Instruction Set Design and Architecture

COE608: Computer Organization and Architecture

Dr. Gul N. Khan

http://www.ee.ryerson.ca/~gnkhan

Electrical and Computer Engineering

Ryerson University

Overview

- Computer Organization and Instructions
- ISA Classes and MIPS Instruction Set
  - MIPS arithmetic
  - MIPS Memory Organization and Registers
  - Branch Instructions and Control flow
- MIPS Instruction Format
- MIPS Instruction Set
- PowerPC: Alternative Architecture

Chapter 2 of the Text

Processor Instructions

- Native Language of a Computer CPU
- A primitive language with no sophisticated control flow

  As compared to high-level languages:

  C, Java, and FORTRAN.
- Very restrictive

  e.g. MIPS-Processor Arithmetic Instructions

We'll be working with the MIPS instruction set architecture. A series of RISC Processors

- Similar to other architectures developed since the 1980's.
- Used by NEC, Nintendo, Silicon Graphics, Sony, etc.

Instruction Set Design Goals

- Maximize performance
- Minimize cost
- Reduce design time

Computer Organization

All computers consist of five components

- Processor: (1) datapath and (2) control
- (3) Memory
- (4) Input devices and (5) Output devices

Not all “memory” are created equally

- Cache: expensive
  Fast memory: Placed closer to the processor
- Main memory: less expensive memory
  We can have more.

Input and output (I/O) devices

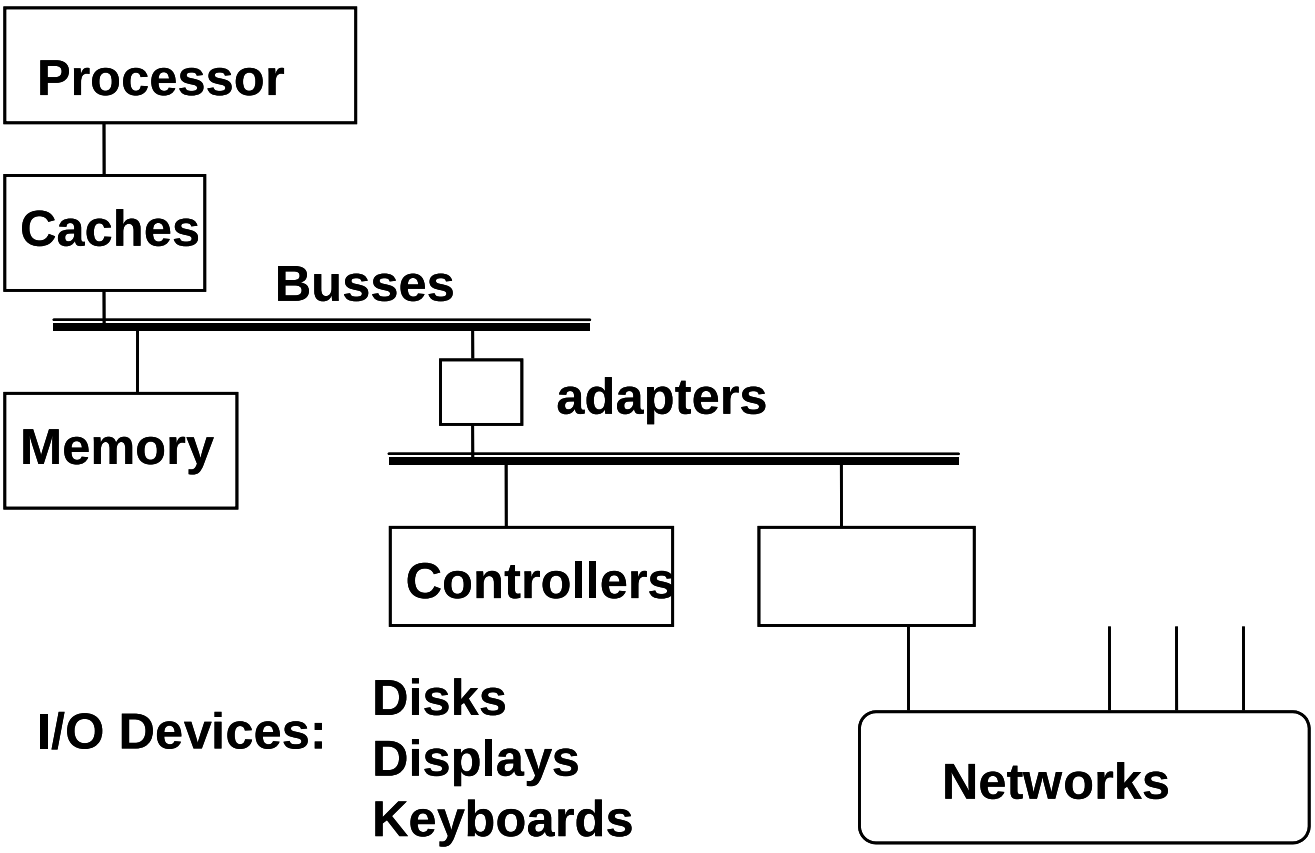
The messiest organization

- Wide range of speed
  Graphics vs. Keyboard/mouse
- Wide range of requirements
  Speed, Standard, Cost ...
- Least amount of research (so far)

Computer Organization

Computer System Components

All the components have interfaces and their own organization.



Computer Word Size

Computers are often described in terms of their word size. General-purpose computers range from 16-bit to 64-bit word sizes.

- In a 32-bit computer data and instructions are stored in memory as 32-bit units.
- Word size indicates the size of the data bus of a computer system.

Two Types of Computer Words

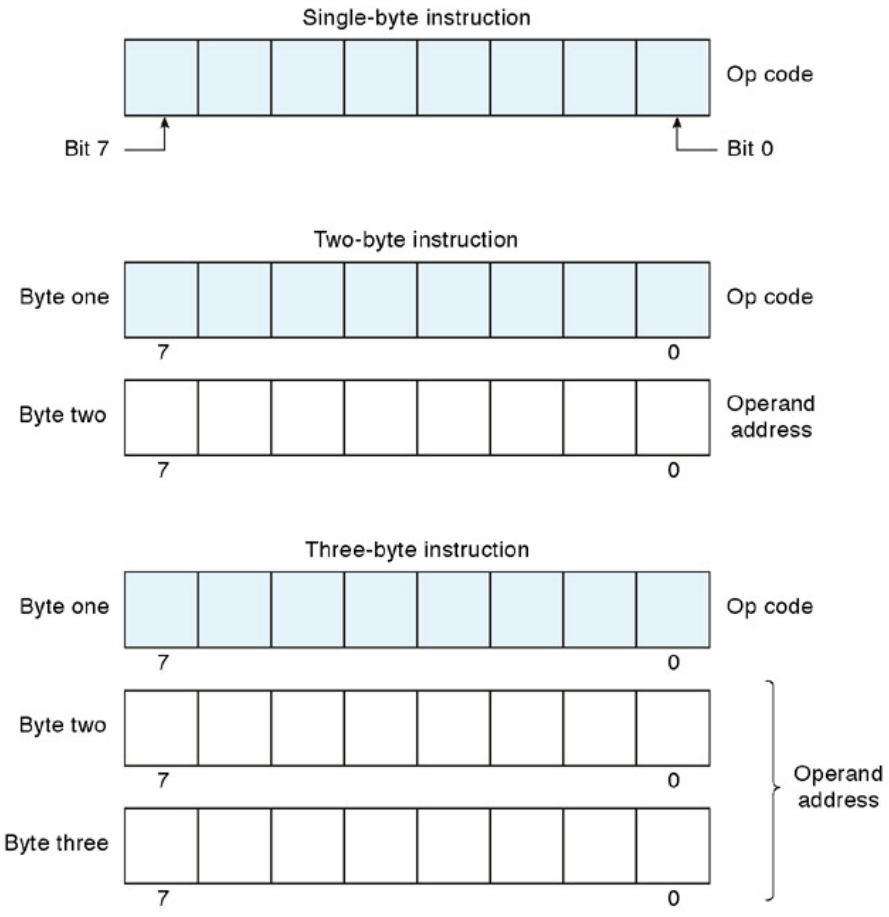
- Instruction word
- Data words
  - BCD
  - Integer
  - Floating Point
  - ASCII

Computer Instructions

Two components of a computer instruction:

- Opcode field
- Address fields (one or more)

Instruction word size = opcode field + address fields
(0, 1 or More)



ADD A, B, C

Instruction Set

Type and number of instructions vary:

Three, two, one and even zero address instructions are common

Instruction Types

Data transfer, Arithmetic, Logical, Program Control, System Control and I/O.

