Chapter 3

3.1 Computer (machine) Language:

- **Language of the Machine is called instructions**
  - **Its vocabulary is called an instruction set**
- **More primitive than human languages**
  - **no sophisticated control flow**
  - **Very restrictive: e.g., MIPS Arithmetic Instructions**
  - **machine languages are quite similar: there are mainly three types of language**
    - High level language: C, Pascal, Fortran etc., portable
    - Low level language: assembly language, hardware oriented
    - Artificial Language: Prolog, Lisp
- ***Design goals of computer language : maximize performance and minimize cost, reduce program design time***
- **We'll be working with the MIPS instruction set architecture**
  - **similar to other architectures developed since the 1980's**
  - **used by NEC, Nintendo, Silicon Graphics, Sony**

3.2 MIPS Arithmetic Instruction

- Each instruction performs only one operation
  - It may not be true for other language
- All instructions have 3 operands
- Operand order is fixed (destination first)
  - Example:

```
C code:      a = b + c;
MIPS code:   add a,b,c
```

- Design Principle: simplicity favors regularity. Why?
- Of course this complicates some things…
  - C code:

```
a = b + c + d;
e = f - a;
```

MIPS code:

```
add a, b, c
add a, a, d
sub e, f, a
```

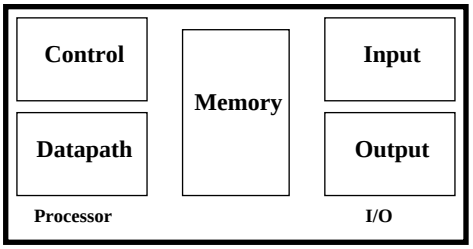
- Do the examples in p108-109

3.3 Operands of MIPS

- Operands must be registers and can not be any variable in the memory
  - only 32 registers word are provided in MIPS and each register word is 32 bits
  - Register is the D register that you learn in “數位系統導論”
- Design Principle: smaller is faster. Why?
  - More registers are convenient for programmer but more complicated for hardware designer (Time and complexity)
- $s1, $s2 for registers that correspond to variables in C and $t0, $t1 for temporary registers needed to compile
  - Do example in p.110
- What about programs with lots of variables ?

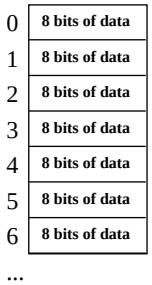
Registers vs. Memory

- Arithmetic instructions operands must be registers,
  — only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables



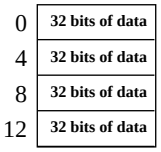
Memory Organization

- 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.



Memory Organization

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



Registers hold 32 bits of data

- 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
  i.e., what are the least 2 significant bits of a word address?

Memory Organization

- MIPS includes ***data transfer*** instructions that transfer data between memory and registers
- Load and store instructions
  - Example:
    C code: `A[8] = h + A[8];`
    MIPS code:
    ```
    lw $t0, 32($s3)
    add $t0, $s2, $t0
    sw $t0, 32($s3)
    ```
    Note: $s3 store the address of A[0]
- Store word has destination last
- Remember arithmetic operands are registers, not memory!
  - Do example in p.112 and 113

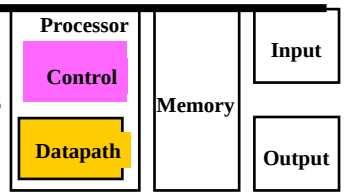
5.1 Introduction

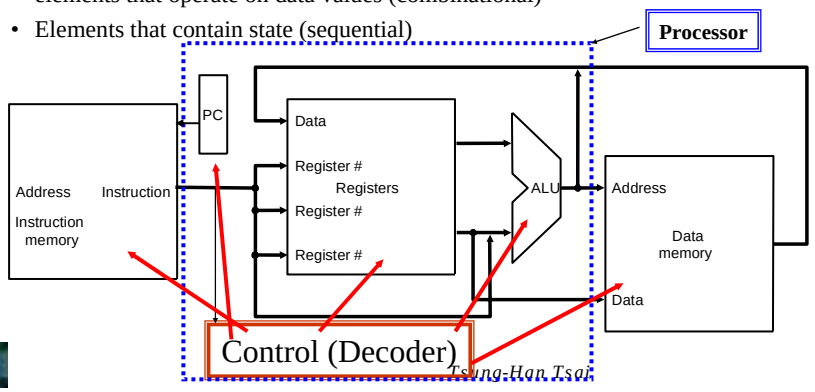
– **The Five Classic Components of a Computer**

