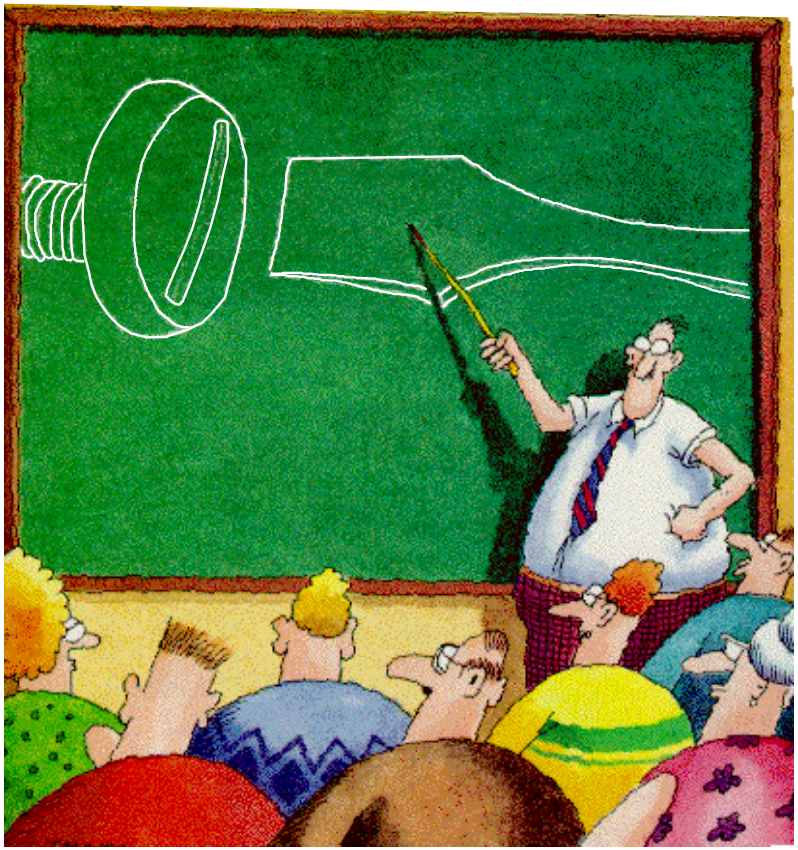
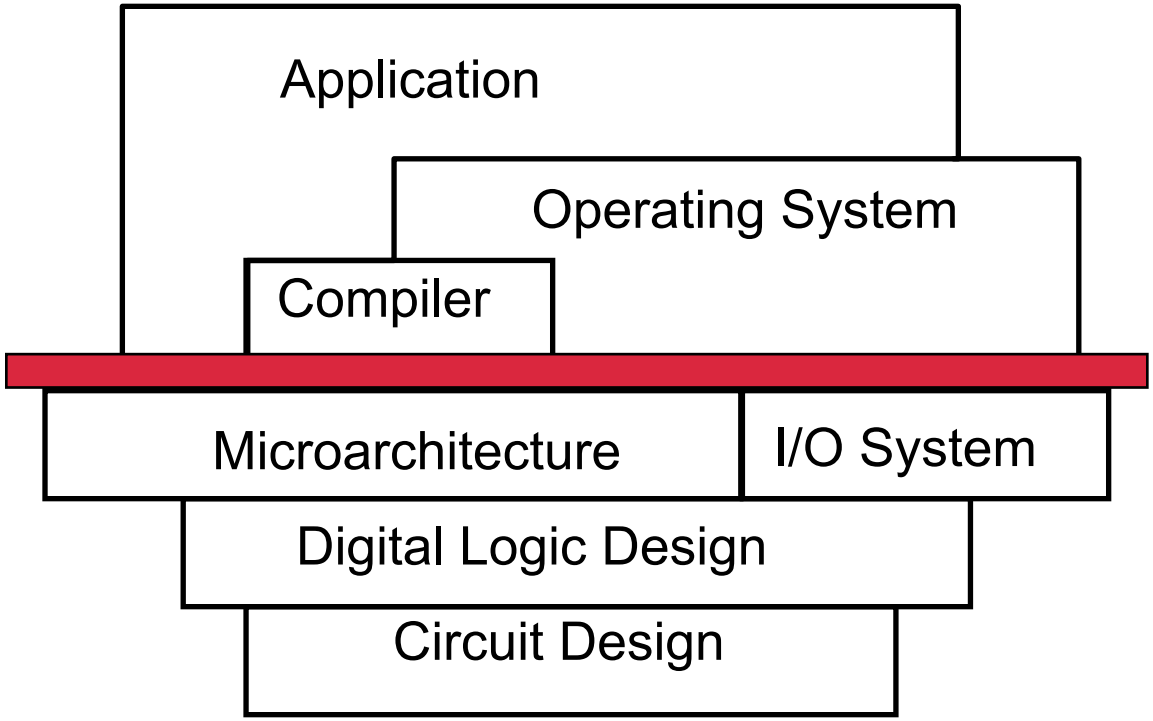
Instruction Set Architecture (I)



Today's Menu:

- **ISA & Assembly Language**
- **Instruction Set Definition**
  - Registers and Memory
  - Arithmetic Instructions
  - Load/store Instructions
  - Control Instructions
  - Instruction Formats
  - Example ISA: MIPS
- **Summary**

Instruction Set Architecture (ISA)



Assembly Language

|||

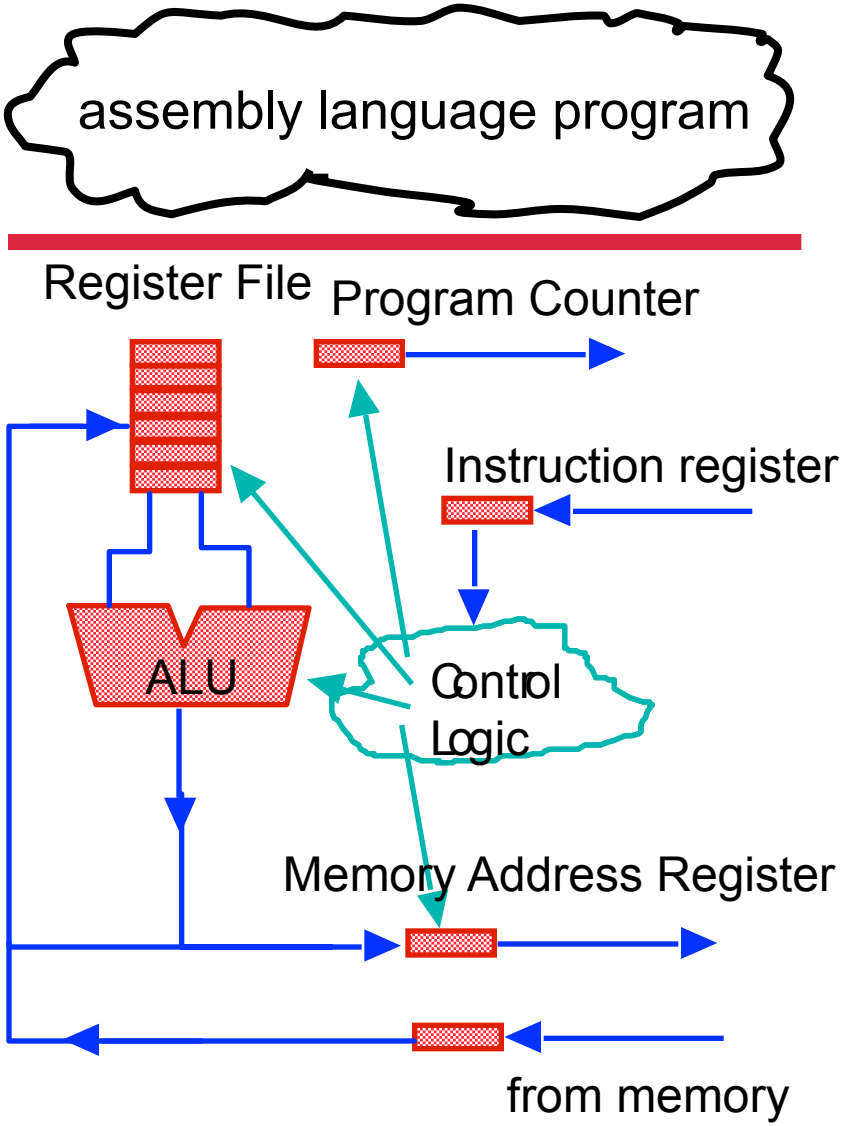
<u>Instruction Set Architecture</u>

|||

Machine Language

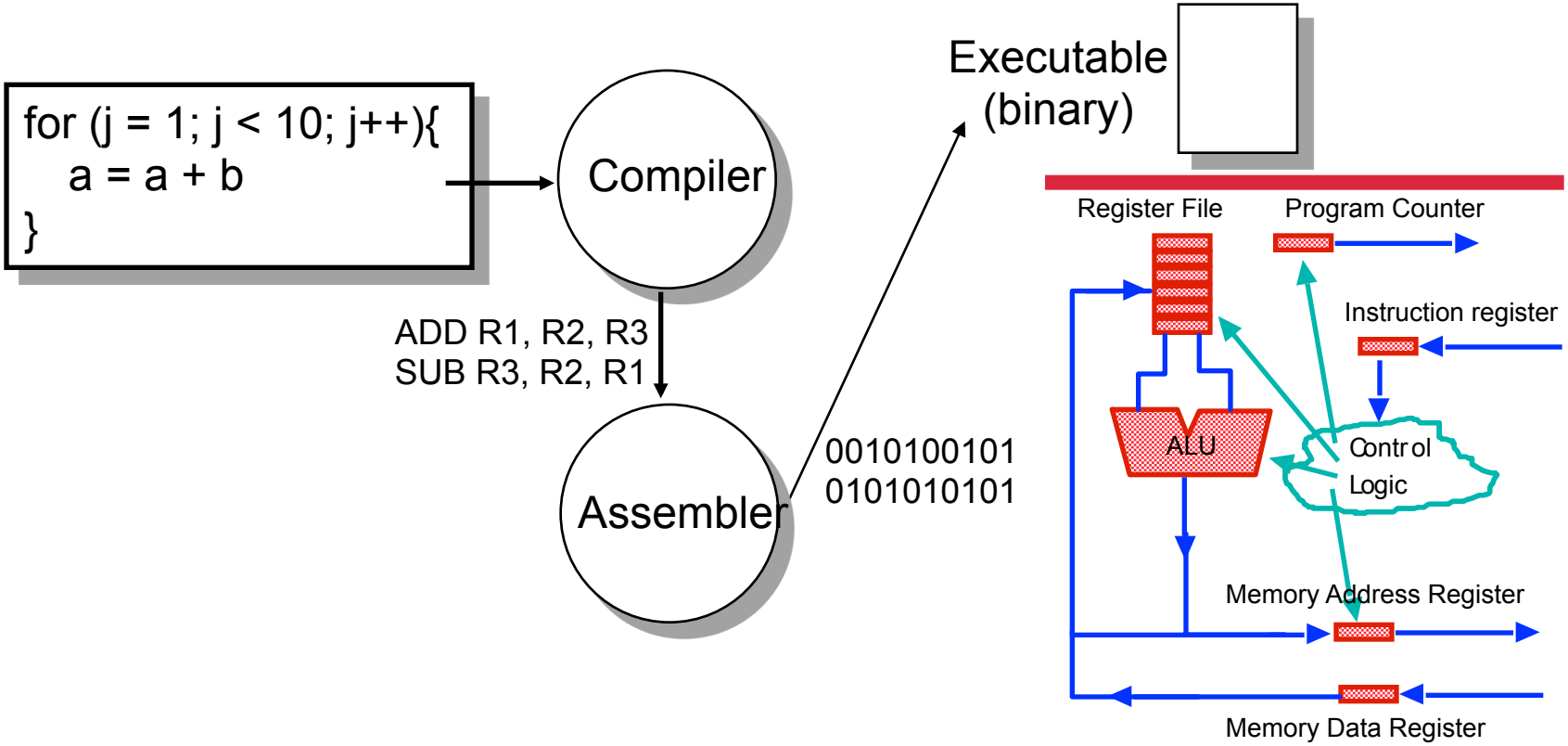
The Big Picture

- **Assembly Language**
  - **Interface** the architecture presents to user, compiler, & operating system
  - "Low-level" instructions that use the datapath & memory to perform basic types of operations
    - arithmetic: add, sub, mul, div
    - logical: and, or, shift
    - data transfer: load, store
    - (un)conditional branch: jump, branch on condition