Addressing Modes

Inherent, Immediate, Absolute, Register, Indirect Register, Indexed, Index Base Register, Index Offset, Index relative etc.

Processor architecture has considerable influence on a specific instruction format:

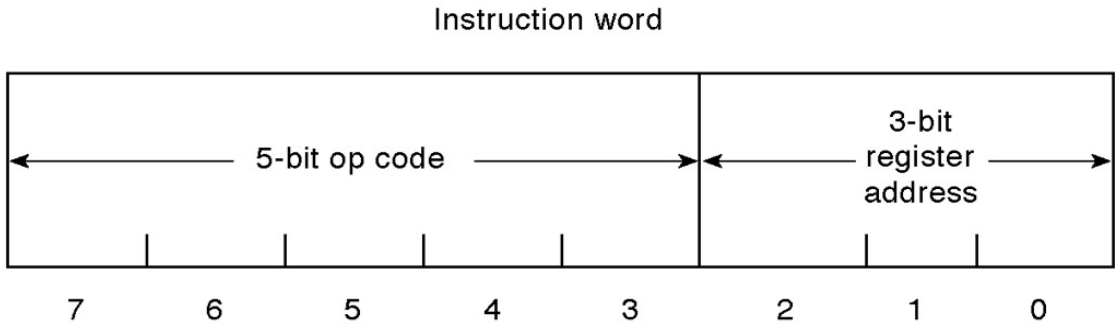
- Opcode field size determines the maximum number of instructions.
- A unique binary pattern of opcode defines each instruction.

Huffman Encoding: Less frequently used instructions have large opcode field than the frequently used instructions.

Instruction Word

A single-byte instruction for an 8051 μ-controller consists of 5-bit opcode and 3-bit register address.

- 5-bit opcode field allows 32 instructions.
- 3-bit register address field can address only 8 registers or memory locations.



The opcode field tells the processor what to do and register field provides the operand.

MOV A, R3 instruction code: **11101 011**

Opcode (11101) provides the move instruction

Register address field (011) indicates R3

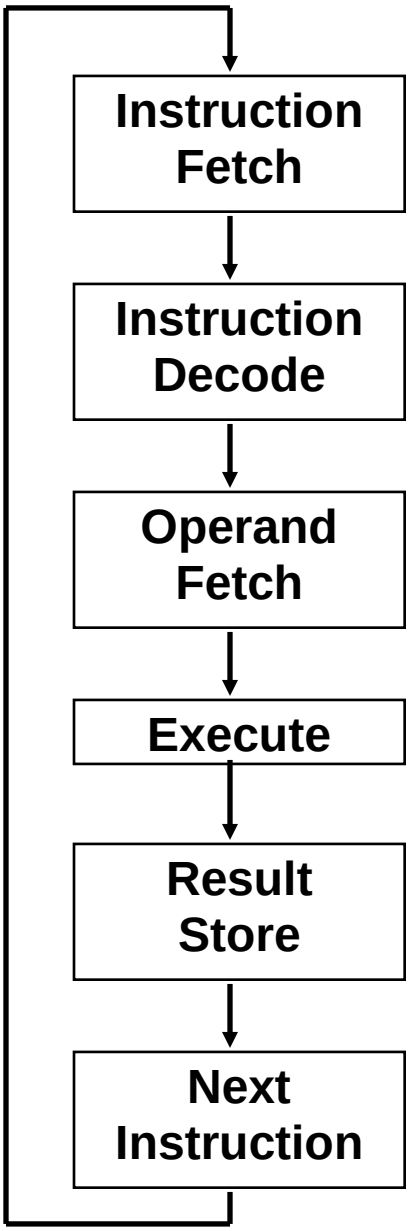
Reg A ← R3: Copy the content of R3 into Reg A

Small opcode/address fields limit the capabilities of a processor.

Multi-byte instructions can be a solution to this problem as indicated earlier.

ISA: Instruction Set Architecture

What must be specified?



- Instruction Format/Encoding
  - How is it decoded?
- Operands & Result Location
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Data Type and Size
- Operations
  - what are supported
- Successor Instruction
  - jumps, conditional and other branches

ISA Classes

Basic ISA (Instruction Set Architecture) Types

Accumulator (one register):

1 address	add A	acc ← acc + mem[A]
(1+x) address	addx A	acc ← acc + mem[A + x]

Stack:

Zero address	add	tos ← tos + next

GPR: General Purpose Register

2 address	add A, B	EA(A) ← EA(A) + EA(B)
3 address	add A, B, C	EA(C) ← EA(B)+EA(A)

EA: Effective Address

Load/Store:

load R1, A	R1 ← mem[A]
load R2, B	R2 ← mem[B]
add R3, R2, R1	R3 ← R2 + R1
store C, R3	mem[C] ← R3

Comparison

- Bytes per instruction?
- Number of Instructions?
- Cycles per instruction? CPI

Different ISA Classes

Consider the implementation of C = A + B
Code sequence for four classes of instruction sets

Stack:

```
Push A;          Push B
Add
Pop  C
```

Accumulator:

```
Load  Acc
Add   B          Acc ← B + Acc
Store C
```

Register: (register-memory)

```
Load  R1, A      R1 ← A
Add   R1, B
Store C, R1
```

Register: (load-store) e.g. MIPS-Processor

```
Load  R1, A ;    Load  R2, B
Add   R3,R1,R2
Store C, R3
```

Instruction Set Classes

General Purpose Registers

Almost all the machines used it during 1975-1995

Advantages of registers

- Registers are faster than memory.
- Easier for a compiler to use.
  (A*B) – (C*D) – (E*F) multiplies in any order
- Registers can hold variables
- Memory traffic is reduced (programs speed up)
  Registers access is faster than memory.
- Code density improves
  Registers are named with fewer bits as compared to memory location (address).