– **An abstract view of major functions of MIPS is shown**

– **in Fig.5.1. Two types of functional units:**

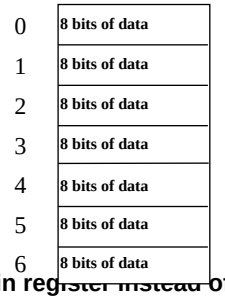
- elements that operate on data values (combinational)
- Elements that contain state (sequential)





Memory Organization

- **"Byte-addressing" means that the index points to a byte of memory**
  - **8-bit bytes are useful**
  - **a word occupy 4 bytes**
  - **address of sequential words differ by 4**



- **2^{32} bytes with byte addresses from 0 to 2^{32}-1**
- **2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4**
- **Data access are faster and useful if data is kept in register instead of memory**
  - **This is due to RC delay and arithmetic instruction can read two register**
  - **To achieve highest performance, MIPS compilers must use register efficiently**
- **Summary : Fig.3.4**

Our First Example

- **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:
    muli $2, $5, 4
    add $2, $4, $2
    lw $15, 0($2)
    lw $16, 4($2)
    sw $16, 0($2)
    sw $15, 4($2)
    jr $31
```

So far we've learned:

- **MIPS**
  - **— 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`

So far we've learned:

- **MIPS**
  - **— loading words but addressing bytes**
  - **— arithmetic on registers only**
- **Can we figure out the code?**

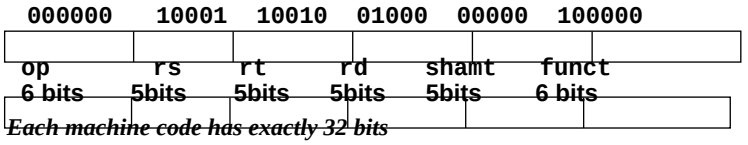
```
swap(int v[], int k);
{ int temp;
      temp = v[k]
      v[k] = v[k+1];
      v[k+1] = temp;
}
```