Software Layers

- High-level languages such as C, C++, FORTRAN, JAVA are translated into assembly code by a **compiler**
- Assembly language translated to machine language by **assembler**



Basic ISA Classes

- **Memory to Memory Machines**
  - Can access memory directly in instructions: e.g., **Mem[0] = Mem[1] + 1**
  - But we need storage for temporaries
  - Memory is **slow** (hard to optimize code)
  - Memory is **big** (need lots of address bits in code → large code)

- **Architectural Registers**
  - registers can hold **temporary variables**
  - registers are (unbelievably) **faster** than memory
  - memory traffic is reduced, so program is sped up
    (since registers are faster than memory)
  - code density improves → smaller code
    (since register named with fewer bits than memory location)

Basic ISA Classes (cont'd)

- **Accumulator** (1 register):
  - 1 address — add A — acc ← acc + mem[A]
  - 1+x address — addx A — acc ← acc + mem[A + x]
- **General Purpose Register File** (Load/Store):
  - 3 address — add Ra Rb Rc — Ra ← Rb + Rc
  - load Ra Rb — Ra ← mem[Rb]
  - store Ra Rb — mem[Rb] ← Ra
- **General Purpose Register File** (Register-Memory):
  - 2 address — add A B — EA(A) ← EA(A) + EA(B)
  - 3 address — add A B C — EA(A) ← EA(B) + EA(C)
- **Stack** (not a register file but an operand stack)
  - 0 address — add — tos ← tos + next (tos=top of stack)
- **Comparison:**
  - Bytes per instruction? Number of Instructions? Cycles per instruction?

Comparing Number of Instructions

- Code sequence for **C = A + B** for four classes of instruction sets:

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R1,B	Load R2,B
Add	Store C	Store C, R1	Add R3,R1,R2
Pop C			Store C,R3

MIPS is one of these: this is what we'll be learning

General Purpose Register Machines Dominate

- **Literally all machines use general purpose registers**
- **Advantages of registers**
  - **registers are faster than memory**
    - **memory traffic is reduced, so program is sped up**
      (since registers are unbelievably faster than memory)
  - **registers can hold variables**
    - **registers are easier for a compiler to use:**
      (A*B) – (C*D) – (E*F) → can do multiplies in any order vs. stack
    - **code density improves**
      (since register named with fewer bits than memory location)

Example: MIPS Assembly Language Notation

- **Generic**

Destination Source Source

```
op    x,  y,  z          # x <-- y op z
```

- **Addition**

```
add   a, b,  c           # a <-- b + c
addi  a, a, 10           # a <-- a + 10
```

- **Subtraction**

```
sub   a, b, c            # a <-- b - c
```

- **f = (g + h) - (i + j)**

```
add   t0,  g,  h         #  t0 <-- g + h
add   t1,  i,  j         #  t1 <-- i + j
sub   f,  t0,  t1        #  f <-- t0 - t1
```

Instruction Set Definition (programming model)

- **Objects** = architected entities = machine state
  - **Registers**
    - General purpose
    - Special purpose (e.g. program counter, condition code, stack pointer)
  - **Memory locations**
    - Linear address space: 0, 1, 2, … , 2^s-1
- **Operations** = instruction types
  - **Data operation**
    - Arithmetic (add, multiply, subtract, divide, etc.)
    - Logical (and, or, xor, not, etc.)
  - **Data transfer**
    - Move (register → register)
    - Load (memory → register)
    - Store (register → memory)
  - **Instruction sequencing**
    - Branch (conditional, e.g., less than, greater than, equal)
    - Jump (unconditional)

Registers and Memory (MIPS)

- **32 registers provided**
  - R0 .. R31
    - You'll sometimes see $ instead of R
      (R6 and $6 both denote register 6)
  - Some special-use registers
    - Register **R0** is hard-wired to **zero**
    - Register **R29** is the **stack pointer**
    - Register **R31** is used for procedure **return address**

- **Arithmetic instructions operands must be registers**

  → This is a load/store machine! Must load all data to registers before using it.

Memory Organization

- **Viewed as a large, single-dimension array, with an address.**
  - **A memory address is an index into the array**
  - **"Byte addressing" means that the index points to a byte of memory.**
- **Bytes are nice, but most data items use larger "words"**
  - **For MIPS, a word is 32 bits or 4 bytes.**

Byte-addressable view of memory

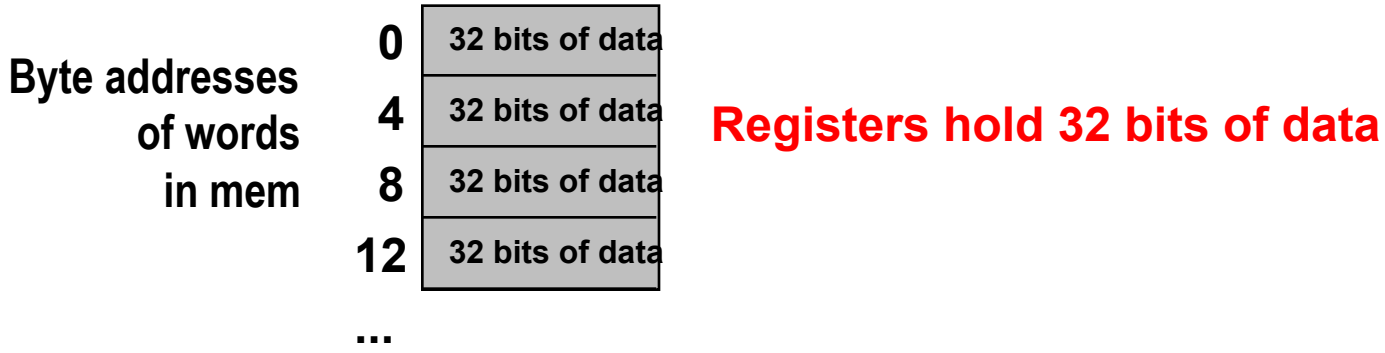
0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
...	

Word-aligned view of memory

0	8 bits of data	8 bits of data	8 bits of data	8 bits of data
4	8 bits of data	8 bits of data	8 bits of data	8 bits of data
8	8 bits of data	8 bits of data	8 bits of data	8 bits of data
12	8 bits of data	8 bits of data	8 bits of data	8 bits of data
16	8 bits of data	8 bits of data	8 bits of data	8 bits of data
20	8 bits of data	8 bits of data	8 bits of data	8 bits of data
24	8 bits of data	8 bits of data	8 bits of data	8 bits of data
...				

Memory Organization

- **Bytes are nice, but most data items use larger "words"**
- **For MIPS, a word is 32 bits or 4 bytes.**

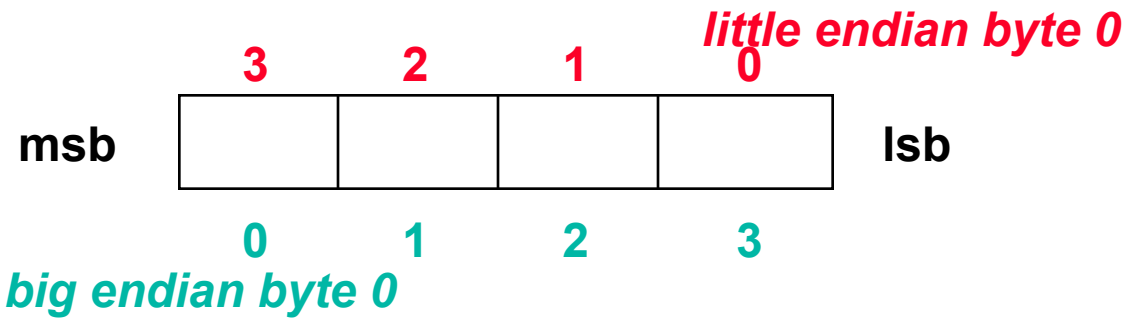


- **32-bit computer:**
  - **2^{32} bytes with byte addresses from 0 to 2^{32}-1**
  - **2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4**
- **Words are aligned**

  what are the least 2 significant bits of a word address?