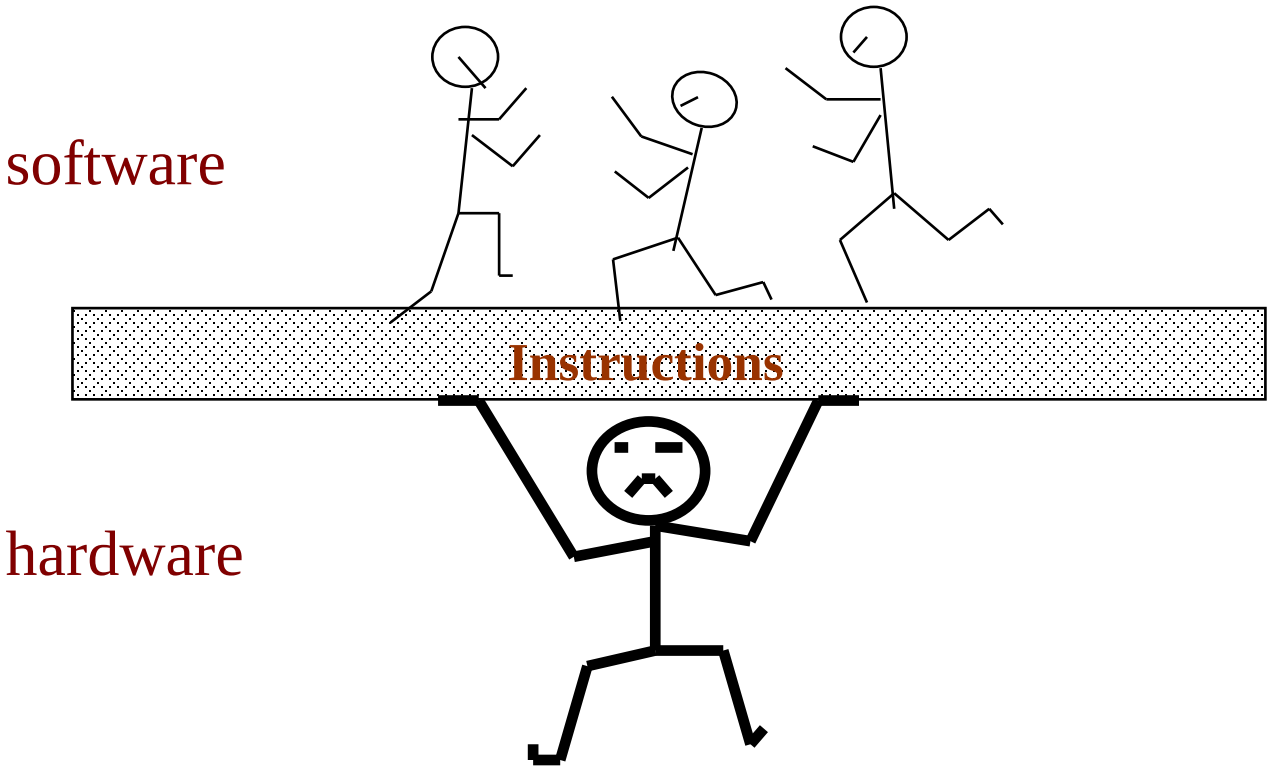
To Summarize

- Expect new instruction set architecture to employ general-purpose register architecture.
- Pipelining ⇒ Expect it to use load/store variant of GPR ISA

Instruction Set Design

Main Goals

Maximize performance, minimize cost and reduce design time



Between Software and Hardware

Which is easier to modify?

Instruction Set Architectures

- Early trend was to add more instructions to new CPUs to do elaborate operations
  VAX architecture had an instruction to multiply polynomials!

- RISC philosophy (Cocke IBM, Patterson, Hennessy, 1980s) –
  Reduced Instruction Set Computing
  - Keep the instruction set small and simple, makes it easier to build fast hardware.
  - Let software do complicated operations by composing simpler ones.

MIPS – A semiconductor company that built one of the first commercial RISC architectures

We will study the MIPS architecture in detail here, in this class

Computer Instructions

- Language of the Machine
  Machine language or binary representation of assembly language.
- More primitive than the higher level languages like C and Java.

  No sophisticated control flow
- Very Restrictive
  e.g., MIPS-Processor Arithmetic Instructions

We'll be working with the MIPS-Processor (Load/Store) instruction set architecture

  – similar to other architectures developed since the 1980's

  Used by NEC, Nintendo, Silicon Graphics, and Sony systems

Design Goals

- Maximize performance (CPI)
- Minimize cost
- Reduce design time

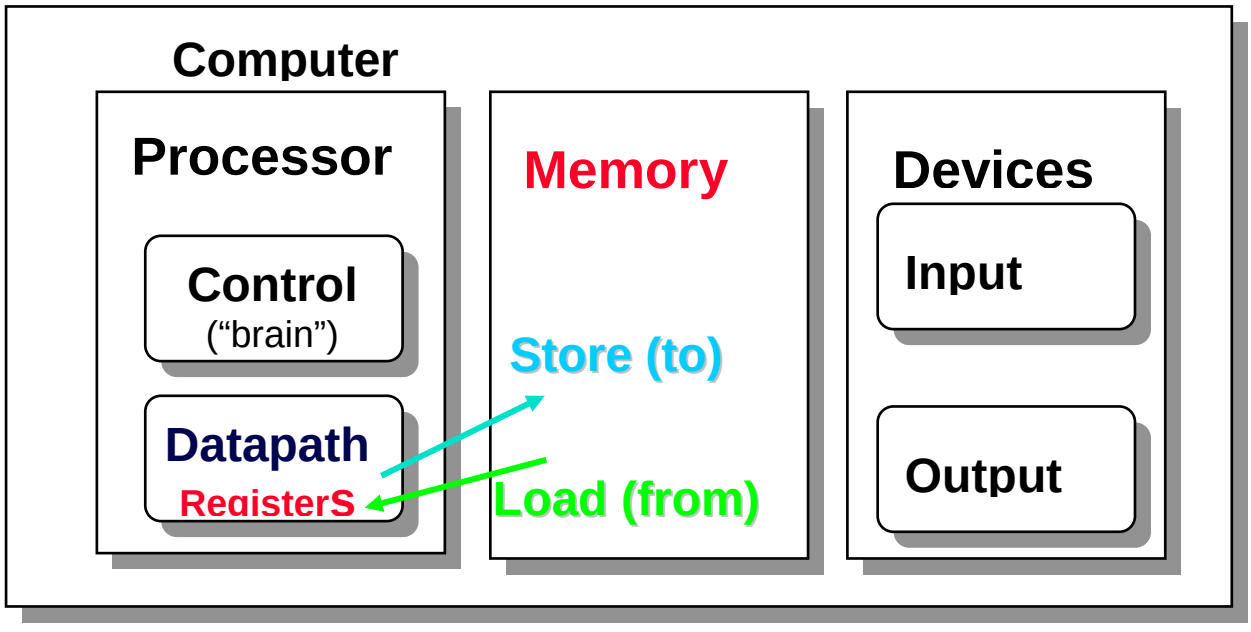
MIPS-Processor Instruction

All instructions have three operands
Operand order is fixed (destination first)

Assembly Operands: Memory

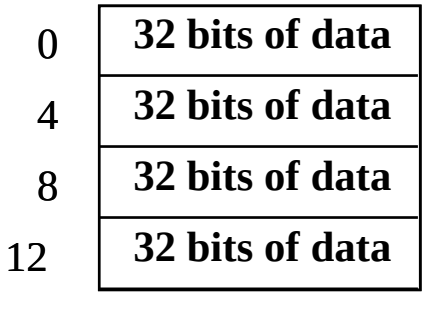
- C variables map onto registers; what about large data structures like arrays?
  Memory contains such data structures
- But MIPS arithmetic instructions only operate on registers, never directly on memory.

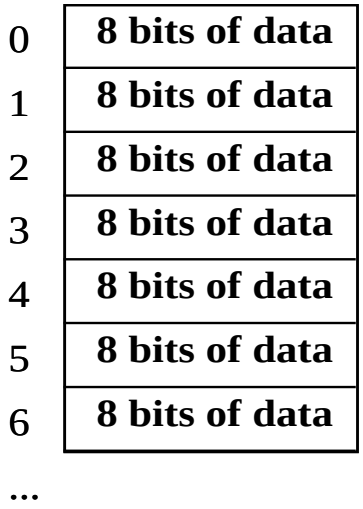
Data transfer instructions Transfer data between registers and memory: Memory to register and Register to memory



Memory Organization

Viewed as a large, one-dimension array
A memory address is an index into the array





"Byte addressing" means that the index points to a byte of memory.
Most data items use larger "words" (16-64 bit size)

For MIPS, a word is of 32 bits or 4 bytes.
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, **Alignment:** Objects must fall on address that is multiple of their size.

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

Memory *vs.* Registers

What if more variables than registers?

Compiler tries to keep most frequently used variable in registers
Less common in memory:

Spilling

Why not keep all variables in memory?

Smaller is faster:

Registers are faster than memory

Registers more versatile:

- MIPS arithmetic instructions can read two regsiters, operate on them, and write one regsiter per instruction
- MIPS data transfer only read or write one operand per instruction, and no operation

MIPS-Processor Arithmetic

All instructions have three operands
Operand order is fixed (destination first)
Example: C code: A = B + C
MIPS code: add $s0, $s1, $s2
(associated with variables by compiler)

Design Principle: simplicity favors regularity.

Of course this complicates few things...
e.g. C code: A = B + C + D;
E = F - A;
MIPS-Processor code: $s0 <= A, $s1 <= B
add $t0, $s1, $s2 ; $s2 <= C, $s3 <= D
add $s0, $t0, $s3 ; $t0 <= B + C
sub $s4, $s5, $s0 ;

Design Principle: smaller is faster.
Why?
Arithmetic instruction operands must be in registers.
only 32 registers provided

Compiler associates variables with registers.
What about programs with lots of variables?