```
swap:
      muli $2, $5, 4
      add $2, $4, $2
      lw $15, 0($2)
      lw $16, 4($2)
      sw $16, 0($2)
      sw $15, 4($2)
      jr $31
```

***Note: $5 stores k and $4 stores the address of v[0]**

3.4 Machine Language

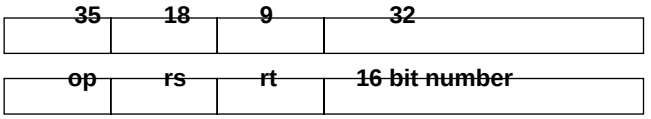
- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t0, $s1, $s2`
  - registers have numbers ⇒ $t0 to $t7: registers 8 to 15; $s0 to $s7: register 16 to 23
- Instruction Format of machine language



- *Each machine code has exactly 32 bits*
  - *there are six fields in R-type instruction*
  - *Can you guess what the field names stand for?*
- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do? 5 bits for constant in this type of Ins ?
  - New principle: Good design demands a compromise

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
  - other format was R-type for register
- Example: `lw $t0, 32($s2)`



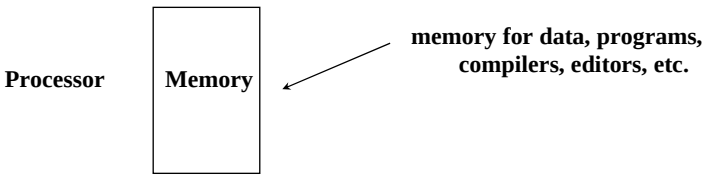
- Where's the compromise?

Machine Language

- Where's the compromise?
  - Constant can have $\pm 2^{15}$ ➠range of $\pm$ 8192 (2^{13}) words
  - The first three fields are the same ➠the complexity of decoding is reduced (Fig.3.5)
- Do example in p.119 and see Fig.3.6
- Stored Program Concept
  - Instructions are bits,
  - programs are stored in memory and to be read or written just like data
- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the “next” instruction and continue

Stored Program Concept

- Instructions are bits
- Programs are stored in memory
  — to be read or written just like data



- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the “next” instruction and continue

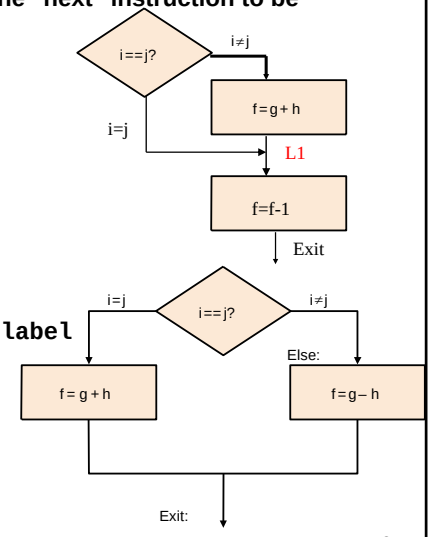
3.5 Make decision: control the flow of program execution

- Decision making instructions
  - alter the control flow, i.e., change the "next" instruction to be executed
- MIPS conditional branch instructions:

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

  - Example,p.123: if (i==j) go to L1;
    f= g+h;
    L1: f= f- i;

```
    beq $s3, $s4, L1
    add $s0, $s1, $s2
L1: sub $s0, $s0, $s3
```



- Unconditional branch instructions: `j label`
  - Example, p.124 if (I ==j)f=g+h;
    else f=g-h;

```
bne $s3,$s4, Else
add $s0,$s1,$s2
j Exit
Else: sub $s0,$s1,$s2
Exit:
```


Make decision: Loops /Branch

- Loop with variable array index
  - Example, p.126

```
Loop: g=g+A[I];
      i=i+j;
      if (i != h) go to Loop;
Loop: add $t1,$s3,$s3
      add $t1, $t1,$t1
      add $t1,$t1,$s5
      lw $t0,0($t1)
      add $s1,$s1,$t0
      add $s3,$s3, $s4
      bne $s3,$s2,Loop
```

  Note:g,h,I,j ->$s1,$s2,$s3,$s4;base of A is in $s5

- While Loop: see p.127
  - Formats:

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

So far:

- **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
bne $s4,$s5,L       Next instr. is at Label if $s4 ° $s5
beq $s4,$s5,L       Next instr. is at Label if $s4 = $s5
j Label             Next instr. is at Label
```

Make decision: IF-Then/Switch

- We have: beq, bne, what about Branch-if-less-than?
  - New instruction: slt;set on less than

```
if  $s0 < $s1 then  go to Less
slt $t0, $s1, $s2
bne $t0,$zero,  Less
```

- Can use this instruction to build "`blt $s1, $s2, Label`"
  — can now build general control structures
- Note that the assembler needs a register to do this,
  — there are policy of use conventions for registers
- Case/switch statement: see p.129
  - New instruction, jr :jump register
- Summary: Fig.3.9

3.6 Supporting procedures in computer hardware

- A procedure or subroutine is used to make program more structure
  - –easier to understand, and code reused
  - –pass values and return results through
- To execute a procedure, the computer must do the six steps: p.132
  - –MIPS allocates seven registers for procedure calling:$a0-$a3, $v0-$v1 and $ra
  - –A jump-and-link instruction (jal) for the procedure: It jumps to an address and simultaneously saves the address of the following instruction (PC+4) in register $ra
- Stack: last-in-first-out queue
  - –push: placing data onto the stack, pop:removing data from the stack
  - –stack pointer: stores in $sp register
  - –do example in p.134
- Nested procedures and recursive procedure
- summary of procedure call:Fig.3.11