Addressing Objects: Endianess

- Big Endian: address of most significant byte = word address (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: address of least significant byte = word address (xx00 = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha
- Programmable: set a bit at boot time
  - IBM/Motorola PowerPC

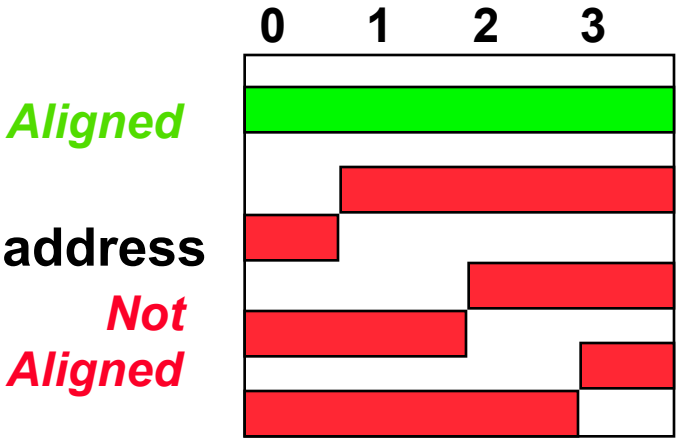


Addressing Objects: Alignment

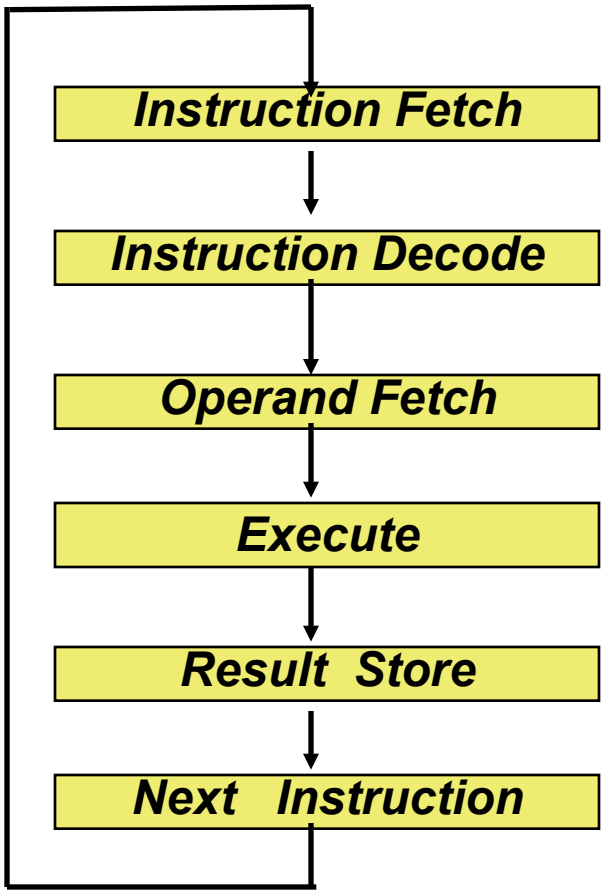
- Hardware may or may not support "unaligned" load/store
  - E.g., Load word from address 0x203

- Possible alternatives:
  - Full hardware support, multiple "aligned" accesses by hardware
  - Hardware trap to OS, multiple "aligned" accesses by software
  - Compiler can guarantee/prevent "unaligned" accesses

Alignment: require that objects fall on address that is multiple of their size.

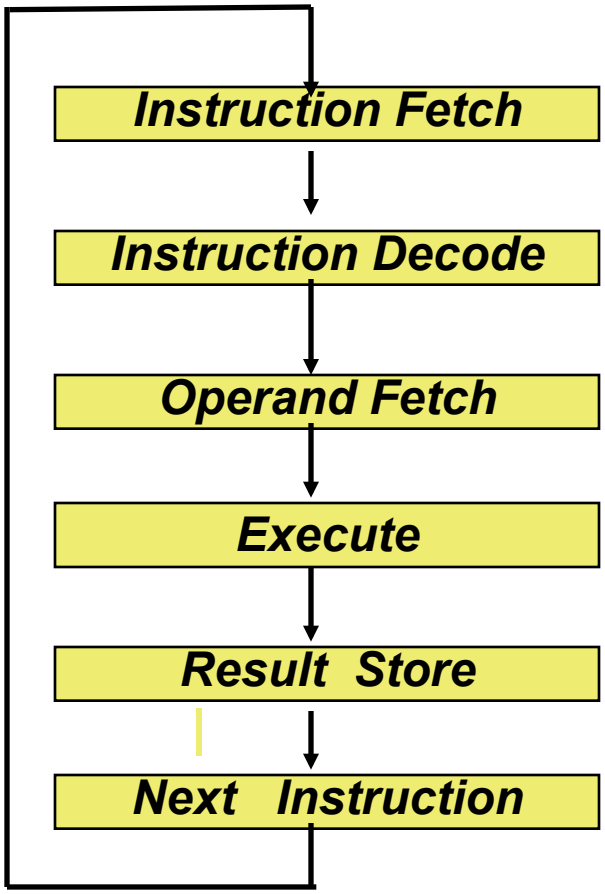


Instruction Cycle (execution model)



- **Sequential Execution Model**
  - Program is a **sequence** of instructions
  - Instructions are **atomic** and executed **sequentially**
- **Stored Program Concept**
  - Program and data **both** are stored in memory
  - Instructions are **fetched** from memory for execution

Instruction Cycle (execution model)