MIPS-Processor Instructions

Load and store instructions
Example: C code: A[8] = h + A[8];

If h is associated with register $s2 and
base address for A[i] (i.e. A[0]) is in register $s3

MIPS code:

```
lw      $t0, 32($s3)     A[8]
add     $t0, $s2, $t0    h + A[8]
sw      $t0, 32($s3)
```

Arithmetic operands are registers, not memory!

Can we figure out the code?

```
swap(int v[], int k);
{ int temp;
      temp = v[k]
      v[k] = v[k+1];
      v[k+1] = temp;
}
```

⇒

```
swap:   add $t1, $a1, $a1    2 x k
        add $t1, $t1, $t1    4 x k
        add $t1, $a0, $t1 a0+4xk
        lw $t0, 0($t1) load v[k]
        lw $t2, 4($t1) load v[k+1]
        sw $t2, 0($t1)
        sw $t0, 4($t1)
        jr $ra
```

v[] and k are in $a0 and $a1

Pointers *vs.* Values

Key Concept: A register can hold any 32-bit value. (signed) int, an unsigned int, a pointer (memory address), etc.

If we write **add** $t2,$t1,$t0

then $t0 and $t1 better contain values

If we write **lw** $t2,0($t0)

then $t0 better contain a pointer

Compilation with Memory

What offset in lw to select A[5] in C Language?

4 x 5 = 20 to select A[5], byte *vs* word

Compile by hand using registers:

g = h + A[5];

g: $s1, h: $s2, $s3 is the base address of A

First transfer from memory to register:

lw $t0,20($s3) *# $t0 gets A[5]*

Add 20 to $s3 to select A[5], put into $t0

Next add it to h and place in g

add $s1,$s2,$t0 *# $s1 = h+A[5]*

MIPS-Processor Instructions

Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

addu i.e. add two unsigned numbers

Instruction Format: R-format

000000	10001	10010	01000	00000	100001
op	rs	rt	rd	shamt	funct
0	17	18	8	0	33
	$s1	$s2	$t0		

addu $t0, $s1, $s2
$t0 is the destination

So far we've learned:

MIPS-Processor

loading words but addressing bytes

arithmetic on registers only

Instruction	Meaning
add $s1, $s2, $s3	$s1 <= $s2 + $s3
sub $s1, $s2, $s3	$s1 <= $s2 – $s3
lw $s1, 100($s2)	$s1 <= Memory[$s2+100]
sw $s1, 100($s2)	Memory[$s2+100] => $s1

Instructions, like registers and words of data, are also 32 bits long

- Example: add $t1, $s1, $s2
- A register's address is a 5-bit number
  $t1=9, $s1=17, $s2=18

Instruction Format: R-type

000000	10001	10010	01001	00000	100000
op	rs	rt	rd	shamt	funct
	$s1	$s2	$t1		

Can you guess what the field names stand for?

Machine Language

Consider the load-word and store-word instructions
What would the regularity principle have us do?
New principle:

Good design demands a compromise

Introduce a new type of instruction format
I-type for data transfer instructions.
The other format was R-type for register.

I-Format: For load/store instructions

op	rs	rt	address (16-bit)

Example: lw $t0, 32($s3)

35	19	8	32

Where's the compromise?

A different kind of instruction formats for different types of instructions.

Stored Program

Instructions are bits
Programs are stored in memory (I-Format and R-Format) to be read or written just like data.

Fetch & Execute Cycle

- Instructions are fetched and put into a special register (Instruction Register).
- Instruction register bits "control" the subsequent actions.
- Fetch the “next” instruction and continue

Control: Decision making instructions

- alter the control flow, J-Format
  change the "next" instruction to be executed.

MIPS-Processor conditional branch instructions:

```
bne $t0, $t1, Label    (If $t0 is not = $t1)
beq $t0, $t1, Label
```

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

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

MIPS Decision Instructions

Decision instruction in MIPS:

beq register1, register2, L1 beq is "Branch if (registers are) equal"

Same meaning as (using C):

if (register1==register2) goto L1

Complementary MIPS decision Instruction:

bne register1, register2, L1

bne is "Branch if (registers are) not equal"

Same meaning as (using C):

if (register1!=register2) goto L1

Called conditional branches

In addition to conditional branches, MIPS has an unconditional branch:

j label Called a Jump Instruction:

Same meaning as (using C): goto label

Technically, same as: beq $0, $0, label

Branch Instructions

MIPS-Processor unconditional branch instructions:

j label

Example: $s4 <= i, $s5 <= j, $s3 <= h

```
    if (i ! = j)          beq $s4, $s5, Label1
       h = i + j;         add $s3, $s4, $s5
    else                  j Label2
       h = i - j;     Label1: sub $s3, $s4, $s5
                      Label2: ...
```

Can you build a simple instruction for a loop?

So far:

Instruction	Meaning
add $s1, $s2, $s3	$s1 = $s2 + $s3 *R-type*
sub $s1, $s2, $s3	$s1 = $s2 – $s3
lw $s1,100($s2)	$s1 = Mem[$s2+100] *I-type*
sw $s1,100($s2)	Memory[$s2+100] = $s1
bne $s4, $s5, L-1	Next instr. at L-1 if $s4 ≠ $s5
beq $s4, $s5, L-2	Next instr. at L-2 if $s4 = $s5
j Label	Next instr. is at the Label *j-type*

Control Flow

We have seen **beq** and **bne**
what about Branch-if-less-than?

New instruction: *if $s1 < $s2 then $t0 = 1*
slt $t0, $s1, $s2 *else $t0 = 0*

Can use this instruction to build
"blt $s1, $s2, Label" *If s1 < s2 goto Label*

One can now build general control structures.

Name	Number	Usage	Preserved across a call?
$zero	0	the value 0	n/a
$v0-$v1	2-3	return values	no
$a0-$a3	4-7	arguments	no
$t0-$t7	8-15	temporaries	no
$s0-$s7	16-23	saved	yes
$t18-$t19	24-25	temporaries	no
$sp	29	stack pointer	yes
$ra	31	return address	yes

"caller saved"
"callee saved"

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**		

Could specify a register (like lw and sw) and add it to address.

- use Instruction Address Register
  (i.e. PC: program counter)
- most branches are local
  (principle of locality)

Jump instructions just use high order bits of PC address boundaries of 256 MB

MIPS Instruction Formats

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rd	immediate		
J	opcode	target address				

rs (Source Register): *generally* used to specify register containing first operand
rt (Target Register): *generally* used to specify register containing second operand (note that name is misleading and some cases as dest. reg. address)
rd (Destination Register): *generally* used to specify register which will receive result of computation

R-format: used for all other instructions
It will soon become clear why the instructions have been partitioned in this way.

Define "fields" of the following number of bits each: 6 + 5 + 5 + 5 + 5 + 6 = 32

6	5	5	5	5	6

Instruction R-Format

MIPS Instruction: **add** $8, $9, $10

Decimal number per field representation:

0	9	10	8	0	32

Binary number per field representation:

000000	01001	01010	01000	00000	100000

hex

Hex representation: $012A\ 4020_{hex}$

Decimal representation: $19{,}546{,}144_{ten}$

Which instruction has same representation as 35?

