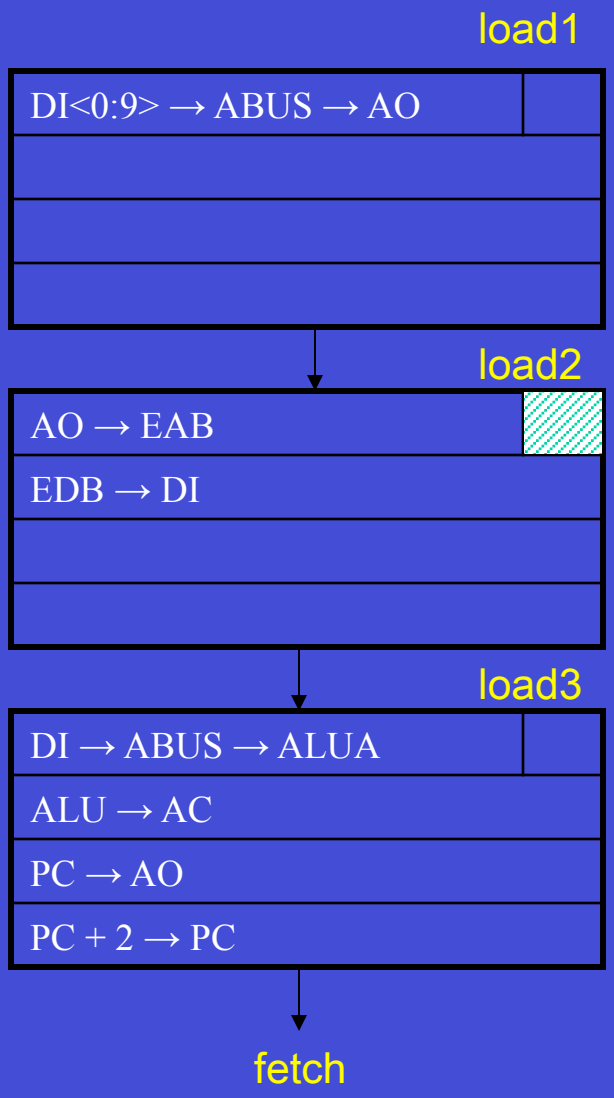
SOpr

Easy I
Control Unit
(Level 2 Flowcharts)

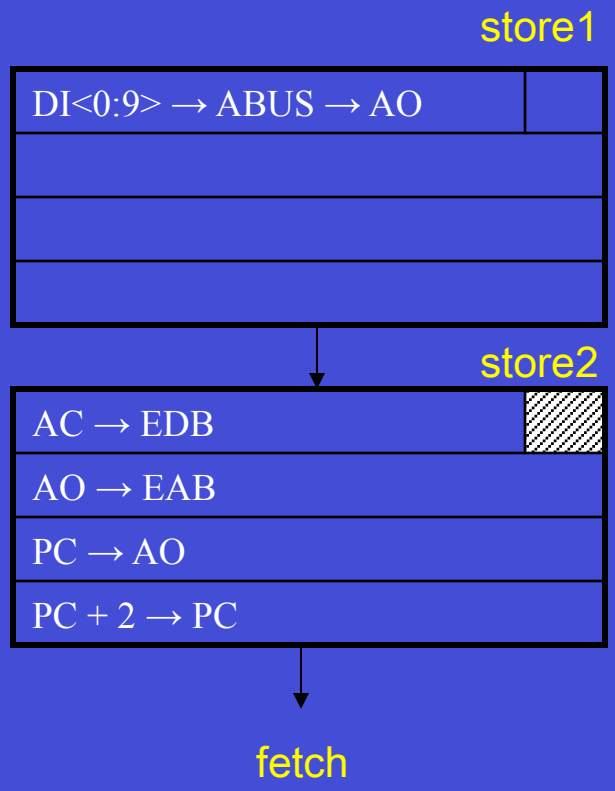


Easy I
Control Unit
(Level 2 Flowcharts)