ISA Issues

Get instruction from memory

Instruction Format/Encoding

Addressing Modes

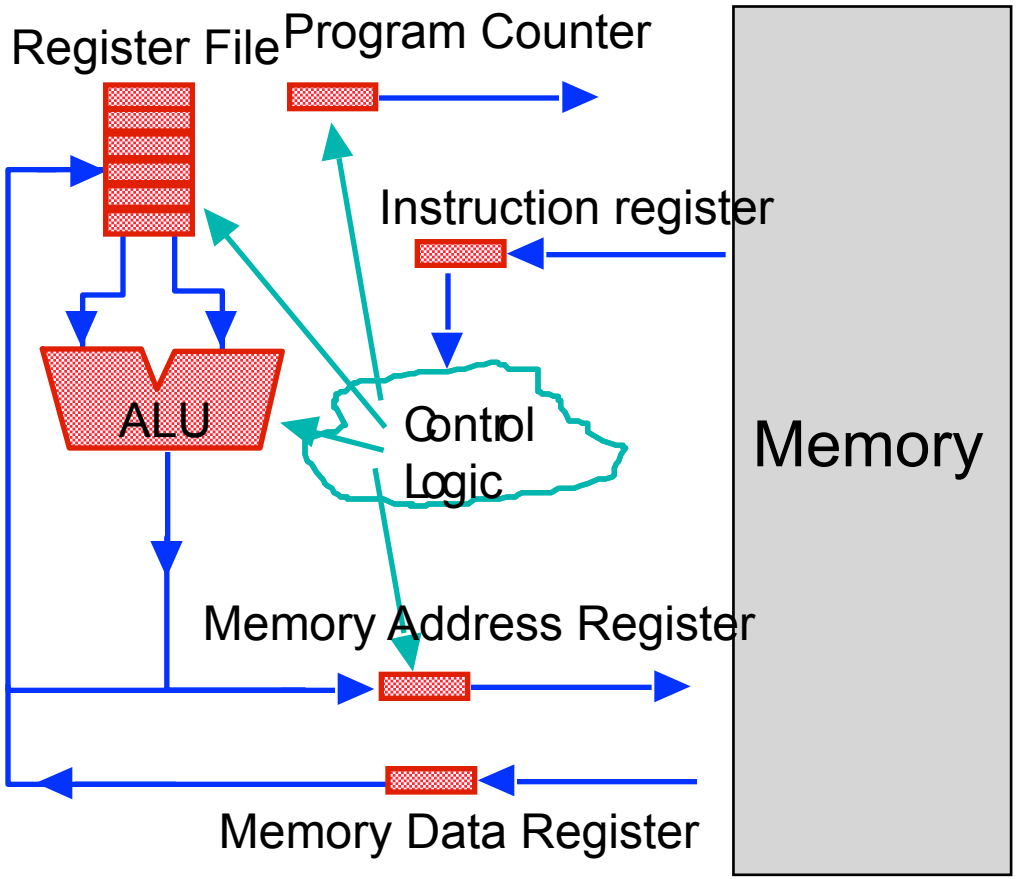
Op-codes and Data Types

Addressing Modes

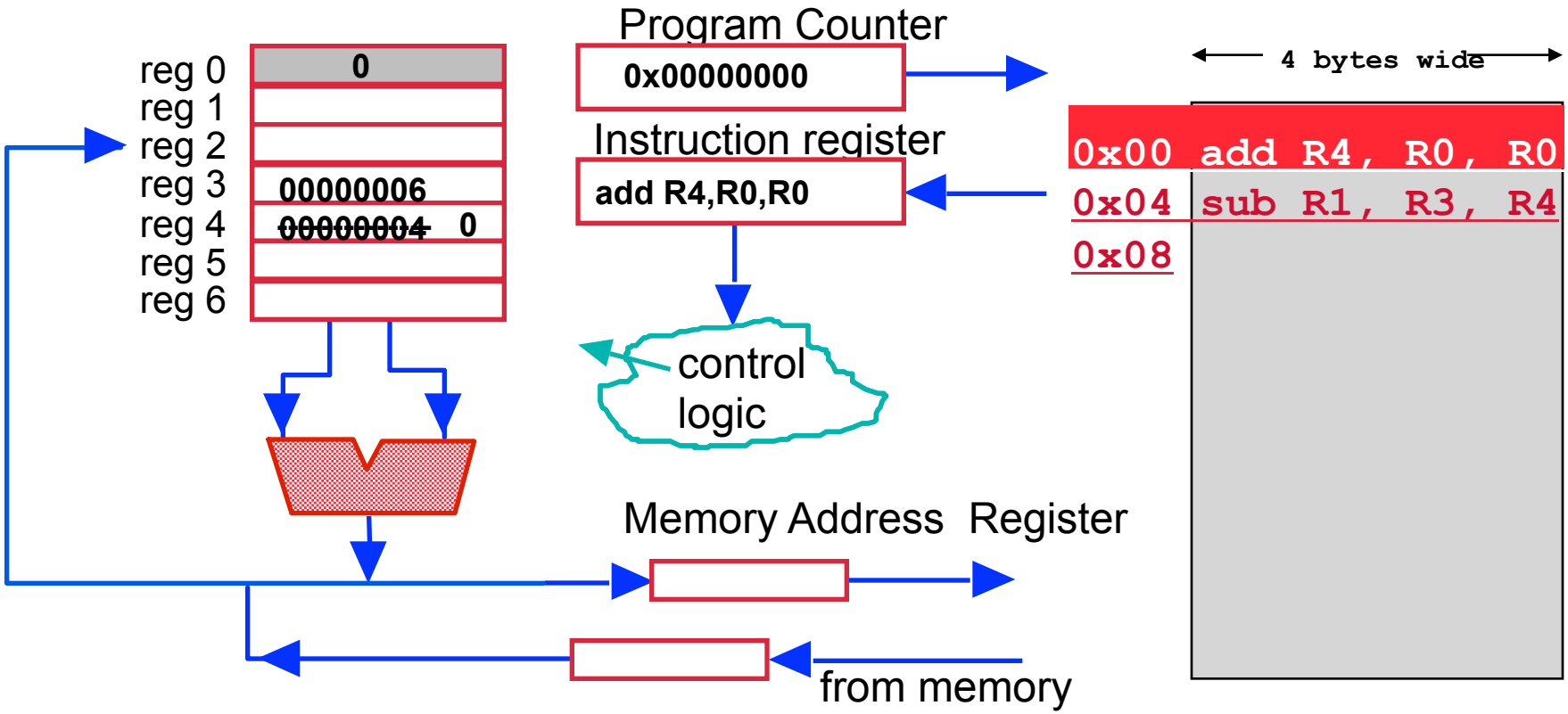
Instruction Sequencing

Executing an Assembly Instruction

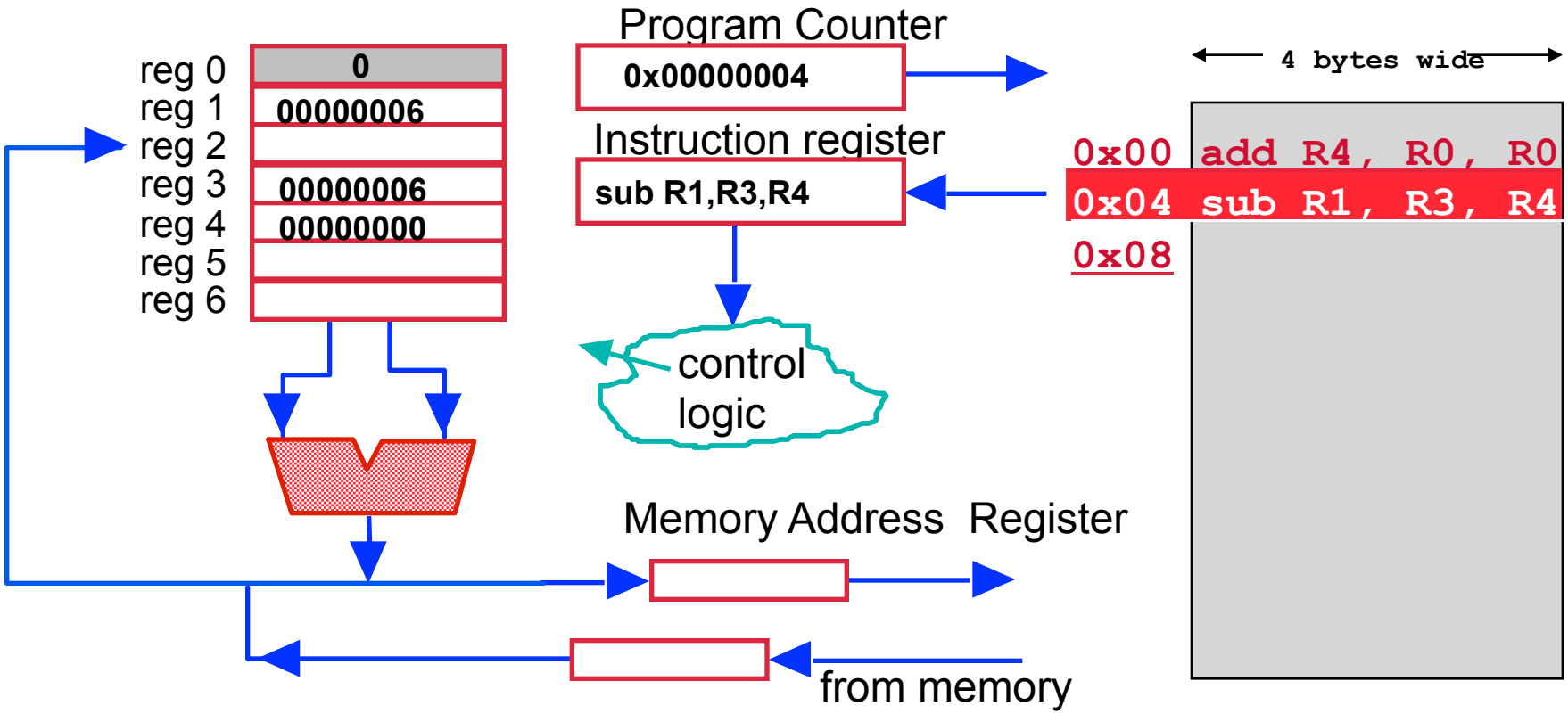
- **Program Counter** holds the instruction address
- Sequencer (FSM) fetches instruction from memory and puts it into the Instruction Register
- Control logic ***decodes*** the instruction and tells the register file, alu and other registers what to do
- If an ALU operation (e.g. add) data flows from register file, through ALU and back to register file