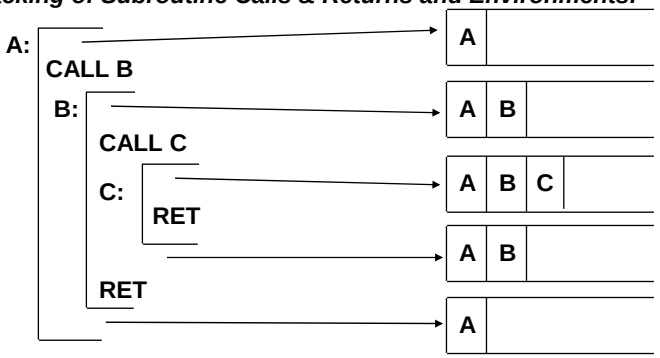
Registers in MIPS

0	zero constant 0	
1	reserved for assembler	
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	
16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (cont)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	frame pointer
31	ra	Return Address (HW)

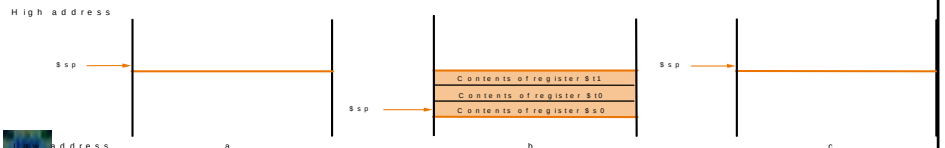
Plus a 3-deep stack of mode bits.

Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:



Some machines provide a memory stack as part of the architecture (VAX)
Sometimes stacks are implemented via software convention (MIPS)



3.8 Other styles of MIPS addressing

- MIPS provided two more ways of accessing operands
  - constant or immediate operands: faster to access small constants
  - J-type jump instruction
- Small constants are used quite frequently (50% of operands)

  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:I-type for constant is 16 bits

    addi $29, $29, 4
    slti $8, $18, 10
    andi $29, $29, 6
    ori $29, $29, 4
  - Example in P.145
- We'd like to be able to load a 32 bit constant into a register
  - Must use two instructions, new "load upper immediate" instruction

Constants

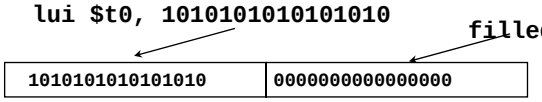
- Small constants are used quite frequently (50% of operands)
  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;
- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.
- MIPS Instructions:

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

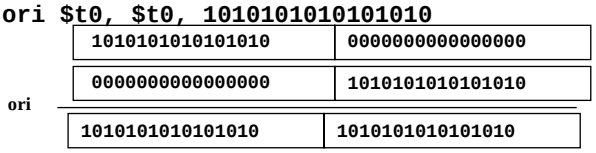
- How do we make this work?

How about larger constants?