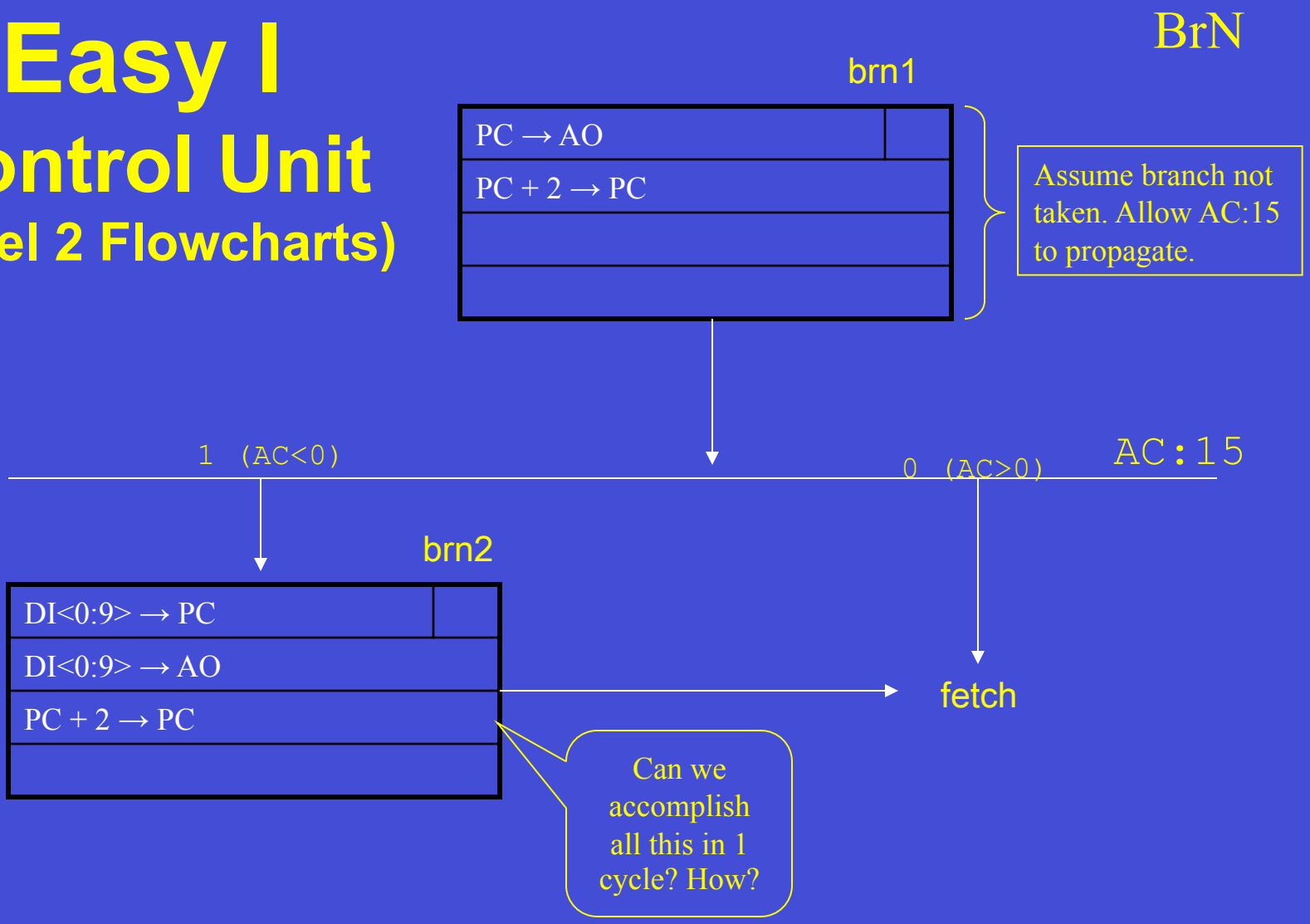
Load



Easy I
Control Unit
(Level 2 Flowcharts)

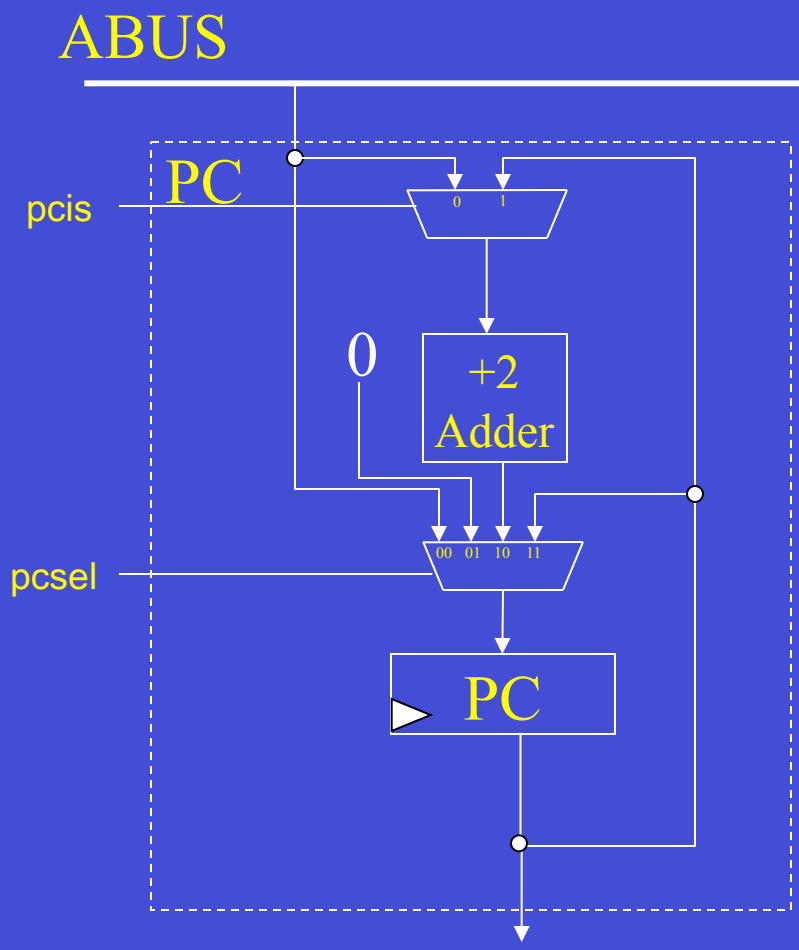


Easy I
Control Unit
(Level 2 Flowcharts)



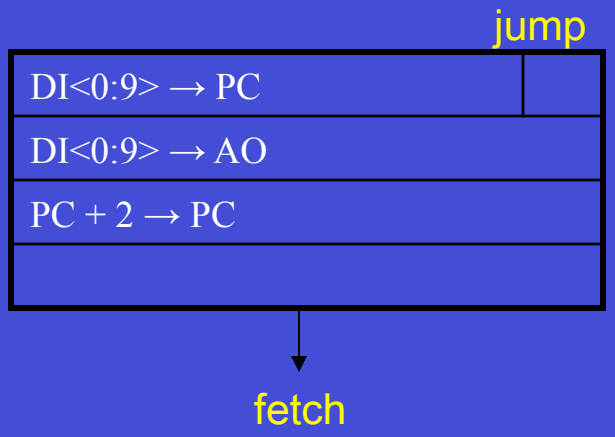
Bit 15 of AC input to the CU's sequential circuit