Register File Program Execution

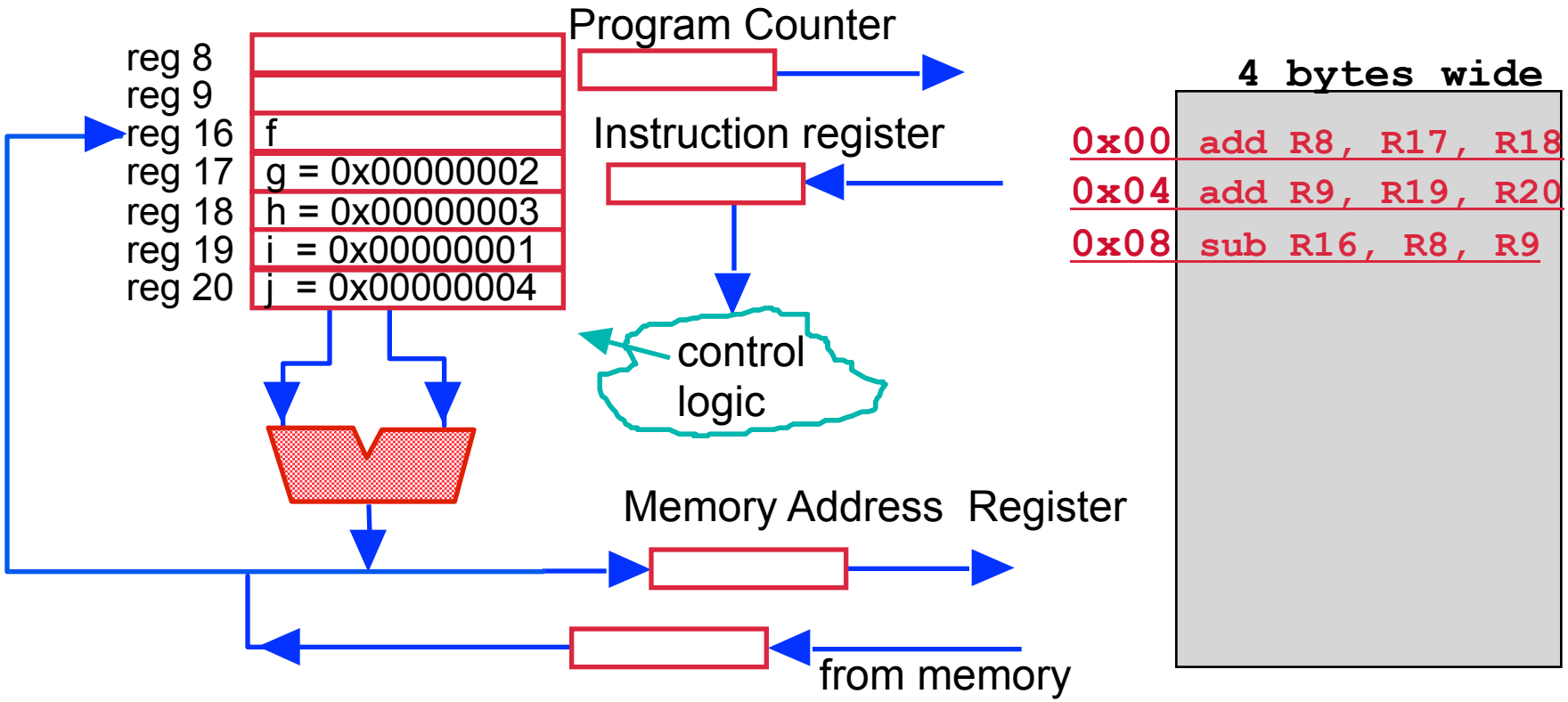


Register File Program Execution



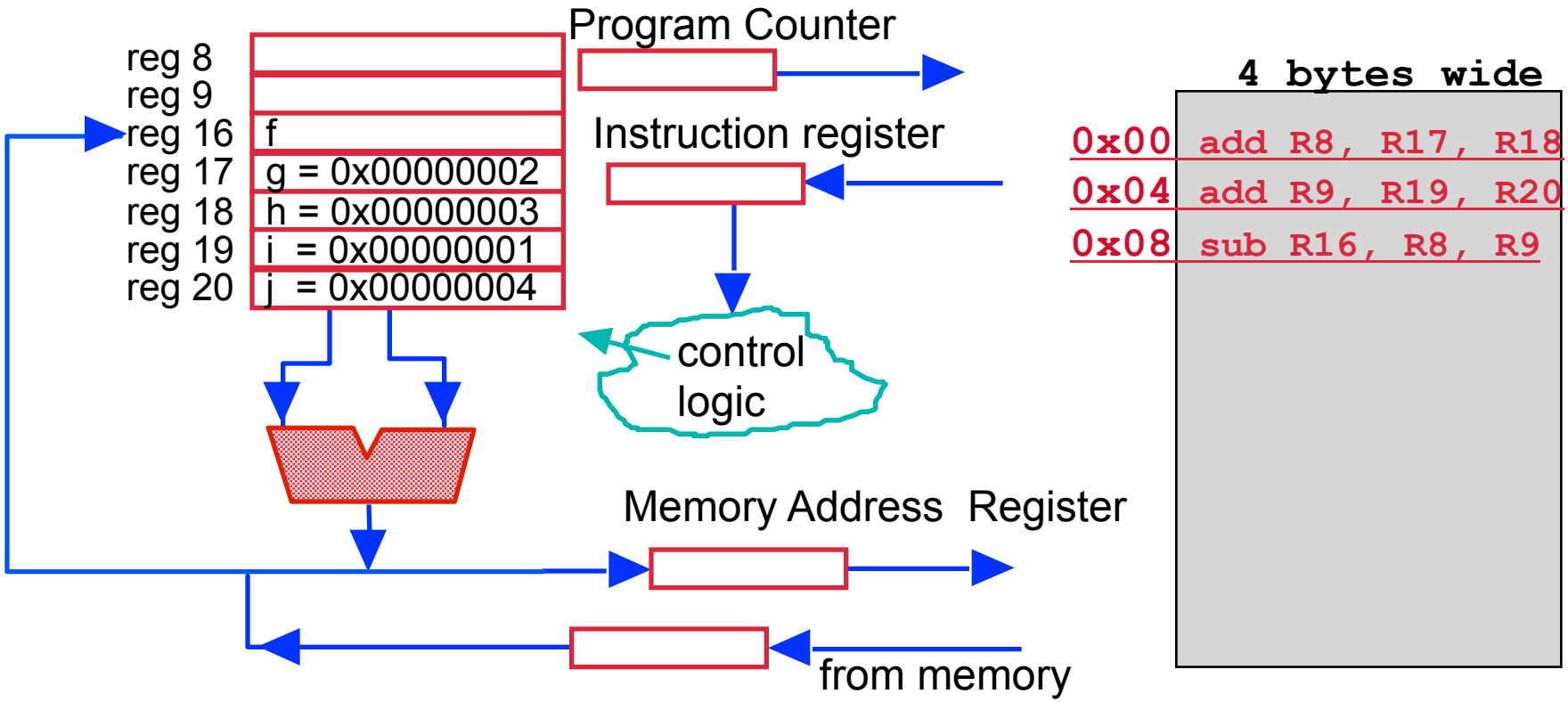
Try This

- **f = (g + h) - (i + j)**
  - R16 == f, R17 == g, R18 == h, R19 == i, R20 == j



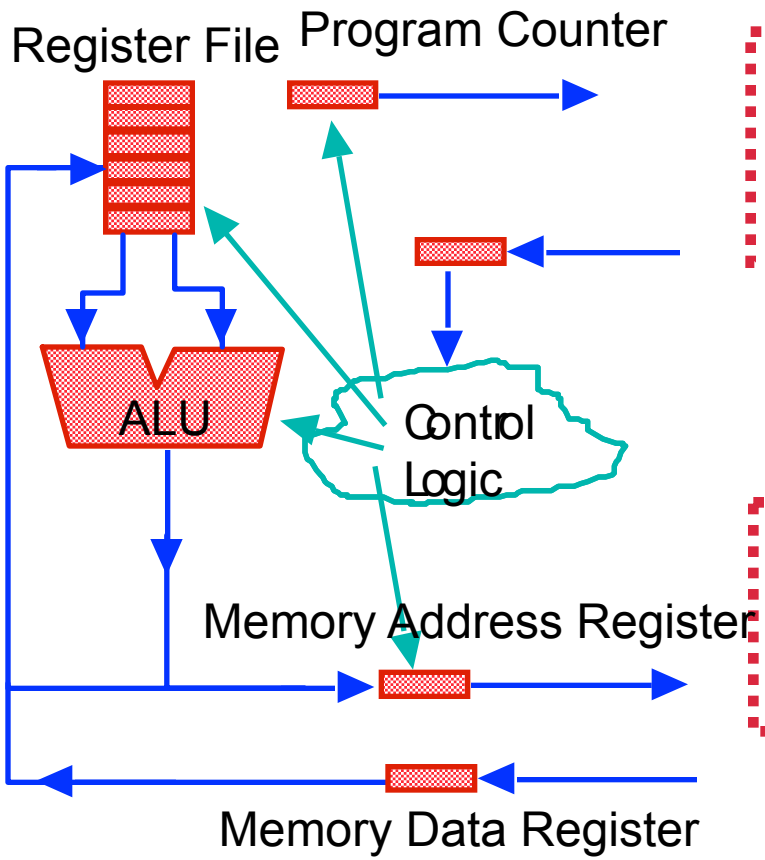
Try This

- **f = (g + h) - (i + j)**
  - R16 == f, R17 == g, R18 == h, R19 == i, R20 == j



Accessing Data

- ALU generated address
- Address goes to ***Memory Address Register***
- When memory is accessed, results are returned to ***Memory Data Register***
- Notice that data and instruction addresses can be the same - both just address memory

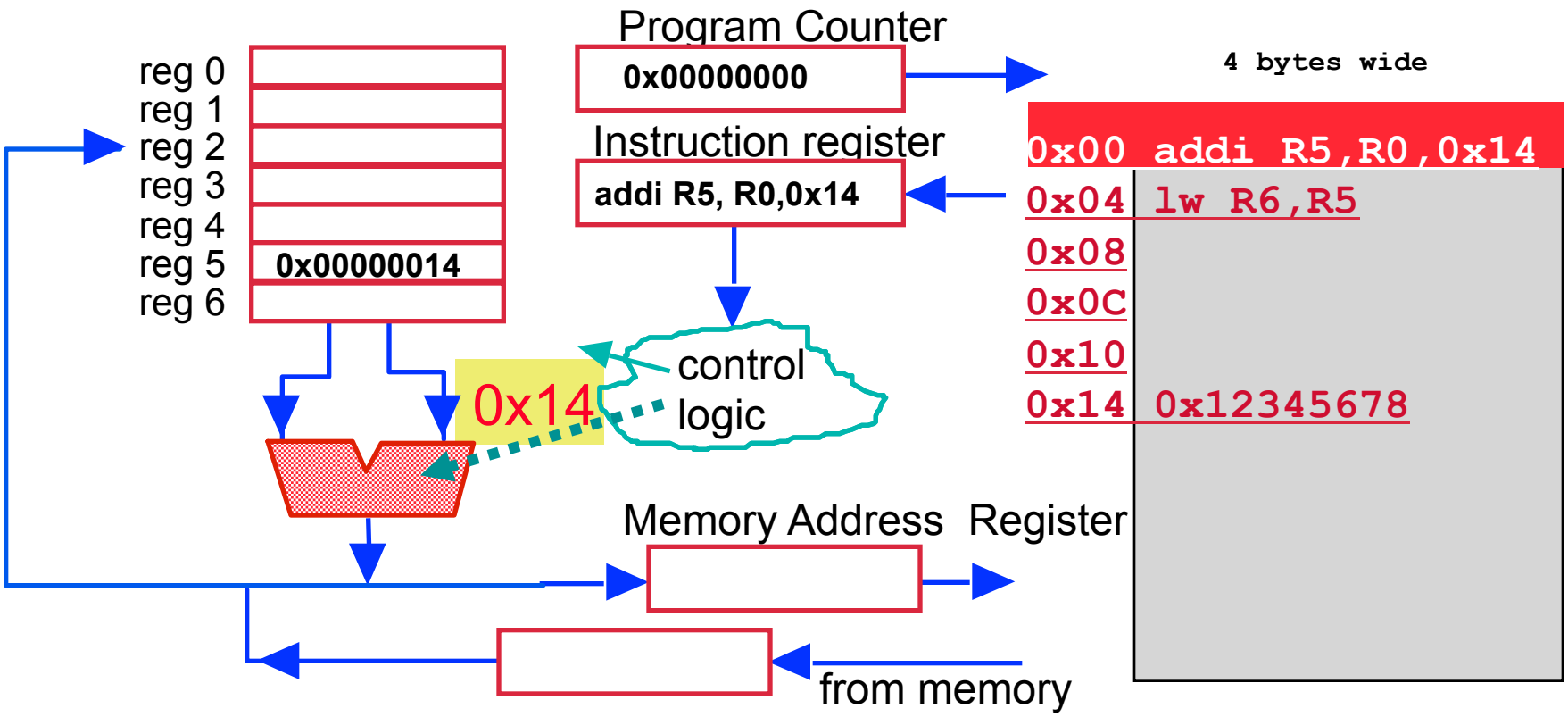


0x00	00101101
0x01	00100001
0x02	00110000
0x03	00001111
0x04	11010101
0x05	01010101
0x06	00101010
0x07	01010101
0x08	11110011
0x09	00111100
0x0A	00001100
0x0B	00000000
0x0C	00011000
0x0D	11111111

Memory Operations - Loads

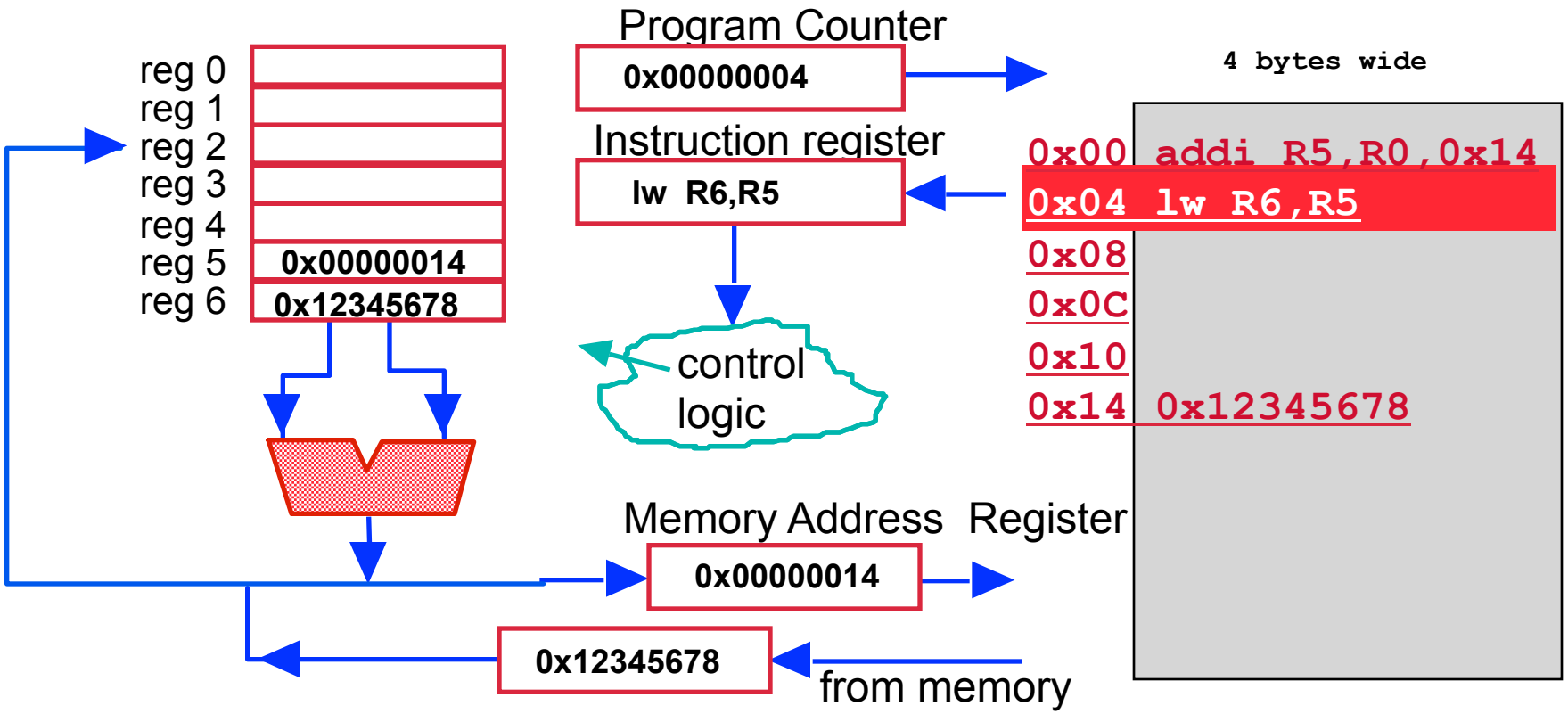
- **Loading data from memory**
  R6 <-- mem[0x14]
  Assume &A = 0x14