- `lui $t0, 1010101010101010`

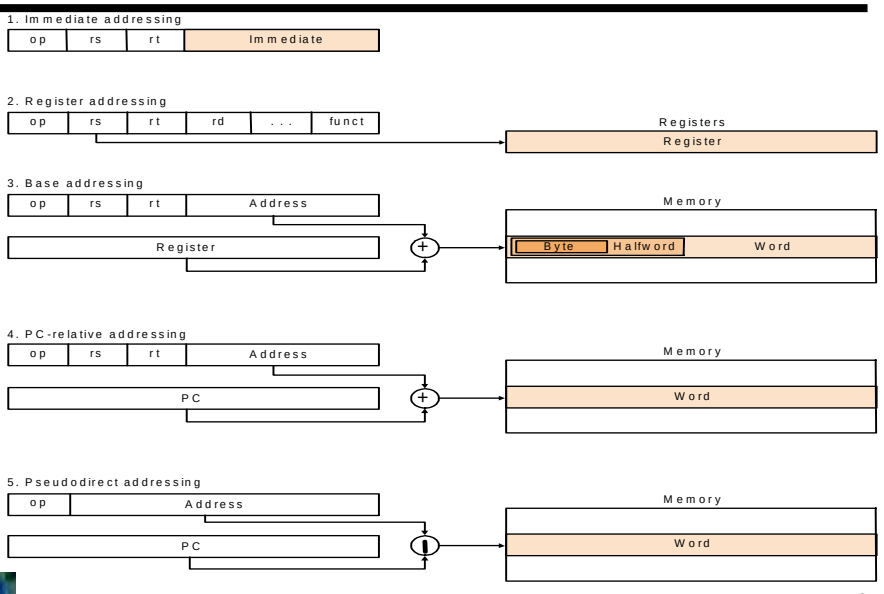


- Then must get the lower order bits right, i.e.,
  `ori $t0, $t0, 1010101010101010`



- Addressing in branches and jumps: example in p.149
- MIPS addressing mode summary
  - Register addressing
  - Base or displacement addressing
  - Immediate addressing
  - PC-relative addressing
  - Pseudodirect addressing

Five MIPS addressing modes



To summarize:

MIPS operands

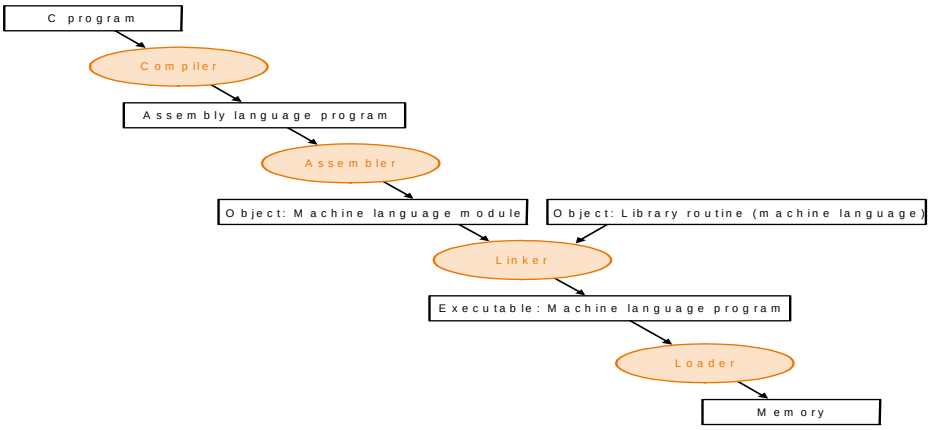
Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} 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

3.9 Running a program

- **A translation hierarchy**



Assembly Language vs. Machine Language

- **Assembly provides 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 'pseudoinstructions'**
  - **e.g., “move $t0, $t1” exists only in Assembly**
  - **would be implemented using “add $t0,$t1,$zero”**
- **When considering performance you should count real instructions**

3.12 Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as “RISC vs. CISC”
  - 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 and 80x86

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
  - a special counter register “`bc Loop`”

    decrement counter, if not 0 goto loop

80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions
  (mostly designed for higher performance)
- 1997: MMX is added

- 2001: Pentium 4, 144 new insructions
  "This history illustrates the impact of the "golden handcuffs" of compatibility

  "adding new features as someone might add clothing to a packed bag"

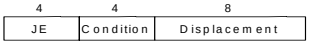
  "an architecture that is difficult to explain and impossible to love"

A dominant architecture: 80x86

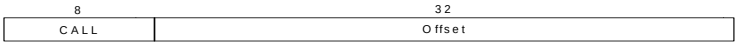
- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes, eg., "base or scaled index with 8 or 32 bit displacement"
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