Inside the Easy-I PC

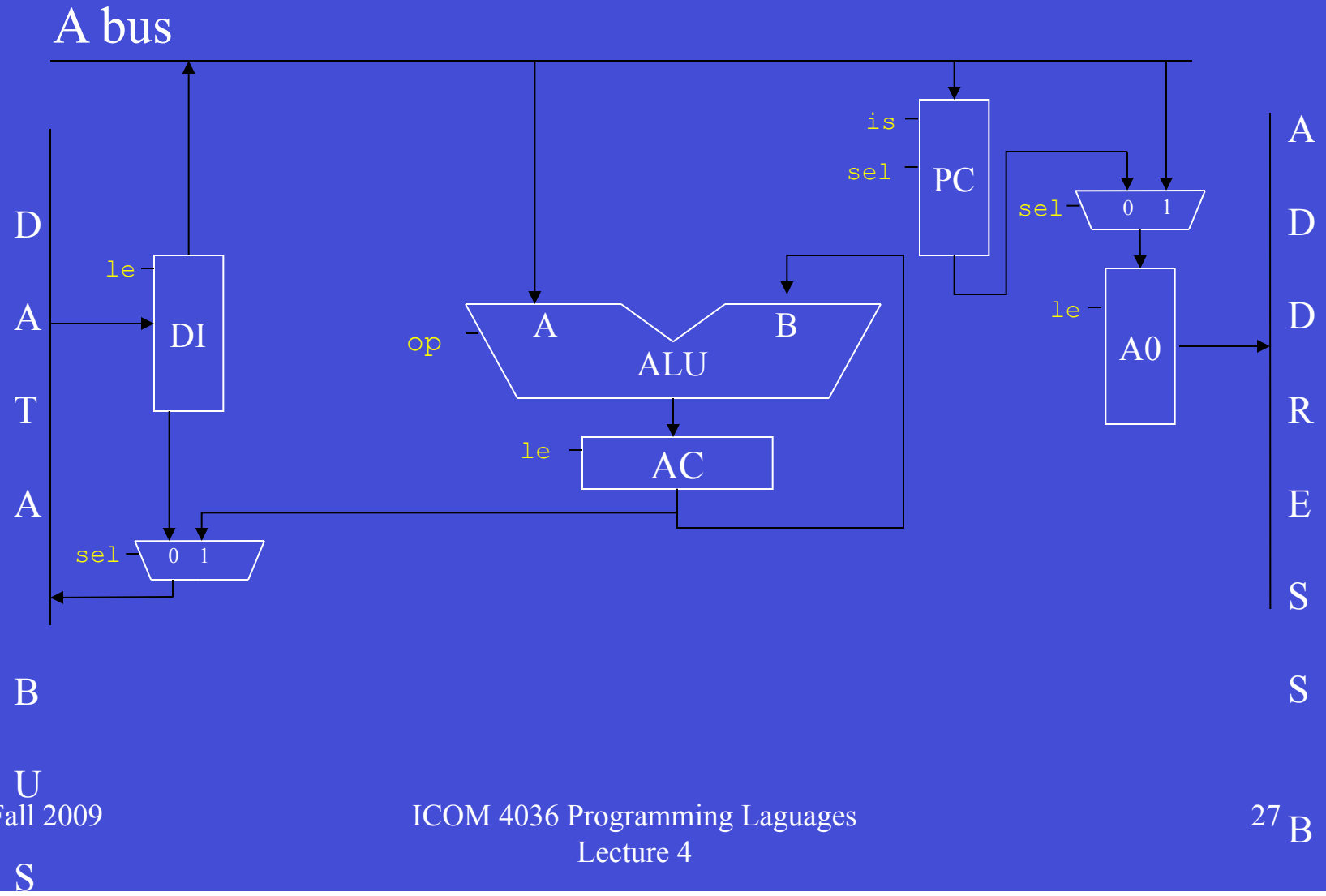


PC capable of loading and incrementing simultaneously

Easy I
Control Unit
(Level 2 Flowcharts)



Easy I
Data Paths (with control points)



Easy I
Control Unit State Transition Table (Part I)

Curr State	opcode	AC:15	Next State	ALU op	Mem OP	PC sel	PC is	DI le	AC le	AO sel	AO le	EDB sel
reset1	xx xxx	x	reset2	XXX	NOP	01	X	0	0	X	0	X
reset2	xx xxx	x	fetch	XXX	NOP	10	1	0	0	0	1	X
fetch	00 00x	x	sopr	XXX	NOP	11	X	1	0	X	0	X
fetch	00 010	x	brn1	XXX	RD	11	X	1	0	X	0	X
fetch	00 011	x	jump	XXX	RD	11	X	1	0	X	0	X
fetch	00 100	x	store1	XXX	RD	11	X	1	0	X	0	X
fetch	00 101	x	load1	XXX	RD	11	X	1	0	X	0	X
fetch	00 11x	x	aopr	XXX	RD	11	X	1	0	X	0	X
aopr	00 110	x	fetch	AND	NOP	10	1	0	1	0	1	X
aopr	00 111	x	fetch	ADD	NOP	10	1	0	1	0	1	X
sopr	00 000	x	fetch	NOTB	NOP	10	1	0	1	0	1	X
sopr	00 001	x	fetch	SHRB	NOP	10	1	0	1	0	1	X

Easy I
Control Unit State Transition Table (Part II)

Current State	opcode	AC:15	Next State	ALU op	Mem OP	PC sel	PC is	DI le	AC le	AO sel	AO le	EDB sel
store1	xx xxx	x	store2	XXX	NOP	11	X	0	0	1	1	X
store2	xx xxx	x	store3	XXX	WR	10	1	0	0	0	1	1
load1	xx xxx	x	load2	XXX	NOP	11	X	0	0	1	1	X
load2	xx xxx	x	load3	XXX	RD	11	X	1	0	X	0	X
load3	xx xxx	x	fetch	XXX	NOP	10	1	0	1	0	1	X
brn1	xx xxx	0	fetch	XXX	NOP	10	1	0	0	0	1	X
brn1	xx xxx	1	brn2	XXX	NOP	10	1	0	0	0	1	X
brn2	xx xxx	x	fetch	XXX	NOP	10	0	0	0	1	1	X
jump	xx xxx	x	fetch	XXX	NOP	10	0	0	0	1	1	X

CU with 14 states => 4 bits of state

This is a (micro)program that interprets machine code

Easy-I Control Unit – Some missing details

4-bit Encodings for States

State	Encoding
reset1	0000
reset2	0001
fetch	0010
aopr	0011
sopr	0100
store1	0101
store2	0110
store3	0111
load1	1000
load2	1001
load3	1010
brn1	1011
brn2	1100
jump	1101

ALU Operation Table

Operation	Code	Output
A	000	A
NOTB	001	not B
AND	010	A and B
ADD	011	A + B
SHRB	100	B / 2

We know how to implement this ALU !