addi: adds 16-bit constant to source register



Memory Operations - Loads

- **Loading data from memory**
  R6 <-- mem[0x14]
  Assume &A = 0x14

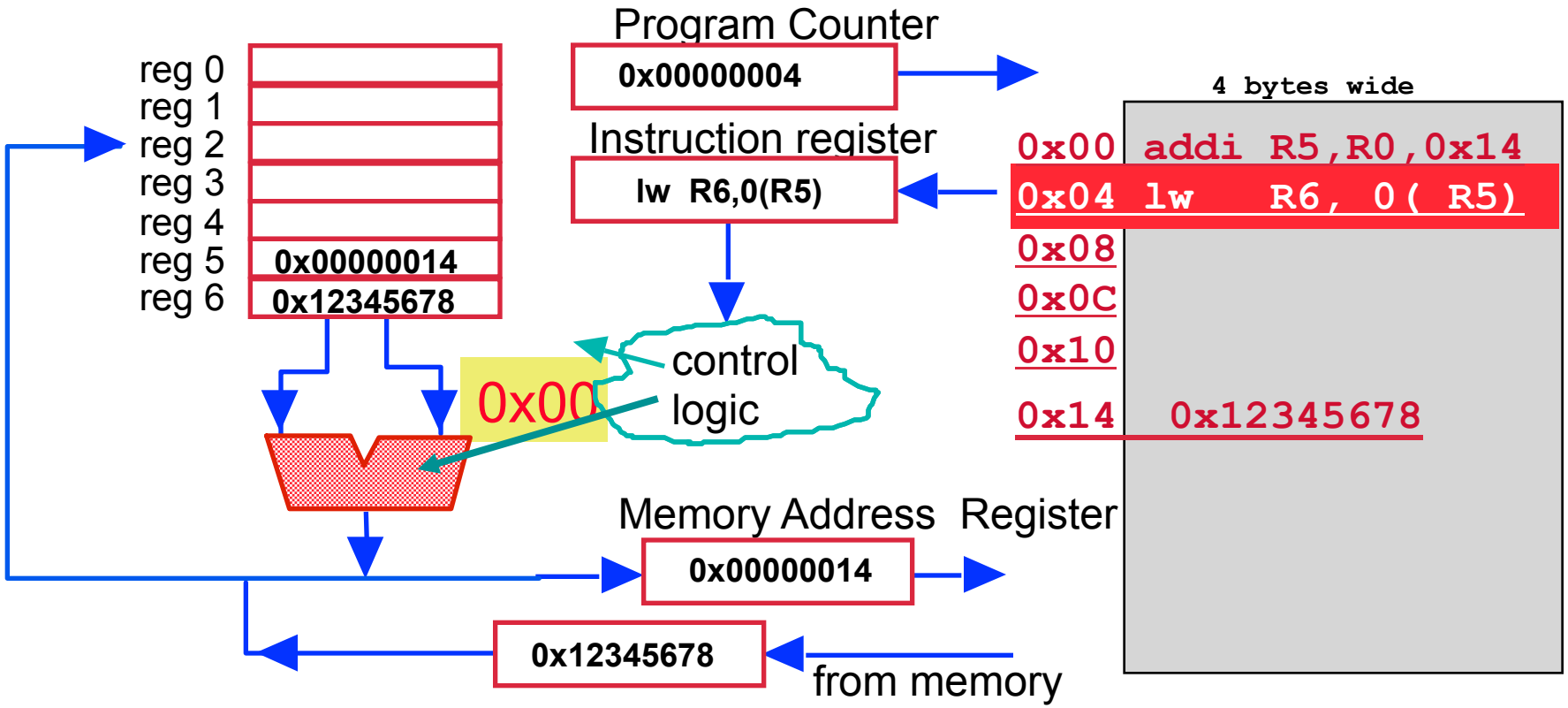


Memory Operations - Loads (con't)

- **Address can also be computed by adding an offset to register**

  LW R6, 0(R5)

  R6 <-- memory[0 + R5]

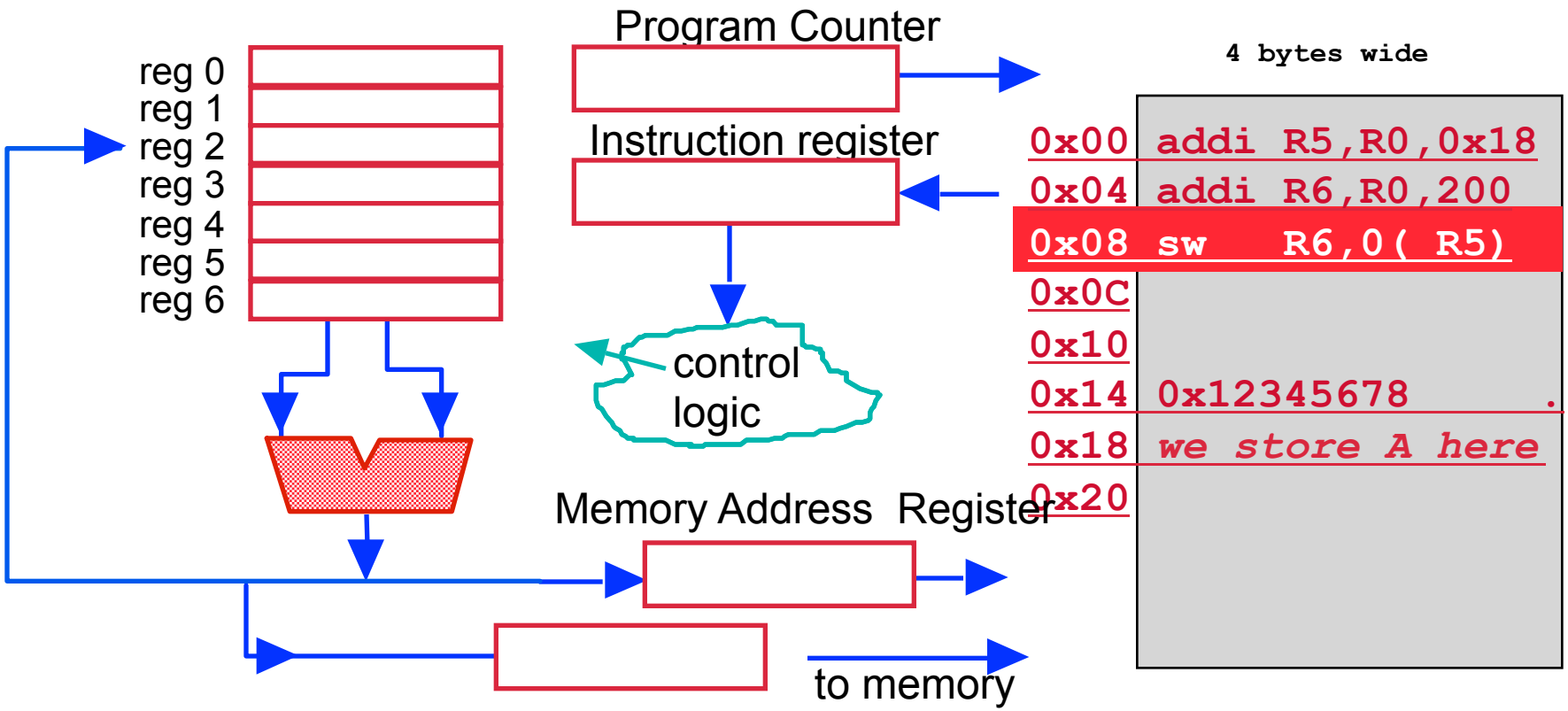


Try This: Memory Operations - Stores

- **Storing data to memory works essentially the same way**

  A = 200; let's assume &A = 0x18

  mem[0x18] <-- 200

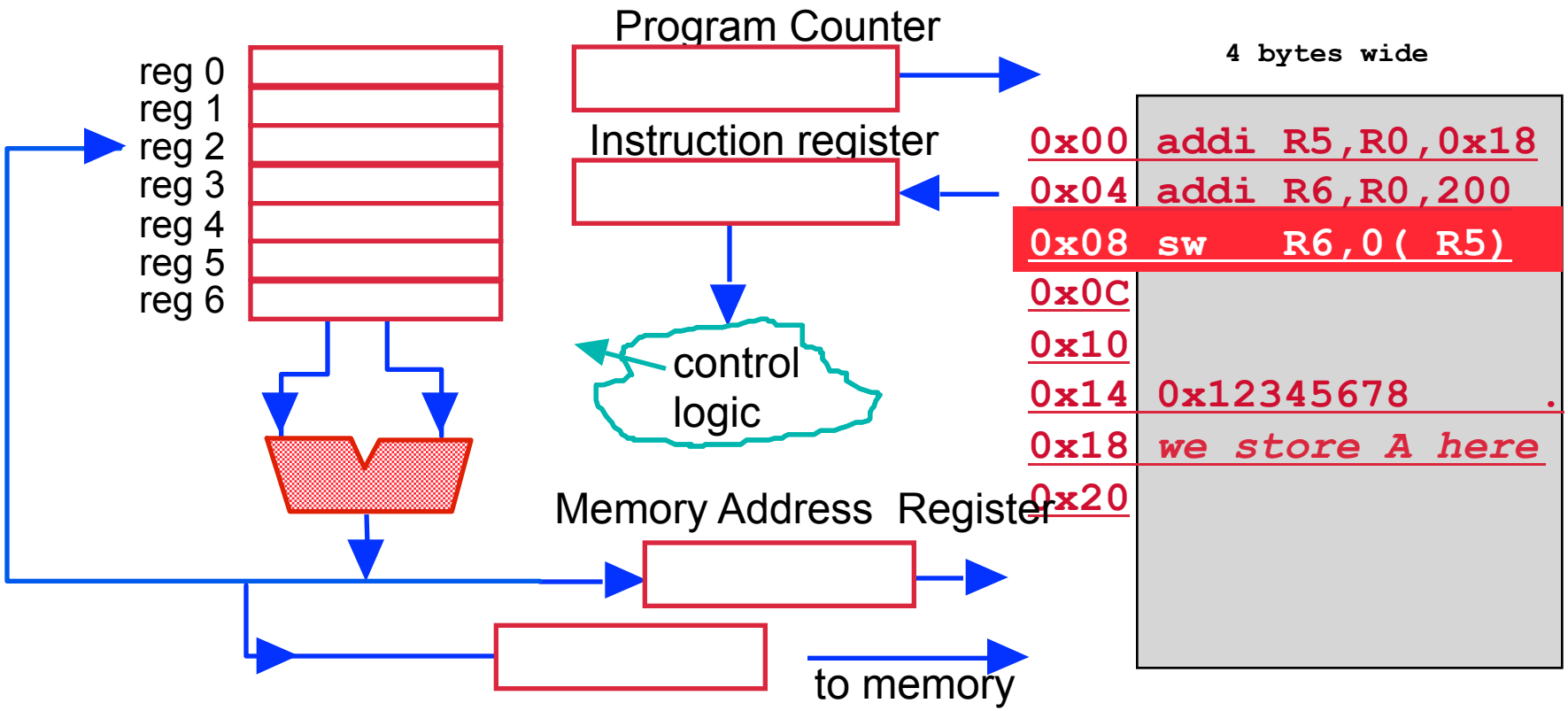


Try This: Memory Operations - Stores

- **Storing data to memory works essentially the same way**

  A = 200; let's assume &A = 0x18

  mem[0x18] <-- 200



Instruction Format (Machine Language)

- **add R8, R17, R18**
  - is stored in binary format as
    00000010 00110010 01000000 00100000
  - MIPS lays out instructions into "fields"

31 – 26	25 – 21	20 – 16	15 – 11	10 – 6	5 – 0
000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

  - **op** operation of the instruction
  - **rs** first register source operand
  - **rt** second register source operand
  - **rd** register destination operand
  - **shamt** shift amount
  - **funct** function (select type of operation)
    - add = 32_{10}
    - sub = 34_{10}

Why are there 5 bits in the register field?