  "what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective"

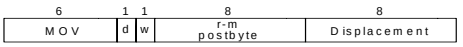
a. JE EIP + displacement



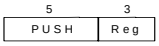
b. CALL



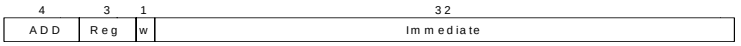
c. MOV EBX, [EDI + 45]



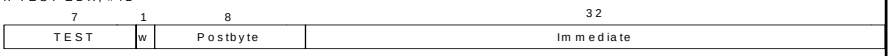
d. PUSH ESI



e. ADD EAX, #6765



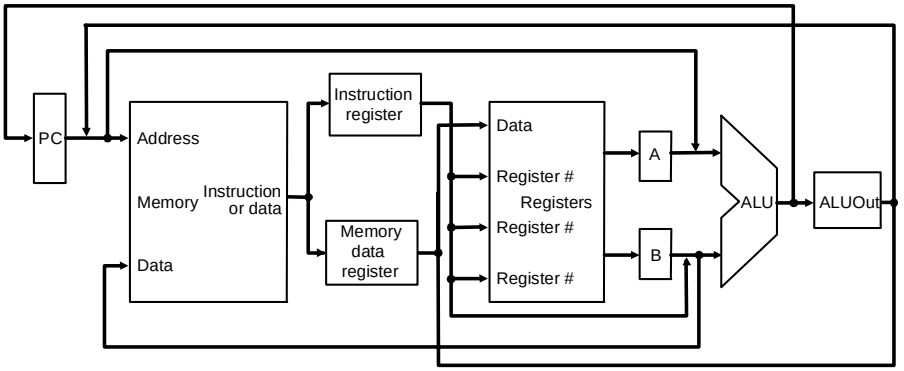
f. TEST EDX, #42



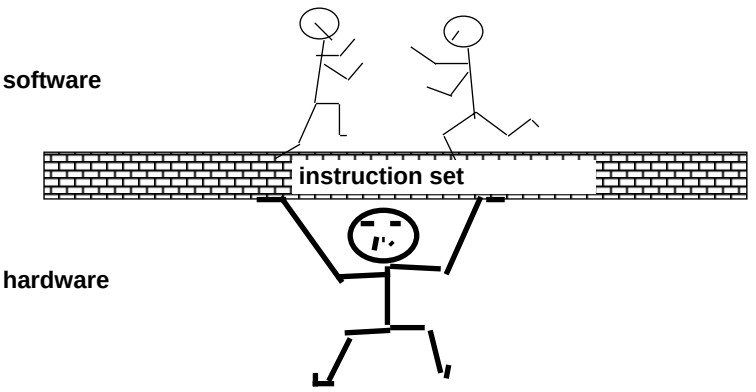
Summary

- **Instruction complexity is only one variable**
  - **lower instruction count vs. higher CPI / 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!**
- **Fig.3.39**

Fig.5.30 of book



The Instruction Set: a Critical Interface



MIPS R3000 Instruction Set Architecture (Summary)

- **Instruction Categories**
  - **Load/Store**
  - **Computational**
  - **Jump and Branch**
  - **Floating Point**
    - coprocessor
  - **Memory Management**
  - **Special**