A. add $0, $0, $0
B. subu $s0,$s0,$s0
C. lw $0, 0($0)
D. addi $0, $0, 35
E. subu $0, $0, $0

Instruction Format

Which instruction has same representation as 35?

add $0, $0, $0	opcode	rs	rt	rd	shamt	funct
subu $s0,$s0,$s0	opcode	rs	rt	rd	shamt	funct
lw $0, 0($0)	opcode	rs	rd	offset		
addi $0, $0, 35	opcode	rs	rd	immediate		
subu $0, $0, $0	opcode	rs	rt	rd	shamt	funct

0: $0, 8: $t0, 9:$t1, ..15: $t7, 16: $s0, 17: $s1, .. 23: $s7

add: opcode = 0, funct = 32

subu: opcode = 0, funct = 35

addi: opcode = 8

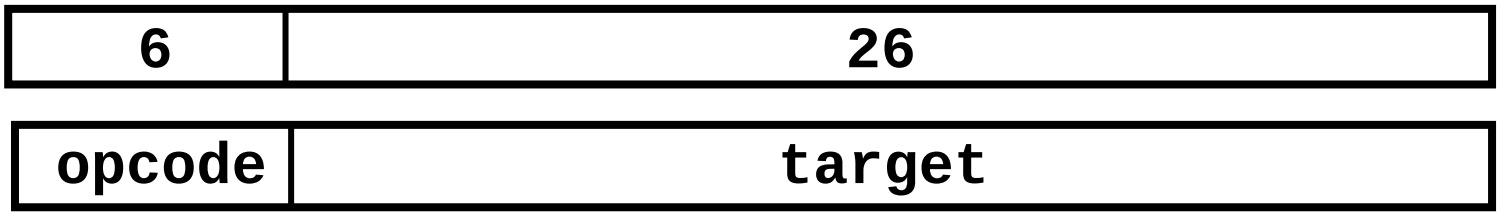
lw: opcode = 35

add $0, $0, $0	0	0	0	0	0	32
subu $s0,$s0,$s0	0	16	16	16	0	35
lw $0, 0($0)	35	0	0	0		
addi $0, $0, 35	8	0	0	35		
subu $0, $0, $0	0	0	0	0	0	35

Instruction J-Format

J-format: used for **j** and **jal**

Define "fields" of the following number of bits each:



New PC = { PC[31..28], target address, 00 }

Understand where each part came from!

{ 4 bits , 26 bits , 2 bits } = 32 bit address

{ 1010, 11111111111111111111111111, 00 }

= 10101111111111111111111111111100

Procedure Call: Jump & link (**jal** proc_lab)

- Put the return addr (next inst address) into link register, ra ($31)
- jump to next instruction

Procedure Return: Jump register (**jr** $ra)

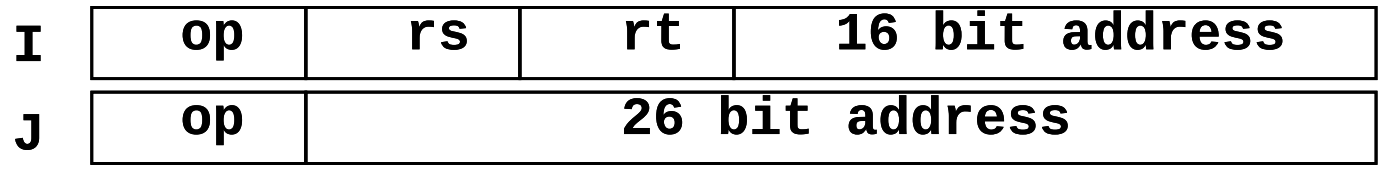
- Copies $ra to program counter
- Can also be used for computed jumps
  e.g. for case/switch statements

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:



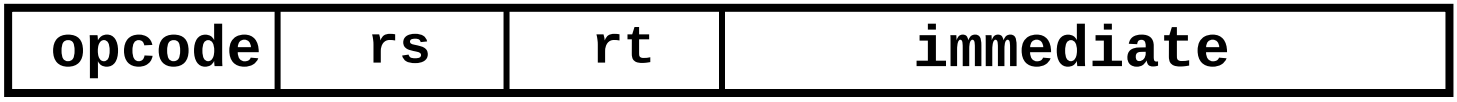
Could specify a register (like lw and sw) and add it to address.

- use Instruction Address Register
  (i.e. PC: program counter)
- most branches are local
  (principle of locality)

Jump instructions just use high order bits of PC address boundaries of 256 MB

Instruction I-Format

I-format: used for instructions with immediates, **lw** and **sw** (since the offset counts as an immediate), and the branches (**beq** and **bne**)



Chances are that **addi**, **lw**, **sw** and **slti** will use immediates small enough to fit in the immediate field. …but what if it's too big?
We need a way to deal with a 32-bit immediate in any I-format instruction.

Solution:
Handle it in software + new instruction
Don't change the current instructions: instead, add a new instruction to help out

lui register, immediate.

stands for **L**oad **U**pper **I**mmediate

- Takes 16-bit immediate and puts these bits in the upper half (high order half) of the specified register
- sets lower half to 0s

Constants

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

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

Solutions? Why not?

- put 'typical constants' in memory and load them.
- 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
```

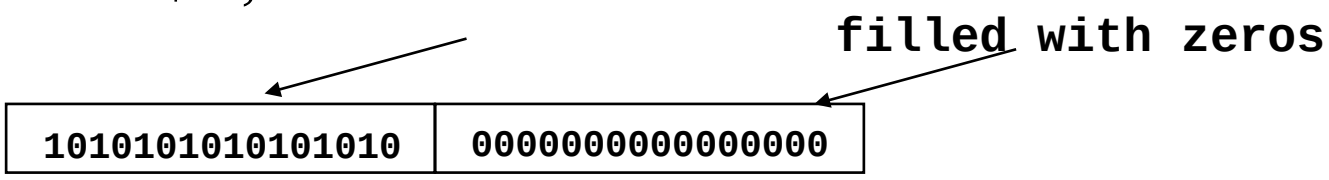
How to make this work?

How about larger constants?

Large Constants

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

lui $t0, 1010101010101010



Then must get the lower order bits right, i.e.

1010101010101010	0000000000000000
0000000000000000	1010101010101010

1010101010101010	1010101010101010

ori $t0, $t0, 1010101010101010

Assembly Language *vs.* Machine Language

Assembly provides a convenient symbolic representation.

- much easier than writing down numbers
- e.g. destination first

Machine language is the underlying reality

- e.g. destination is no longer first

Assembly can provide 'pseudo-instructions'

- e.g., “move $t0, $t1” exists only in Assembly
- would be implemented using “add $t0,$t1,$zero”

When considering performance you should count the real instructions.

Switch Statement

Choose among four alternatives depending on whether ***k*** has the value 0, 1, 2 or 3. Compile the following C code manually:

```
switch (k) {
    case 0: f = i+j; break;
    case 1: f = g+h; break;
    case 2: f = g–h; break;
    case 3: f = i–j; break;
}
```

Rewrite it as a chain of if-else statements, which we already know how to compile:

```
if (k = = 0) f = i+j;
    else if (k = = 1) f = g+h;
        else if (k = = 2) f = g–h;
            else if (k = = 3) f = i–j;
```

Using the following mapping:

f:$s0, g:$s1, h:$s2,
i:$s3, j:$s4, k:$s5

Switch Statement

Final compiled MIPS code:

```
    bne $s5, $0, L1
    add  $s0, $s3, $s4
    j    Exit

L1: addi $t0, $s5, -1
    bne  $t0, $0, L2
    add  $s0, $s1, $s2
    j    Exit

L2: addi $t0, $s5, -2
    bne  $t0, $0, L3
    sub  $s0, $s1, $s2
    j    Exit

L3: addi $t0, $s5, -3
    bne  $t0, $0, Exit
    sub  $s0, $s3, $s4
Exit:
```

Function/Procedure Call

Registers play a major role in keeping track of information for function calls.

MIPS CPU Registers

The constant 0	$0	$zero
Reserved for Assembler	$1	$at
Return Values	$2-$3	$v0-$v1
Arguments	$4-$7	$a0-$a3
Temporary	$8-$15	$t0-$t7
Saved	$16-$23	$s0-$s7
More Temporary	$24-$25	$t8-$t9
Used by Kernel	$26-27	$k0-$k1
Global Pointer	$28	$gp
Stack Pointer	$29	$sp
Frame Pointer	$30	$fp
Return Address	$31	$ra

```
... sum( a, b ); ... // a, b: $s0,$s1
          int sum( int x, int y) {
            return x + y;
          }

1000 add  $a0,$s0,$zero
1004 add  $a1,$s1,$zero
1008 addi $ra,$zero,1016
1012 j    sum
1016 …

2000 sum: add $v0,$a0,$a1
2004 jr   $ra
```

Another Procedure Call

```
void swap(int v[], int k)
{
 int temp;
 temp = v[k];
 v[k] = v[k+1];
 v[k+1] = temp;
}
```

Procedure body