Control Bus Operation Table

Operation	Code
NOP	00
ReaD	01
WRite	10

Easy I
Control Unit State Transition Table (Part I)

Curr State	opcode	AC:15	Next State	ALU op	Mem OP	PC sel	PC is	DI le	AC le	AO sel	AO le	EDB sel
0000	xx xxx	x	0001	XXX	00	01	X	0	0	X	0	X
0001	xx xxx	x	0010	XXX	00	10	1	0	0	0	1	X
0010	00 00x	x	0100	XXX	00	11	X	1	0	X	0	X
0010	00 010	x	1011	XXX	01	11	X	1	0	X	0	X
0010	00 011	x	1101	XXX	01	11	X	1	0	X	0	X
0010	00 100	x	0101	XXX	01	11	X	1	0	X	0	X
0010	00 101	x	1000	XXX	01	11	X	1	0	X	0	X
0010	00 11x	x	0011	XXX	01	11	X	1	0	X	0	X
0011	00 110	x	0010	010	00	10	1	0	1	0	1	X
0011	00 111	x	0010	011	00	10	1	0	1	0	1	X
0100	00 000	x	0010	001	00	10	1	0	1	0	1	X
0100	00 001	x	0010	100	00	10	1	0	1	0	1	X

Easy I

Control Unit State Transition Table (Part II)

Current State	opcode	AC:15		Next State	ALU op	Mem OP	PC sel	PC is	DI le	AC le	AO sel	AO le	EDB sel
0101	xx xxx	x		0110	XXX	00	11	X	0	0	1	1	X
0110	xx xxx	x		0111	XXX	10	10	1	0	0	0	1	1
1000	xx xxx	x		1001	XXX	00	11	X	0	0	1	1	X
1001	xx xxx	x		1010	XXX	01	11	X	1	0	X	0	X
1010	xx xxx	x		0010	XXX	00	10	1	0	1	0	1	X
1011	xx xxx	0		0010	XXX	00	10	1	0	0	0	1	X
1011	xx xxx	1		1100	XXX	00	10	1	0	0	0	1	X
1100	xx xxx	x		0010	XXX	00	10	0	0	0	1	1	X
1101	xx xxx	x		0010	XXX	00	10	0	0	0	1	1	X

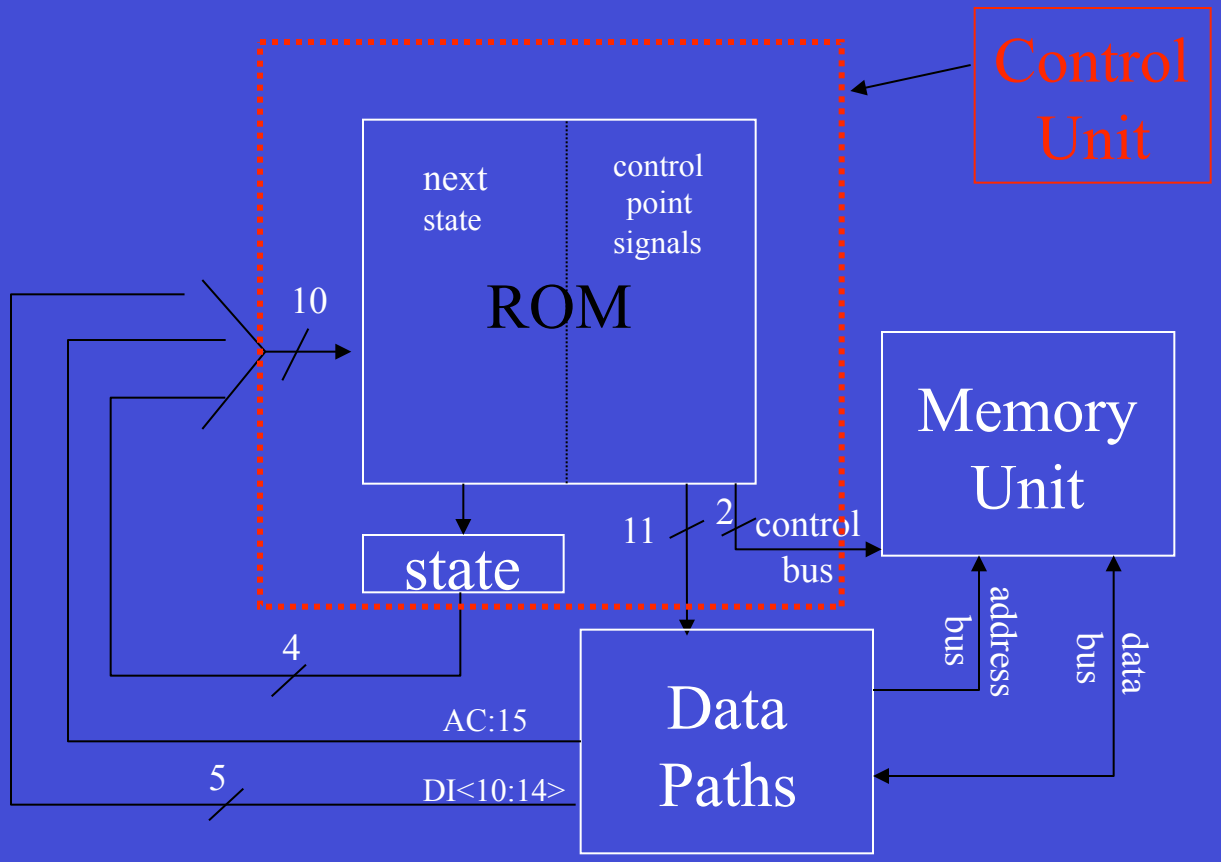
Building the Easy-I C-Unit 2 Approaches

- Harwired
  - Apply well known sequential circuit techniques
- Micro-programmed
  - Treat state transition table as a program
  - Build a new abstraction layer