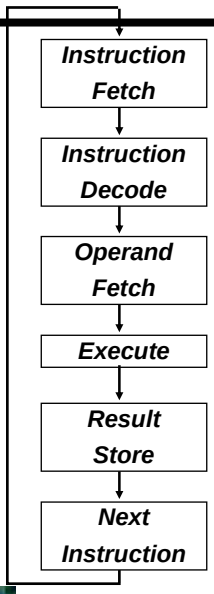
Registers

R0 - R31
PC
HI
LO

3 Instruction Formats: all 32 bits wide

OP	rs	rt	rd	sa	funct
OP	rs	rt	immediate		
OP	jump target				

Execution Cycle



Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

Compute result value or status

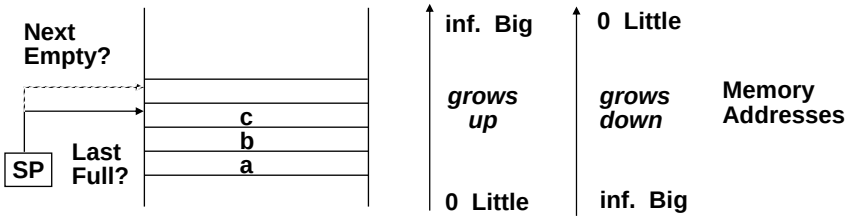
Deposit results in storage for later use

Determine successor instruction

Memory Stacks

Useful for stacked environments/subroutine call & return even if operand stack not part of architecture

Stacks that Grow Up vs. Stacks that Grow Down:



How is empty stack represented?

Little --> Big/Last Full

POP: Read from Mem(SP)
Decrement SP

PUSH: Increment SP
Write to Mem(SP)

Little --> Big/Next Empty

POP: Decrement SP
Read from Mem(SP)

PUSH: Write to Mem(SP)
Increment SP

Addressing Modes

Addressing mode	Example	Meaning
Register	Add R4,R3	R4← R4+R3
Immediate	Add R4,#3	R4 ← R4+3
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Direct or absolute	Add R1,(1001)	R1 ← R1+Mem[1001]
Memory indirect	Add R1,@(R3)	R1 ← R1+Mem[Mem[R3]]
Auto-increment	Add R1,(R2)+	R1 ← R1+Mem[R2]; R2 ← R2+d
Auto-decrement	Add R1,?R2)	R2 ← R2□; R1 ← R1+Mem[R2]
Scaled	Add R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]

Why Auto-increment/decrement? Scaled?

Addressing Mode Usage? (ignore register mode)

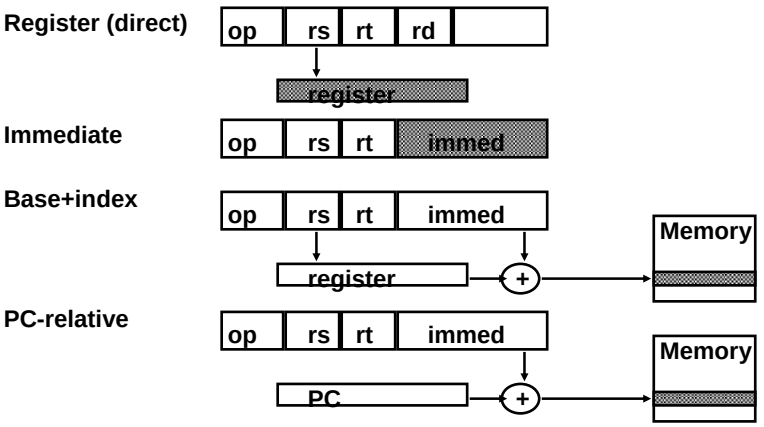
3 programs

--- Displacement:	42% avg, 32% to 55%	75%	
--- Immediate:	33% avg, 17% to 43%		85%
--- Register deferred (indirect):	13% avg, 3% to 24%		
--- Scaled:	7% avg, 0% to 16%		
--- Memory indirect:	3% avg, 1% to 6%		
--- Misc:	2% avg, 0% to 3%		

75% displacement & immediate
88% displacement, immediate & register indirect

MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide



- Register Indirect?

Typical Operations (little change since 1960)

Data Movement	Load (from memory) Store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
Arithmetic	integer (binary + decimal) or FP Add, Subtract, Multiply, Divide
Shift	shift left/right, rotate left/right
Logical	not, and, or, set, clear
Control (Jump/Branch)	unconditional, conditional
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronization	test & set (atomic r-m-w)
String	search, translate
Graphics (MMX)	parallel subword ops (4 16bit add)

Top 10 80x86 Instructions

° Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

° Simple instructions dominate instruction frequency