```
swap:  sll    $t1, $a1, 2          # reg $t1 = k * 4
       add    $t1, $a0, $t1        # reg $t1 = v + (k * 4)
                                   # reg $t1 has the address of v[k]
       lw     $t0, 0($t1)          # reg $t0 (temp) = v[k]
       lw     $t2, 4($t1)          # reg $t2 = v[k + 1]
                                   # refers to next element of v
       sw     $t2, 0($t1)          # v[k] = reg $t2
       sw     $t0, 4($t1)          # v[k+1] = reg $t0 (temp)
```

Procedure return

```
       jr     $ra                  # return to calling routine
```

Function/Procedure Call

Why use **jr** here? Why not simply use **j**?

Procedure return: jump register

jr $ra

Copies $ra to program counter

Can also be used for computed jumps

e.g. for case/switch statements

Procedure call: jump and link

jal Procedure-Label

Address of following instruction put in $ra

Jumps to target address

Single instruction to jump and save return address: jump and link (jal)

With **jr** $ra as part of sum

```
1008 addi $ra,$zero,1016    #$ra=1016
1012 j sum                  #goto sum with jal
```

```
1008 jal sum  # $ra=1012,goto sum
```

Why have a jal?

Function/Procedure Call Support

Syntax for jal (jump link) is same as for j (jump): jal label

jal should really be called "link and jump":

Step 1 (link): Save address of *next* instruction into $ra
(Why next instruction? Why not current one?)
Step 2 (jump): Jump to the given label

Nested Procedures call other procedures
For nested call, caller needs to save on the stack:
Its return address
Arguments and temporaries needed after the call
Restore from the stack after the call

C code:

```
int fact (int n)
{
  if (n < 1) return f;
  else return n * fact(n - 1);
}
```

Argument ***n*** in $a0
Result in $v0

Nested Procedure Call Support

MIPS code:

```
fact:
  addi $sp, $sp, -8    :adjust stack for 2 items
  sw   $ra, 4($sp)          :save return address
  sw   $a0, 0($sp)     : save argument
  slti  $t0, $a0, 1    : test for n < 1
  beq $t0, $zero, L1
  addi $v0, $zero, 1  :yes, result is 1
  addi $sp, $sp, 8     : pop 2 items from
  jr   $ra             : stack and return