A μprogram

The Microprogramming abstraction level

Building the Easy-I C-Unit
Hardwired Approach



Computing Integer Division
Iterative C++ Version

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

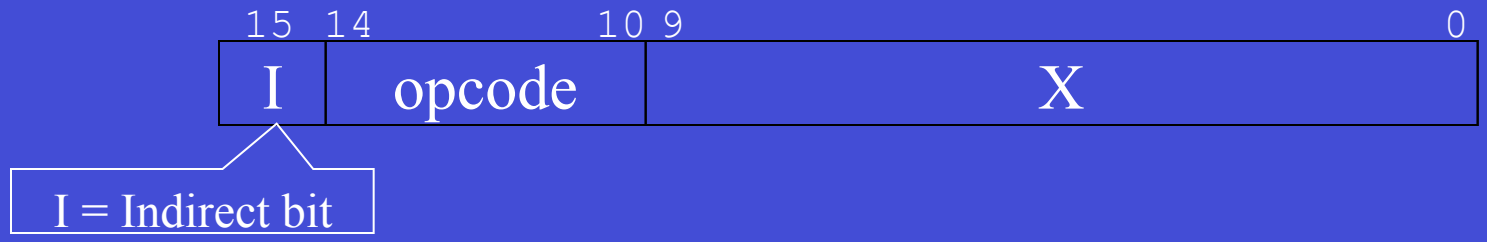
We ignore procedures and I/O for now

Definition
Instruction Set Architecture

- What it is:
  - The programmers view of the processor
  - Visible registers, instruction set, execution model, memory model, I/O model
- What it is not:
  - How the processors if build
  - The processor's internal structure

Easy I
A Simple Accumulator Processor
Instruction Set Architecture (ISA)

Instruction Format (16 bits)

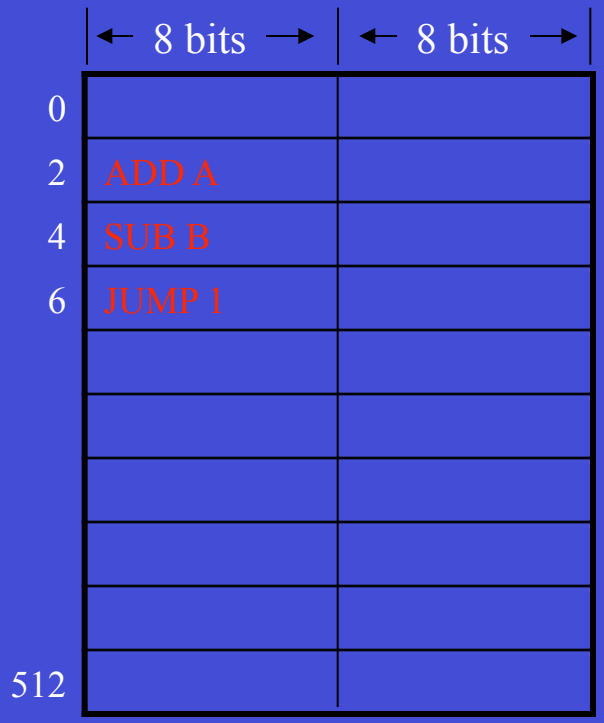


Easy I
A Simple Accumulator Processor
Instruction Set Architecture (ISA)

Instruction Set

Symbolic Name	Opcode	Action I=0	Symbolic Name	Action I=1
Comp	00 000	AC ← not AC	Comp	AC <- not AC
ShR	00 001	AC ← AC / 2	ShR	AC ← AC / 2
BrN*i*	00 010	AC < 0 ⇒ PC ← X	BrN	AC < 0 ⇒ PC ← MEM[X]
Jump*i*	00 011	PC ← X	Jump	PC ← MEM[X]
Store*i*	00 100	MEM[X] ← AC	Store	MEM[MEM[X]] ← AC
Load*i*	00 101	AC ← MEM[X]	Load	AC ← MEM[MEM[X]]
And*i*	00 110	AC ← AC and X	And	AC ← AC and MEM[X]
Add*i*	00 111	AC ← AC + X	Add	AC ← AC + MEM[X]

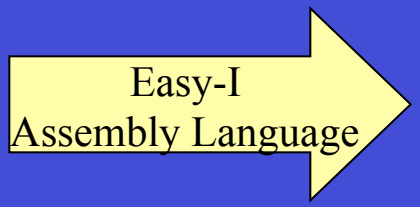
Easy I
Memory Model



Computing Integer Division

Iterative C++ Version

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```



Computing Integer Division
Iterative C++ Version

Translate Data: Global Layout

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++
HLL

Easy-I
Assembly Language

```
0:      andi   0          # AC = 0
        addi   12
        storei 1000       # a = 12 (a   tored @ 1000)
        andi   0          # AC = 0
        addi   4
        storei 1004       # b = 4     tored @ 1004)
        andi   0          # AC =
        storei 1008       # res       0 (result @ 1008)
```

Issues
- Memory allocation
- Data Alignment
- Data Sizing

Computing Integer Division
Iterative C++ Version

Translate Code: Conditionals If-Then

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++ HLL

Easy-I Assembly Language

```
0:      andi   0          # AC = 0
        addi   12
        storei 1000       # a = 12 (a stored @ 1000)
        andi   0          # AC = 0
        addi   4
        storei 1004       # b = 4 (b stored @ 1004)
        andi   0          # AC = 0
        storei 1008       # result = 0 (result @ 1008)
main:   loadi  1004       # compute a - b in AC
        comp              # using 2's complement add
        addi   1
        add    1000
        brni   exit       # exit if AC negative




exit:
```

Issues
- Must translate HLL boolean expression into ISA-level branching condition

Computing Integer Division
Iterative C++ Version

Translate Code: Iteration (loops)

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++ HLL

Easy-I Assembly Language

```
0:        andi   0          # AC = 0
          addi   12
          storei 1000       # a = 12 (a stored @ 1000)
          andi   0          # AC = 0
          addi   4
          storei 1004       # b = 4 (b stored @ 1004)
          andi   0          # AC = 0
          storei 1008       # result = 0 (result @ 1008)
main:     loadi  1004       # compute a - b in AC
          comp              # using 2's complement add
          addi   1
          add    1000
          brni   exit        # exit if AC negative
loop:     loadi  1000
          brni   endloop
          jump   loop
endloop:
exit:
```