MIPS Instruction Formats

- **More than more than one format for instructions, usually**
  - Different kinds of instructions need different kinds of fields, data
  - Example: 3 MIPS instruction formats

Name	Fields						Comments
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shmt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer (load/store), branch, immediate format
J-format	op	target address					Jump instruction format

Questions:

I-format: **How big an immediate can you have?**
Is that big enough? (What's the maximum value?)

J-format: **How far can you jump in instructions?**

Constants

- **Small constants are used quite frequently (50% of operands)**

  e.g., A = A + **5**;
  B = B + **1**;
  C = C - **18**;

- **Solutions? Why not….**
  - …just put 'typical constants' in memory and load them.
  - …just create hard-wired registers (like $zero) for constants like one.

- **MIPS Instructions:**

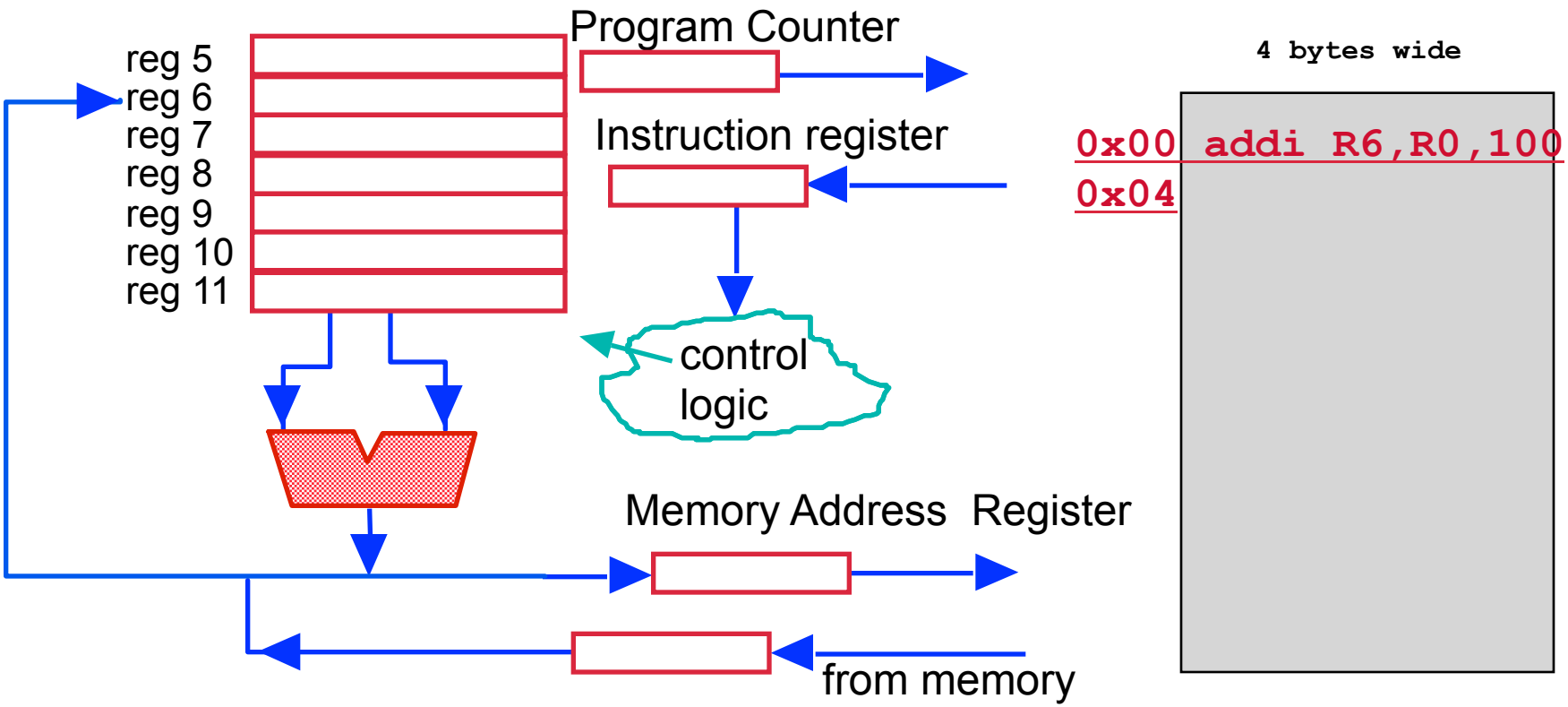
```
addi $29, $29, 4
slti $8,  $18, 10
andi $29, $29, 6
ori  $29, $29, 4
```

How do we get these constants in a efficient way?

- **How do we make this work?**

Loading Immediate Values

- **How do we put a constant (immediate) value into a register?**
  - Put the value **100** into register **R6**: **R6 <- R0** + **100 = 0+100 = 100**



MIPS Machine Language

- **From back cover of Patterson and Hennessy**

Name	Format	Example						Comment s
		6b it s	5 bits	5 bits	5 bits	5 bits	6b it s	
add	R	0	2	3	1	0	3 2	add $1,$2,$3
sub	R	0	2	3	1	0	3 4	sub $1,$2,$3
addi	I	8	2	1	10 0			addi $1,$2, 10 0
addu	R	0	2	3	1	0	3 5	addu $1,$2, $3
and	R	0	2	3	1	0	3 6	and $1,$2, $3
or	R	0	2	3	1	0	3 7	or $1, $2, $3
lw	I	3 5	2	1	10 0			lw $1, 10 0 ($2)
sw	I	4 3	2	1	10 0			sw $1, 100 ($2)
beq	I	4	1	2	25			beq $1, $2, 100
j	J	2	25 0 0					j 10 0 0 0

Loading Immediate Values

- **What's the largest immediate value that can be loaded into a register?**

Name	Fields						Comments
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shmt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, immediate format
J-format	op	target address					Jump instruction format

- **But, then, how do we load <u>larger</u> numbers?**

Load Upper Immediate

- **Example: lui R8, 255**

31 – 26	25 – 21	20 – 16	15 – 11	10 – 6	5 – 0
001111	00000	01000	0000 0000	1111	1111
op	rs	rt	immediate		

Transfers the immediate field into the register's top (upper) 16 bits and fills the register's lower 16 bits with zeros

R8[31:16] <-- IR[15:0] ; top 16 bits of R8 <-- bottom 16 bits of the IR
R8[15:0] <-- 0 ; bottom 16 bits of R8 are zeroed

	31 – 16	15 – 0
Reg. 8	0000 0000 1111 1111	0000 0000 0000 0000

Larger Constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use 2 instructions: first, new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

filled with zeros

1010101010101010	0000000000000000

- Second, must then get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```

	1010101010101010	0000000000000000
ori	0000000000000000	1010101010101010
	1010101010101010	1010101010101010

Control (Instruction Sequencing)

- **Decision making instructions**
  - These instructions **alter** the **"control flow"**
  - Means they **change** the "next" instruction to be executed

- **MIPS conditional branch instructions:**

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- **Example:** **if (i==j) h = i + j;**

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label:        ....
```

Branch here if $s0 != $s1

Control (Instruction Sequencing)

- **MIPS unconditional branch instructions:**

```
j  label
```

- **Example:**

```
if (i!=j)
    h=i+j;
else
    h=i-j;
```

```
beq $s4, $s5, Lab1
add $s3, $s4, $s5
j Lab2
Lab1: sub $s3, $s4, $s5
Lab2: ...
```

Go here if $s4 != $s5

Go here always

- ***OK, so with these--Can you build a simple for(...) {...} loop?***

Branch Instructions

- They exist because we need to change the program counter

  if (a == b) c = 1;
  else c = 2;

- bne (branch not equal) compares regs and branches if regs "!="
- j (jump) goto address, unconditional branch

```
Assume  R5 == a;  R6 == b;  R7 == c

Add    Mnemonic                Description (comment)
0x00   bne    R5, R6, 0x0C     ; if ( R5 != R6) goto 0x0C
0x04   addi   R7, R0, 1        ; R7 <-- 1 + 0
0x08   j      0x10             ; goto 0x10
0x0C   addi   R7, R0, 2        ; R7 <-- 2 + 0
0x10
```

Branch Instructions

- **Branch instructions are used to implement C-style loops**

```
for ( j = 0; j < 10; j++){
    b = b +  j;
}
```

```
assume R5 == j;  R6 == b;
```

Add	Mnemonic	Description (comment)
0x00	addi R5, R0, 0	; R5 ← 0 + 0
0x04	addi R1, R0, 10	; R1 ← 0 + 10
0x08	beq R5, R1, 0x18	; if (R5 == 10) goto 0x18
0x0C	add R6, R6, R5	; R6 ← R6 + R5
0x10	addi R5, R5, 1	; R5 ← R5 + 1
0x14	j 0x08	; goto 0x08
0x18	…	; pop out of loop, continue

Addresses in Branches and Jumps

- **Instructions:**

`bne $t4,$t5,Label`	**Next instruction is at Label if `$t4 != $t5`**
`beq $t4,$t5,Label`	**Next instruction is at Label if `$t4 == $t5`**
`j Label`	**Next instruction is at Label**

- **Formats:**

I	op	rs	rt	16 bit address
J	op	26 bit address		

- **Hey, the addresses in these fields are not 32 bits !**

  — How do we handle this?

Addresses in Branches

- **Instructions:**

`bne $t4,$t5,Label`	**Next instruction is at Label if `$t4 != $t5`**
`beq $t4,$t5,Label`	**Next instruction is at Label if `$t4 == $t5`**
`j Label`	**Next instruction is at Label**

- **Formats:**

I	op	rs	rt	16 bit address
J	op	26 bit address		

- **Could specify a register and add it to this 16b address**
  - Use the PC + (16-bit relative **word** address to find the address to jump to)
  - Note: most branches are **local** ("principle of locality")
- **Jump instructions just use the high order bits of PC**
  - 32-bit jump address = 4 (most significant) bits of PC concatenated with 26-bit word address (or 28-bit byte address)
  - Address boundaries of 256 MB

Branch Instructions

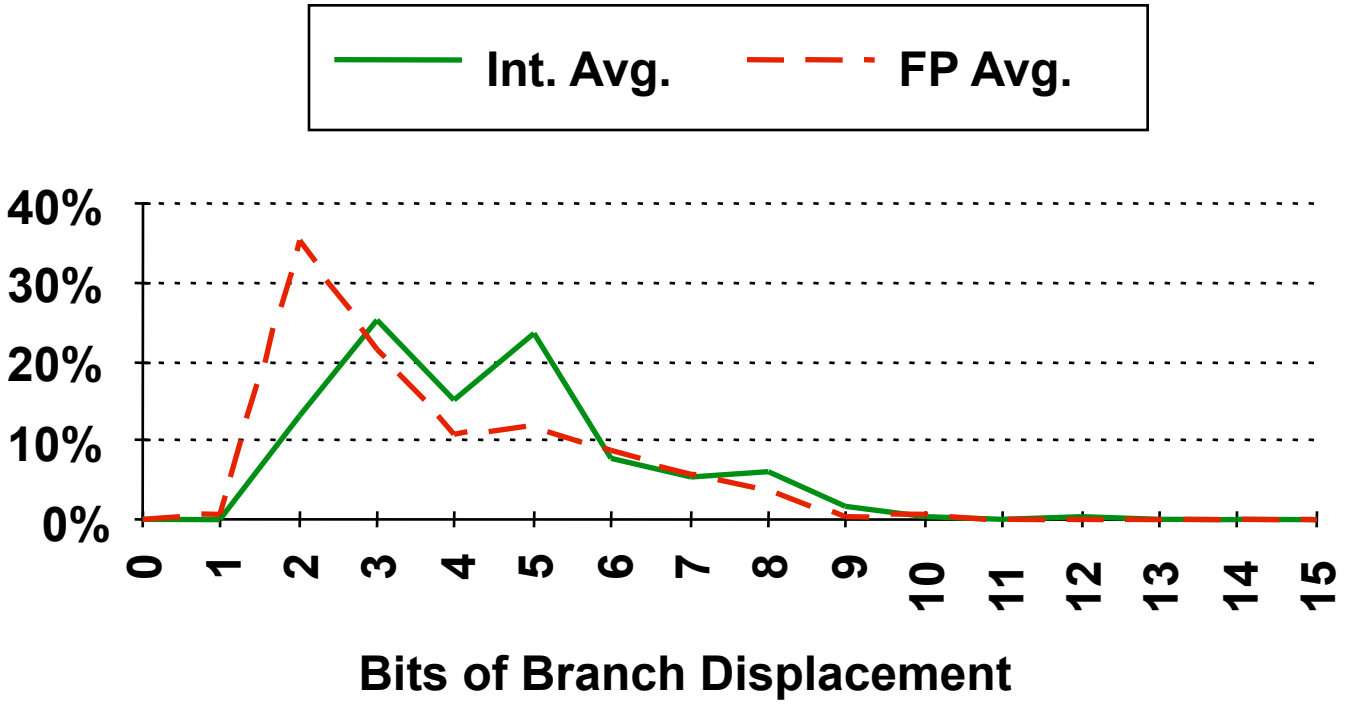
- **Example**