L1:
  addi $a0, $a0, -1    : else decrement n
  jal  fact            : recursive call
  lw   $a0, 0($sp)     :restore original n
  lw   $ra, 4($sp)     :and return address
  addi $sp, $sp, 8     : pop 2 items from stack
  mul  $v0, $a0, $v0   : multiply to get
  jr   $ra             :result and return
```

Other Issues

Things we are not going to cover

Support for procedures, linkers, loaders, memory layout, stacks, frames, recursion, manipulating strings and pointers, interrupts and exceptions. system calls and conventions.

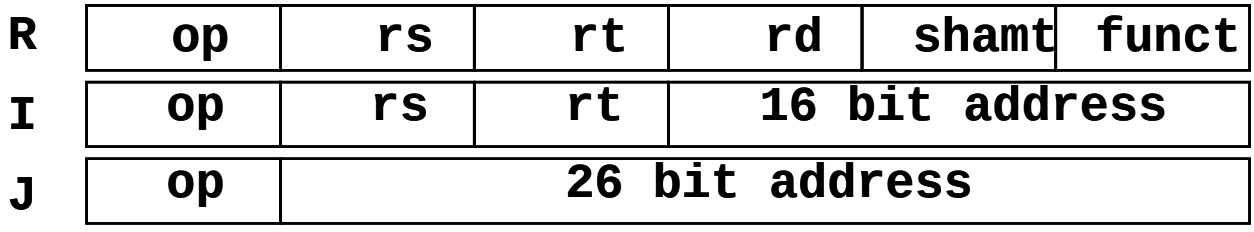
Some of these we'll talk about later

We've focused on architectural issues

- basics of MIPS assembly & machine language
- We'll build a processor to execute some of these instructions.

Overview of MIPS-Processor ISA

- Simple instructions all 32 bits wide.
- Very structured, no unnecessary baggage.
- only three instruction formats.



- rely on compiler to achieve performance
  what are the compiler's goals?
- help compiler where we can.

MIPS Arithmetic Instructions

Instruction	*Example*	*Meaning*	*Comments*
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multiply unsigned	multu$2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient & remainder
Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

jump, branch, compare instructions

Instruction	***Example***	***Meaning***
branch on equal	beq $1,$2,100 *Equal test; PC relative branch*	if ($1 == $2) go to PC+4+100
branch on not eq.	bne $1,$2,100 *Not equal test; PC relative*	if ($1!= $2) go to PC+4+100
set on less than	slt $1,$2,$3 *Compare less than; 2's comp.*	if ($2 < $3) $1=1; else $1=0
set less than imm.	slti $1,$2,100 *Compare < constant; 2's comp.*	if ($2 < 100) $1=1; else $1=0
set less than uns.	sltu $1,$2,$3 *Compare less than; natural numbers*	if ($2 < $3) $1=1; else $1=0
set l. t. imm. uns.	sltiu $1,$2,100 *Compare < constant; natural numbers*	if ($2 < 100) $1=1; else $1=0
jump	j 10000 *Jump to target address*	go to 10000
jump register	jr $31 *For switch, procedure return*	go to $31
jump and link	jal 10000 *For procedure call*	$31 = PC + 4; go to 10000

To summarize:

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Alternative Architectures

Design alternative:

- provide more powerful operations
- goal is to reduce number of instructions to be executed
- danger is a slower cycle time and/or a higher CPI.

Sometimes referred to as “RISC vs. CISC” architecture

- virtually all-new instruction sets since 1982 have been RISC.
- VAX: minimize code size, make assembly language easy.

instructions from 1 to 54 bytes long!

We’ll look at PowerPC

PowerPC

Indexed addressing

- example: lw $t1, $a0+$s3
  $t1 <= Memory[$a0+$s3]
- What do we have to do in MIPS?

Update addressing

- update a register as part of load
  (for marching through arrays)

- example: lwu $t0,4($s3)
  $t0 <= Memory[$s3+4]
  $s3=$s3+4

- What do we have to do in MIPS?

Others:

- load multiple/store multiple words
- a special counter register "bc Loop"

- *decrement counter, if not 0 goto loop*

ARM & MIPS Similarities

ARM: the most popular embedded CPU core.
ARM and MIPS CPUs have a similar basic set of instructions.

	ARM	**MIPS**
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Mem-mapped	Mem-mapped

Compare and Branch in ARM
Uses condition codes for result of an arithmetic or logical instruction

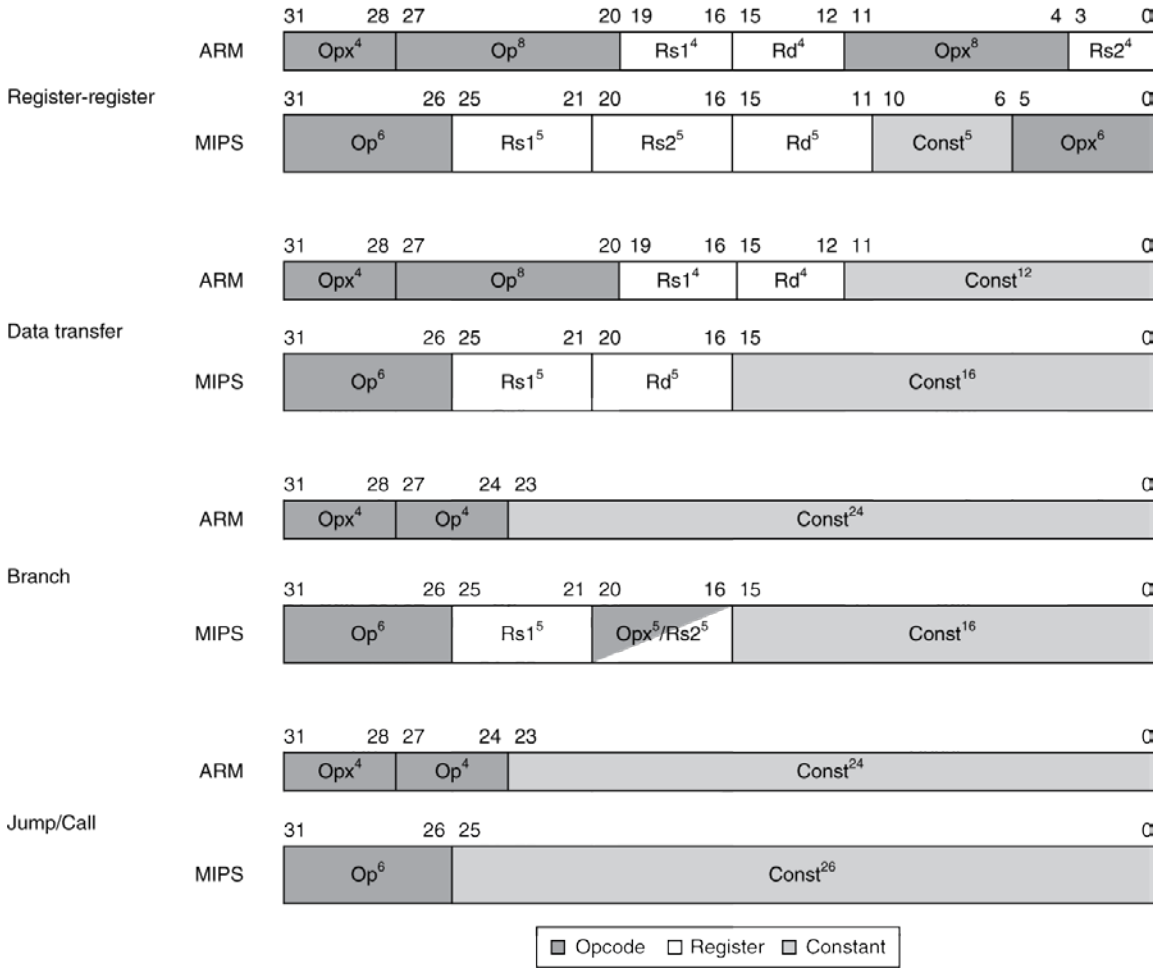
Compare instructions to set condition codes without keeping the result

Each instruction can be conditional

- Top 4 bits of instruction word: condition value
  Can avoid branches over single instructions

Compare and Branch in ARM

- Uses condition codes for result of an arithmetic or logical instruction.
  Negative, zero, carry & overflow
- Compare instructions to set condition codes without keeping the result.
- Each instruction can be conditional.
- Top 4 bits of instruction word has the condition value. It can avoid branches over single instr.



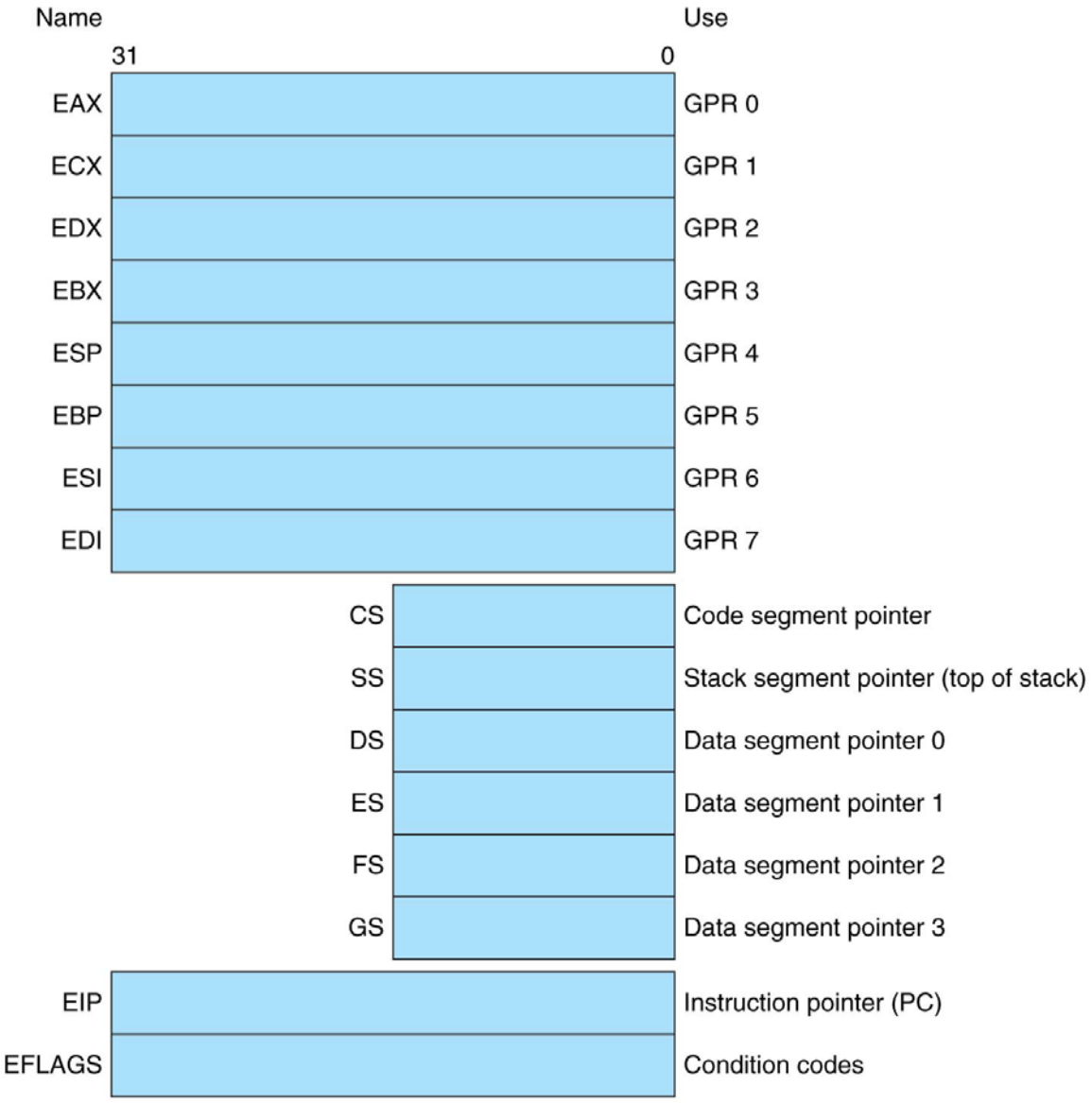
The Intel x86 ISA

Evolution and backward compatibility

- 8080 (1974): 8-bit microprocessor
  - Accumulator, plus 3 index-register pairs
- 8086 (1978): 16-bit extension to 8080
  - Complex instruction set (CISC)
- 8087 (1980): floating-point coprocessor
  - Adds FP instructions and register stack