Fall 2009

Computing Integer Division
Iterative C++ Version

Translate Code: Arithmetic Ops

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++ HLL

Easy-I Assembly Language

```
0:      andi   0          # AC = 0
        addi   12
        storei 1000       # a = 12 (a stored @ 1000)
        andi   0          # AC = 0
        addi   4
        storei 1004       # b = 4 (b stored @ 1004)
        andi   0          # AC = 0
        storei 1008       # result = 0 (result @ 1008)
main:   loadi  1004       # compute a - b in AC
        comp              # using 2's complement add
        addi   1
        add    1000
        brni   exit       # exit if AC negative
loop:   loadi  1000
        brni   endloop
        loadi  1004       # compute a - b in AC
        comp              # using 2's complement add
        addi   1
        add    1000       # Uses indirect bit I = 1



        jumpi  loop
endloop:
exit:
```

Computing Integer Division
Iterative C++ Version

Translate Code: Assignments

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++ HLL

Easy-I Assembly Language

```
0:      andi   0          # AC = 0
        addi   12
        storei 1000       # a = 12 (a stored @ 1000)
        andi   0          # AC = 0
        addi   4
        storei 1004       # b = 4  (b stored @ 1004)
        andi   0          # AC = 0
        storei 1008       # result = 0 (result @ 1008)
main:   loadi  1004       # compute a - b in AC
        comp              # using 2's complement add
        addi   1
        add    1000
        brni   exit       # exit if AC negative
loop:   loadi  1000
        brni   endloop
        loadi  1004       # compute a - b in AC
        comp              # using 2's complement add
        addi   1
        add    1000       # Uses indirect bit I = 1
        storei 1000




        jump loop
endloop:
exit:
```

Computing Integer Division
Iterative C++ Version

Translate Code: Increments

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++ HLL

Easy-I Assembly Language

```
0:      andi   0          # AC = 0
        addi   12
        storei 1000       # a = 12 (a stored @ 1000)
        andi   0          # AC = 0
        addi   4
        storei 1004       # b = 4  (b stored @ 1004)
        andi   0          # AC = 0
        storei 1008       # result = 0 (result @ 1008)
main:   loadi  1004       # compute a - b in AC
        comp              # using 2's complement add
        addi   1
        add    1000
        brni   exit       # exit if AC negative
loop:   loadi  1000
        brni   endloop
        loadi  1004       # compute a - b in AC
        comp              # using 2's complement add
        addi   1
        add    1000       # Uses indirect bit I = 1
        storei 1000
        loadi  1008       # result = result + 1
        addi   1
        storei 1008
        jumpi  loop
endloop:
exit:
```

Computing Integer Division

Easy I Machine Code

Data

Address	Contents
1000	a
1004	b
1008	result

Challenge

Make this program as small and fast as possible

Address	I Bit	Opcode (binary)	X (base 10)
0	0	00 110	0
2	0	00 111	12
4	0	00 100	1000
6	0	00 110	0
8	0	00 111	4
10	0	00 100	1004
12	0	00 110	0
14	0	00 100	1008
16	0	00 101	1004
18	0	00 000	unused
20	0	00 111	1
22	1	00 111	1000
24	0	00 010	46
26	0	00 101	1000
28	0	00 010	46
30	0	00 101	1004
32	0	00 000	unused
34	0	00 111	1
36	0	00 100	1000
38	0	00 101	1008
40	0	00 111	1
42	0	00 100	1008
44	0	00 011	26

Program

The MIPS Architecture
ISA at a Glance

- Reduced Instruction Set Computer (RISC)
- 32 general purpose 32-bit registers
- Load-store architecture: Operands in registers
- Byte Addressable
- 32-bit address space

The MIPS Architecture

32 Register Set (32-bit registers)

Register #	Reg Name	Function
r0	r0	Zero constant
r4-r7	a0-a3	Function arguments
r1	at	Reserved for Operating Systems
r30	fp	Frame pointer
r28	gp	Global memory pointer
r26-r27	k0-k1	Reserved for OS Kernel
r31	ra	Function return address
r16-r23	s0-s7	Callee saved registers
r29	sp	Stack pointer
r8-r15	t0-t7	Temporary variables
r24-r25	t8-t9	Temporary variables
r2-r3	v0-v1	Function return values