```
for ( j = 0; j < 10; j++){
    b = b +  j;
}
```

```
assume R5 == j;  R6 == b;
Add     Mnemonic              Description (comment)
0x00    addi R5, R0, 0        ; R5 <-- 0 + 0
0x04    addi R1, R0, 10       ; R1 <-- 0 + 10
0x08    beq  R5, R1, 0x18     ; if ( R5 == 10) goto 0x18
0x0C    add  R6, R6, R5       ; R6 <-- R6 + R5
0x10    addi R5, R5, 1        ; R5 ← R5 + 1
0x14    j    0x08             ; goto 0x08
0x18    …                     ; pop out of loop, continue
```

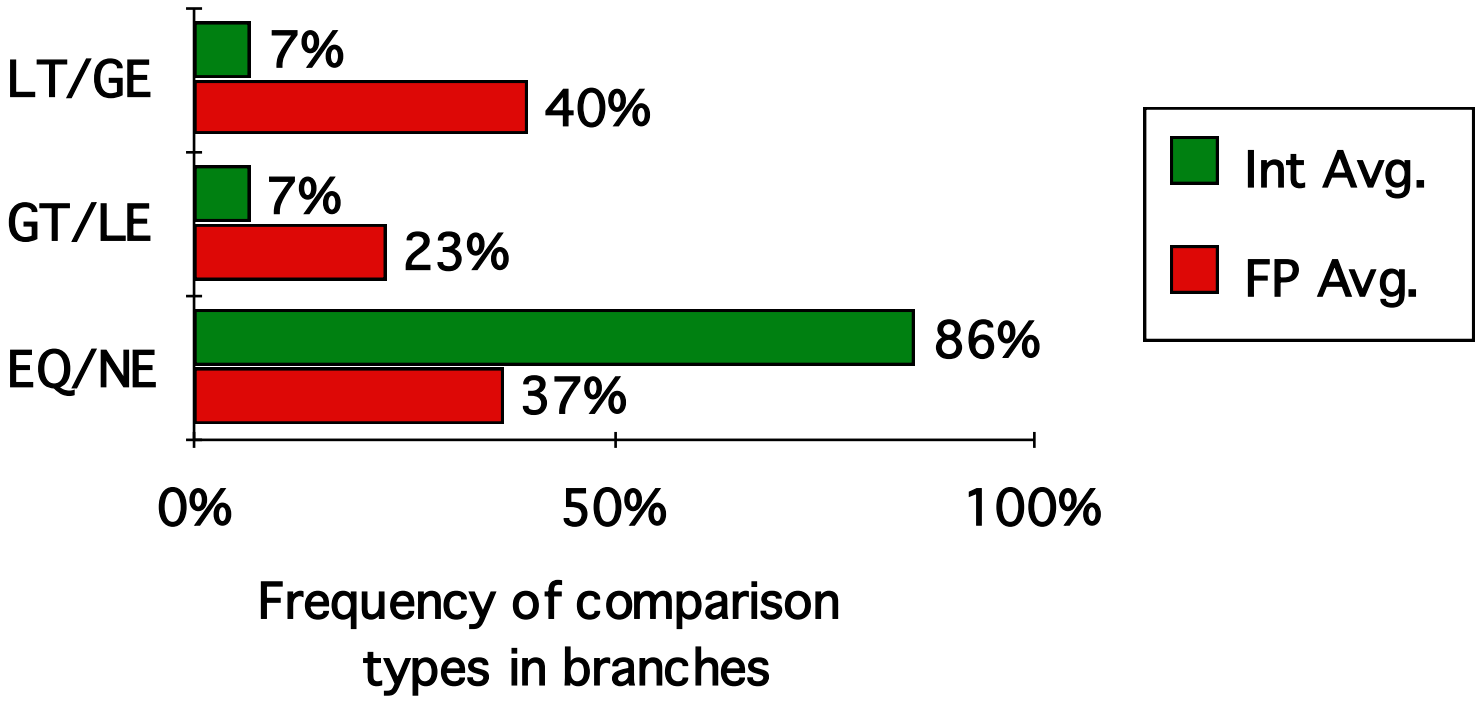
0x08	4	5	1	3	PC = PC + 4 + (3<<2)
0x14	2			2	PC=[PC(31:28):2]<<2

Conditional Branch Distance



Conditional Branch Addressing

- PC-relative since most branches are relatively close to the current PC address
- At least 8 bits suggested (± 128 instructions)
- Compare Equal/Not Equal most important for integer programs (86%)

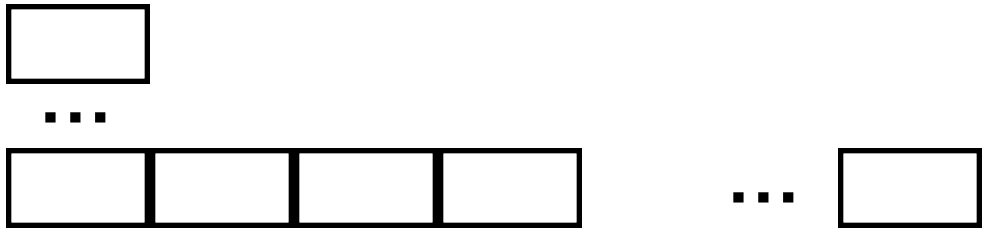


Full MIPS Instruction Set

add	add $1, $2, $3	$1 = $2+$3
sub	sub $1,$2, $3	$1 = $2 - $3
add immediate	addi $1, $2, 100	$1 = $2 + 100
add unsigned	addu $1, $2, $3	$1 = $2 + $3
subtract unsigned	subu $1, $2, $3	$1 = $2 - $3
add imm. unsigned	addiu $1, $2, 100	$1 = $2 + 100
multiply	mult $2, $3	hi, lo = $2 * $3
multiply unsigned	multu $2, $3	hi, lo = $2 * $3
divide	div $2, $3	lo = $2/$3, hi = $2 mod $3
divide unsigned	divu $2, $3	lo = $2/$3, hi = $2 mod $3
move from hi	mfhi $1	$1 = hi
move from low	mflo $1	$1 = lo
and	and $1, $2, $3	$1 = $2 & $3
or	or $1, $2, $3	$1 = $2 | $3
and immediate	andi $1, $2, 100	$1 = $2 & 100
or immediate	ori $1, $2, 100	$1 = $2 | 100
shift left logical	sll $1, $2, 10	$1 = $2 << 10
shift right logical	srl $1, $2, 10	$1 = $2 >> 10
load word	lw $1, 100($2)	$1 = memory[$2+100]
store word	sw $1, 100($2)	memory[$2 + 100] = $1
load upper immediate	lui $1, 100	$1 = 100 * 2^16
branch on equal	beq $1, $2, 100	if ($1 == $2) go to PC + 4 + 100*4
branch on not equal	bne $1, $2, 100	if ($1 != $2) go to PC + 4 + 100*4
set on less than	slt $1, $2, $3	if ($2 < $3) $1 = 1 else $1 = 0
set less than immediate	slti $1, $2, 100	if ($2 < 100) $1 = 1 else $1 = 0
set less than unsigned	sltui $1, $2, $3	if ($2 < $3) $1 = 1 else $1 = 0
set less than immediate unsigned	sltui $1, $2, 100	if ($2 < 100) $1 = 1 else $1 = 0
jump	j 10000	goto 10000
jump register	jr $31	goto $31
jump and link	jal 100000	$31 = PC + 4; goto 10000

Generic Examples of Instruction Format Widths

Variable:



Better for generating compact code

Fixed:

Easier to use for generating assembly code

Summary of Instruction Formats

- If code size is most important, use variable length instructions
- If performance is most important, use fixed length instructions
- Recent embedded machines (ARM, MIPS) have an optional mode to execute subset of 16-bit wide instructions (Thumb, MIPS16); per procedure, decide which one of performance or density is more important

Observation

- **"Simple" computations, movements of data, etc., are not always "simple" in terms of a single, obvious assembly instruction**
  - Often requires a sequence of even more primitive instructions
  - One options is to try to "anticipate" every such computation, and try to provide an assembly instruction for it
    (Complex Instruction Set Computing = CISC)
    - **PRO**: assembly programs are easier to write by hand
    - **CON**: hardware gets really, really complicated by instructions used very rarely. Compilers might be harder to write
  - Other option is to provide a small set of essential primitive instructions
    (Reduced Instruction Set Computing = RISC)
    - **CON**: anything in a high level language turns into LOTS of instructions in assembly language
    - **PRO**: hardware and compiler become easier to design, cleaner, easier to optimize for speed, performance

Summary

- **Architecture = what's visible to the program about the machine**
  - Not everything in the deep implementation is "visible"
- **Microarchitecture = what's invisible in the deep implementation**

A big piece of the ISA = assembly language structure

  - Primitive instructions, execute sequentially, atomically
  - Issues are formats, computations, addressing modes, etc

- **We do one example in some detail: MIPS (from P&H Chap 3)**
  - A RISC machine, its virtue is that it is pretty simple
  - Can pick up the assembly language without too much memorization
- **Next lecture**
  - Addressing modes