- 80286 (1982): 24-bit addresses, MMU
  - Segmented memory mapping and protection
- 80386 (1985): 32-bit extension (IA-32)
  - Additional addressing modes and operations
  - Paged memory mapping as well as segments
- i486 (1989): pipelined, on-chip caches/FPU
  - Compatible competitors: AMD, Cyrix, …
- Pentium (1993): super-scalar, 64-bit datapath
  - Later versions added MMX instructions
  - The infamous FDIV bug
- Pentium Pro (1995), Pentium II (1997)
  - New micro-architecture
- Pentium III (1999)
  - Added SSE (Streaming SIMD Extensions) and associated registers
- Pentium 4 (2001)
  - New micro-architecture & added SSE2 instructions
- i3, i5, i7 (2008)

Basic x86 Registers

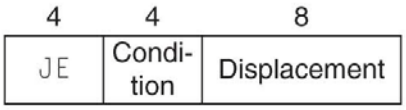
80386 Register Set



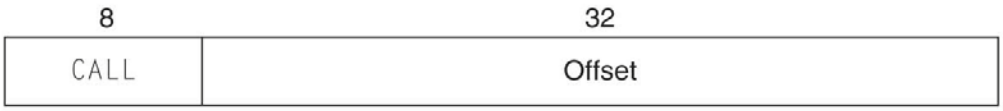
Source/destination operand type	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

x86 Instruction Formats

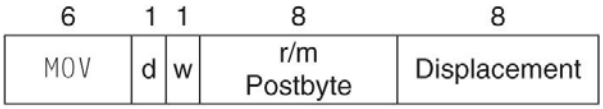
a. JE EIP + displacement



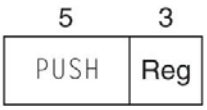
b. CALL



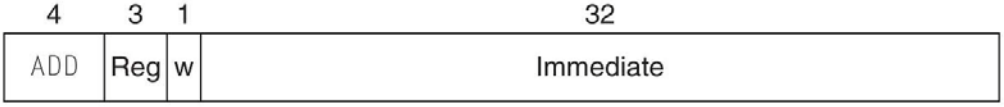
c. MOV EBX, [EDI + 45]



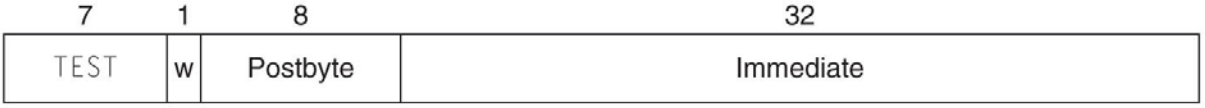
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42

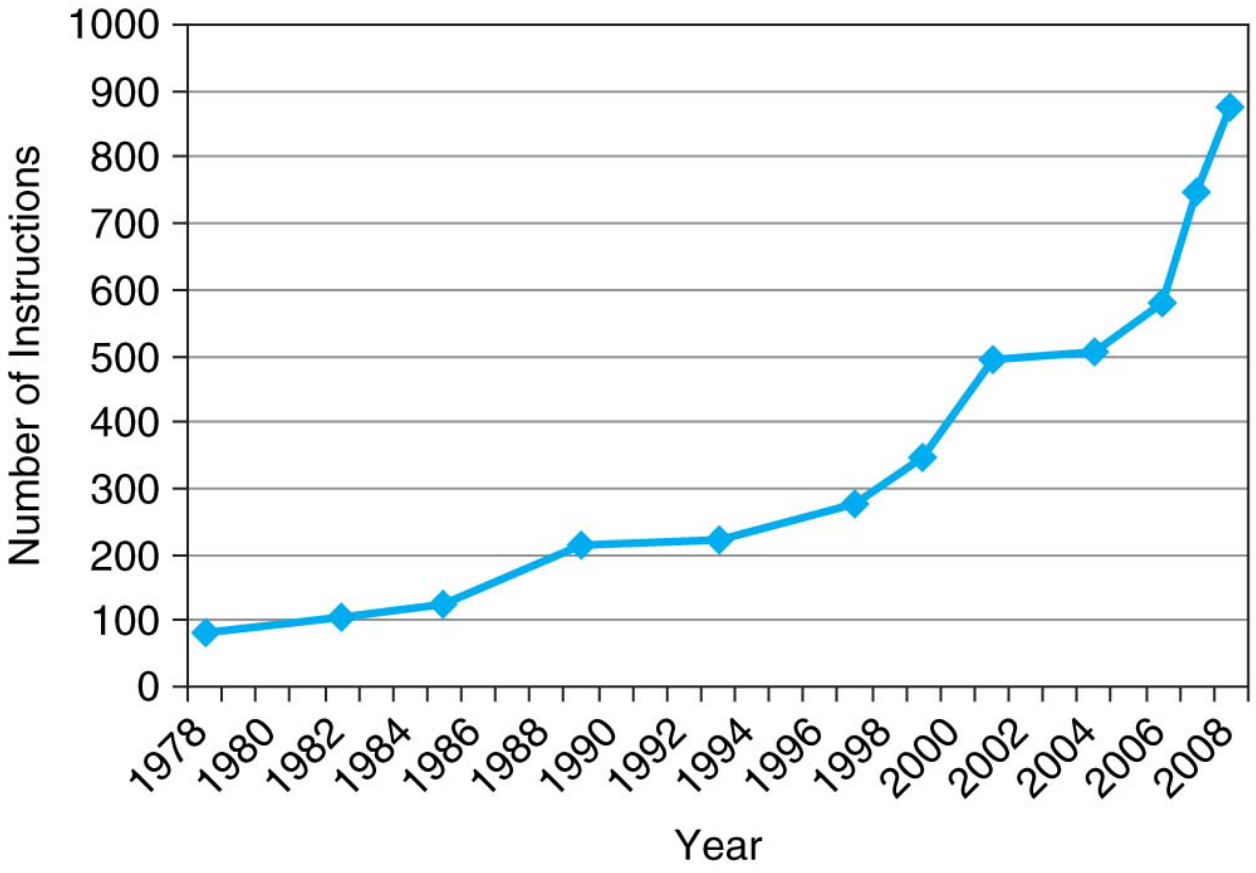


Variable length encoding

- Postfix bytes specify addressing mode
- Prefix bytes modify operation
  - Operand length, repetition, locking, …

Implementing IA-32

- Complex instruction set makes implementation difficult
  - ♦ Hardware translates instructions to simpler micro-operations
    - ■ Simple instructions: 1–1
    - ■ Complex instructions: 1–many
  - ♦ Micro-engine similar to RISC
  - ♦ Market share makes it economically viable
- Comparable performance to RISC
  - ♦ Compilers avoid complex instructions



Typical Instructions of IA-32

Instruction	Meaning
Control	**Conditional and unconditional branches**
`jnz, jz`	Jump if condition to EIP + 8-bit offset; `JNE` (for `JNZ`), `JE` (for `JZ`) are alternative names
`jmp`	Unconditional jump—8-bit or 16-bit offset
`call`	Subroutine call—16-bit offset; return address pushed onto stack
`ret`	Pops return address from stack and jumps to it
`loop`	Loop branch—decrement ECX; jump to EIP + 8-bit displacement if ECX ≠ 0
Data transfer	**Move data between registers or between register and memory**
`move`	Move between two registers or between register and memory
`push, pop`	Push source operand on stack; pop operand from stack top to a register
`les`	Load ES and one of the GPRs from memory
Arithmetic, logical	**Arithmetic and logical operations using the data registers and memory**
`add, sub`	Add source to destination; subtract source from destination; register-memory format
`cmp`	Compare source and destination; register-memory format
`shl, shr, rcr`	Shift left; shift logical right; rotate right with carry condition code as fill
`cbw`	Convert byte in eight rightmost bits of EAX to 16-bit word in right of EAX
`test`	Logical AND of source and destination sets condition codes
`inc, dec`	Increment destination, decrement destination
`or, xor`	Logical OR; exclusive OR; register-memory format
String	**Move between string operands; length given by a repeat prefix**
`movs`	Copies from string source to destination by incrementing ESI and EDI; may be repeated
`lods`	Loads a byte, word, or doubleword of a string into the EAX register

Instruction	Function
`je` name	`if equal(condition code){EIP=name};` `EIP-128 <= name < EIP+128`
`jmp` name	`EIP=name`
`call` name	`SP=SP-4; M[SP]=EIP+5; EIP=name;`
`movw EBX,[EDI+45]`	`EBX=M[EDI+45]`
`push ESI`	`SP=SP-4; M[SP]=ESI`
`pop EDI`	`EDI=M[SP]; SP=SP+4`
`add EAX,#6765`	`EAX= EAX+6765`
`test EDX,#42`	Set condition code (flags) with EDX and 42
`movsl`	`M[EDI]=M[ESI];` `EDI=EDI+4; ESI=ESI+4`

Fallacies

- Powerful instruction $\Rightarrow$ Higher Performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions

- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code $\Rightarrow$ more errors and less productivity

Summary

Instruction complexity is only one variable.

Lower instruction count *vs.* higher CPI and lower clock rate

Design Principles:

- simplicity favors regularity
- smaller is faster
- good design demands compromise
- make the common case fast

Instruction set architecture

– A very important abstraction indeed!