The MIPS Architecture
Main Instruction Formats

Simple and uniform 32-bit 3-operand instruction formats

–R Format: Arithmetic/Logic operations on registers

–I Format: Branches, loads and stores

–J Format: Jump Instruction

MIPS Data Paths

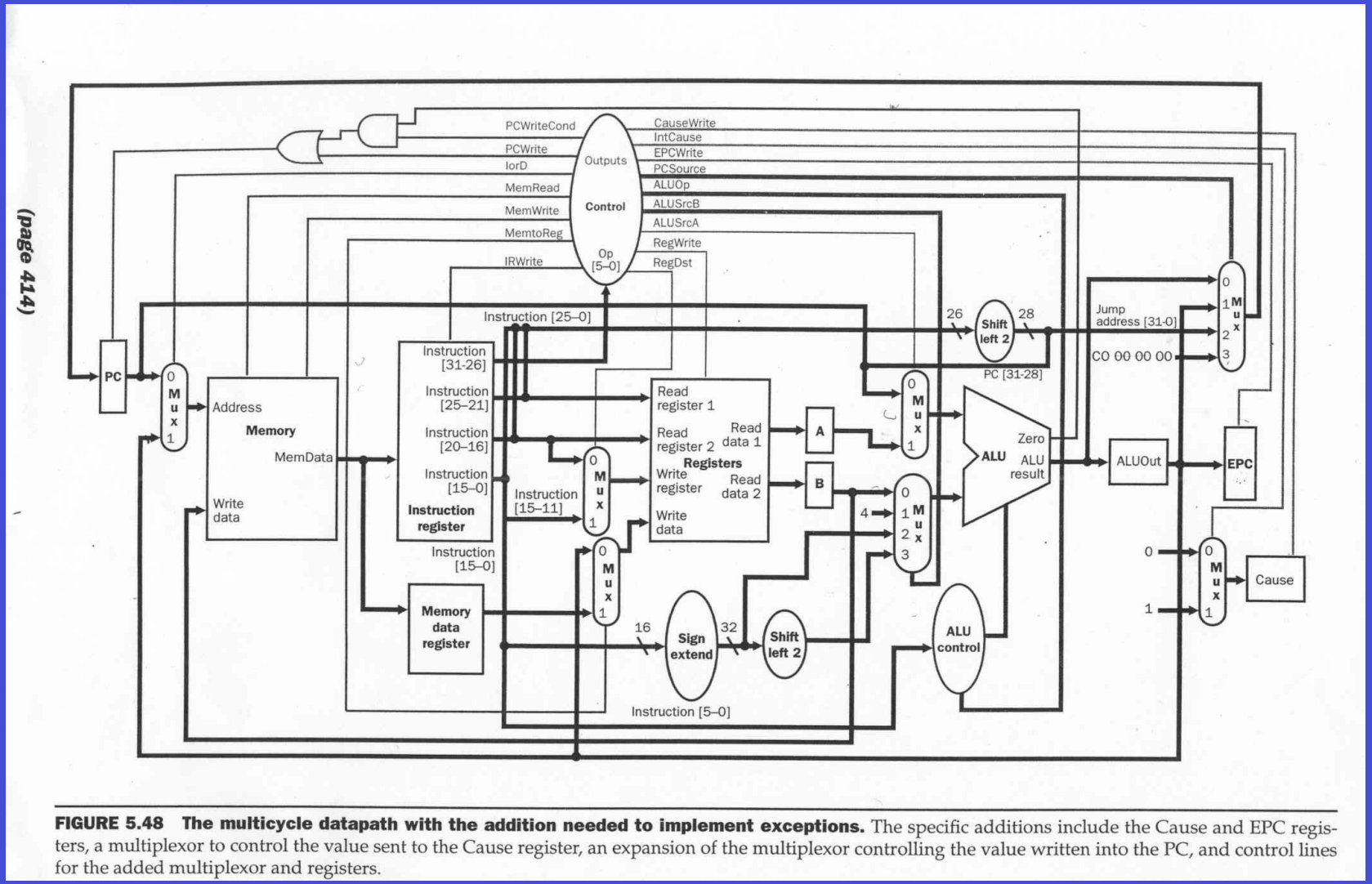
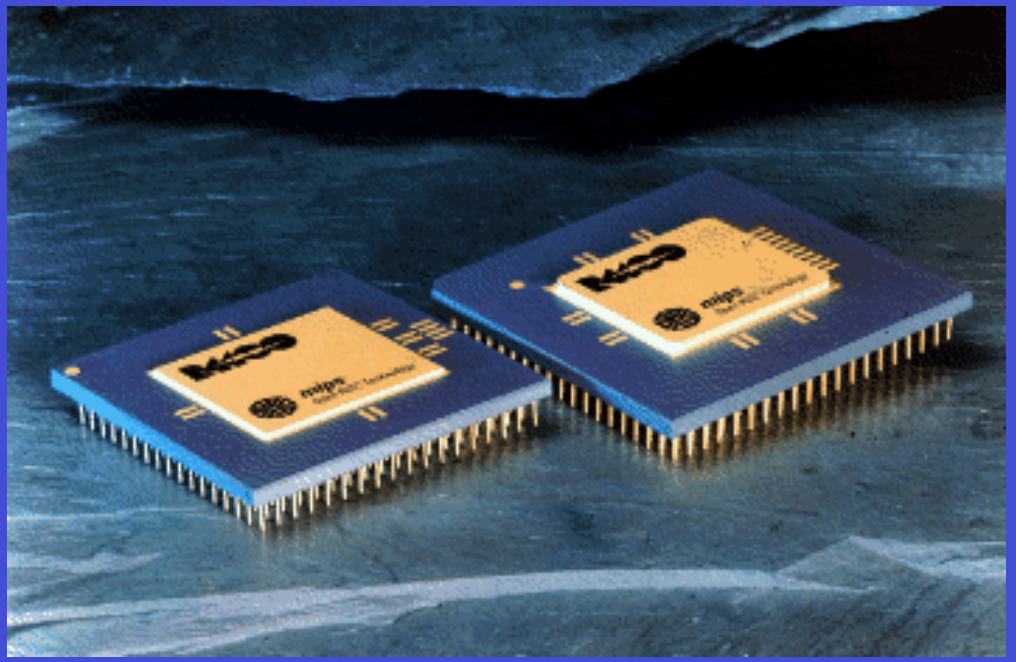


FIGURE 5.48 The multicycle datapath with the addition needed to implement exceptions. The specific additions include the Cause and EPC registers, a multiplexor to control the value sent to the Cause register, an expansion of the multiplexor controlling the value written into the PC, and control lines for the added multiplexor and registers.

Mips Packaging



The MIPS Architecture

Examples of Native Instruction Set

Instruction Group	Instruction	Function
Arithmetic/ Logic	add $s1,$s2,$s3	$s1 = $s2 + $s3
	addi $s1,$s2,K	$s1 = $s2 + K
Load/Store	lw $s1,K($s2)	$s1 = MEM[$s2+K]
	sw $s1,K($s2)	MEM[$s2+K] = $s1
Jumps and Conditional Branches	beq $s1,$s2,K	if ($s1=$s2) goto PC + 4 + K
	slt $s1,$s2,$s3	if ($s2<$s3) $s1=1 else $s1=0
	j K	goto K
Procedures	jal K	$ra = PC + 4; goto K
	jr $ra	goto $ra

The SPIM Assembler

Examples of Pseudo-Instruction Set

Instruction Group	Syntax	Translates to:
Arithmetic/ Logic	neg $s1, $s2	sub $s1, $r0, $s2
	not $s1, $s2	nor $17, $18, $0
Load/Store	li $s1, K	ori $s1, $0, K
	la $s1, K	lui $at, 152 ori $s1, $at, -27008
	move $s1, $s2	
Jumps and Conditional Branches	bgt $s1, $s2, K	slt $at, $s1, $s2 bne $at, $0, K
	sge $s1, $s2, $s3	bne $s3, $s2, foo ori $s1, $0, 1 beq $0, $0, bar foo: slt $s1, $s3, $s2 bar:

Pseudo Instructions: translated to native instructions by Assembler

The SPIM Assembler

Examples of Assembler Directives

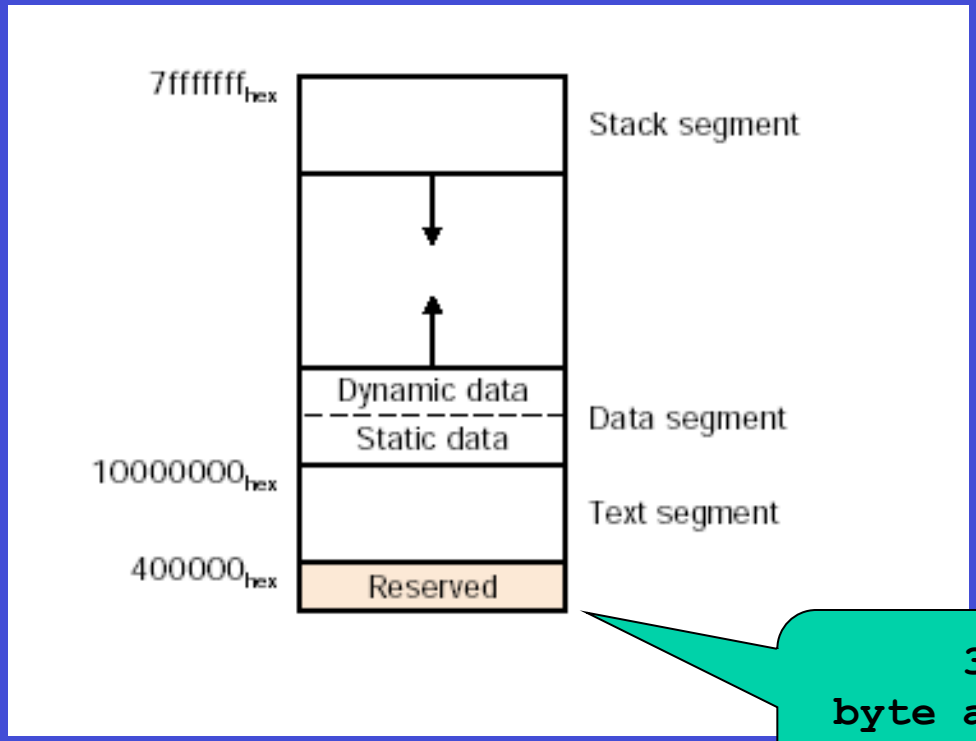
Group	Directive	Function
Memory Segmentation	.data <addr>	Data Segment starting at
	.text <addr>	Text (program) Segment
	.stack <addr>	Stack Segment
	.ktext <addr>	Kernel Text Segment
	.kdata <addr>	Kernel Data Segment
Data Allocation	x: .word <value>	Allocates 32-bit variable
	x: .byte <value>	Allocates 8-bit variable
	x: .ascii "hello"	Allocates 8-bit cell array
Other	.globl x	x is external symbol

Assembler Directives: Provide assembler additional info to generate machine code

Handy MIPS ISA References

- Appendix A: Patterson & Hennessy
- SPIM ISA Summary on class website
- Patterson & Hennessy Back Cover

The MIPS Architecture
Memory Model



32-bit byte addressable address space

Computing Integer Division
Iterative C++ Version

MIPS/SPIM Version

```
int a = 12;
int b = 4;
int result = 0;
main () {
    while (a >= b)
      a = a - b;
      result ++;
    }
  }
}
```

C++

MIPS Assembly Language

```
            .data                       # Use HLL program as a comment
x:          .word       12              # int x = 12;
y:          .word       4               # int y = 4;
res:        .word       0               # int res = 0;
            .globl      main
            .text
main:       la          $s0, x          # Allocate registers for globals
            lw          $s1, 0($s0)     #   x in $s1
            lw          $s2, 4($s0)     #   y in $s2
            lw          $s3, 8($s0)     #   res in $s3
while:      bgt         $s2, $s1, endwhile  # while (x >= y) {
            sub         $s1, $s1, $s2               #   x = x - y;
            addi        $s3, $s3, 1     #   res ++;
            j           while           # }
endwhile:
            la          $s0, x          # Update variables in memory
            sw          $s1, 0($s0)
            sw          $s2, 4($s0)
            sw          $s3, 8($s0)
```

Computing Integer Division
Iterative C++ Version

MIPS/SPIM Version
Input/Output in SPIM

```
int a = 12;
int b = 4;
int result = 0;
main () {
    while (a >= b) {
      a = a - b;
      result ++;
    }
  }
  printf("Result = %
}
```

C++

MIPS Assembly Language

```
            .data                       # Use HLL program as a comment
x:          .word       12              # int x = 12;
y:          .word       4               # int y = 4;
res:        .word       0               # int res = 0;
pf1:        .asciiz     "Result = "
            .globl      main
            .text
main:       la          $s0, x          # Allocate registers for globals
            lw          $s1, 0($s0)     #   x in $s1
            lw          $s2, 4($s0)     #   y in $s2
            lw          $s3, 8($s0)     #   res in $s3
while:      bgt         $s2, $s1, endwhile  # while (x >= y) {
            sub         $s1, $s1, $s2   #   x = x - y;
            addi        $s3, $s3, 1     #   res ++;
            j           while           # }
endwhile:
            la          $a0, pf1        # printf("Result = %d \n");
            li          $v0, 4          # //system call to print_str
            syscall
            move        $a0, $s3
            li          $v0, 1          # //system call to print_int
            syscall
            la          $s0, x          # Update variables in memory
            sw          $s1, 0($s0)
            sw          $s2, 4($s0)
            sw          $s3, 8($s0)
```

SPIM Assembler Abstractions

- Symbolic Labels
  - Instruction addresses and memory locations
- Assembler Directives
  - Memory allocation
  - Memory segments
- Pseudo-Instructions
  - Extend native instruction set without complicating arquitecture
- Macros