Instructions (354 Review)

Instructions are the “words” of a computer

Instruction set architecture (ISA) is its “vocabulary”

With a few other things, this defines the interface of computers

But implementations vary

We use MIPS ISA: simple, sensable, used in games & supercomp

Most common: x86 (IA-32): Intel Pentium* & PC-compatible proc.

Others: PowerPC (Mac), SPARC (Sun), Alpha (Compaq), ...

We won’t write programs

Forecast

Basics

Registers and ALU ops

Memory and load/store

Branches and jumps

And more . . .

Basics

C statement

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

MIPS instructions

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

Opcodes/mnemonic, operands, source/destination

Basics

Opcode: Specifies the kind of operation (mnemonic)

Operands: input and output data (source/destination)

Operands `t0` & `t1` are temps

One operation, two inputs, one output

Multiple instructions for one C statement

Why not have bigger instructions?

Why not have "`f = (g + h) - (i + j)`" as one instruction?

Church's thesis: A very primitive computer can compute anything that a fancy computer can compute - need only logical functions, read and write memory and data-dependent decisions

Therefore, ISA selected for practical reasons

performance and cost, not computability

Regularity tends to improve both

- E.g., H/W to handle arbitrary number of operands
  - complex and slow and NOT NECESSARY

Registers and ALU ops

Ok, I lied!

Operands must be registers, not variables

- `add $8, $17, $18`
- `add $9, $19, $20`
- `sub $16, $8, $9`

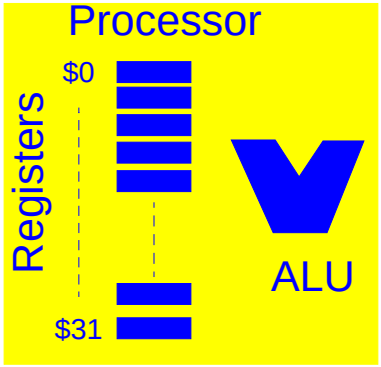
MIPS has 32 registers `$0-$31` (figure next slide)

`$8` & `$9` are temps, `$16` is `f`, `$17 g`, `$18 h`, `$19 i`, & `$20 j`

MIPS also allows one constant called "immediate"

- later we will see immediate is 16 bits [-32768, 32767]

Registers and ALU



ALU ops

Some ALU ops:

- `add, addi, addu, addiu` (immediate, unsigned)
- `sub` . . .
- `mul, div` - weird!
- `and, andi`
- `or, ori`
- `sll, srl,` . . .

Why registers? fit in instructions, smaller is faster

Are registers enough?

Memory and load/store

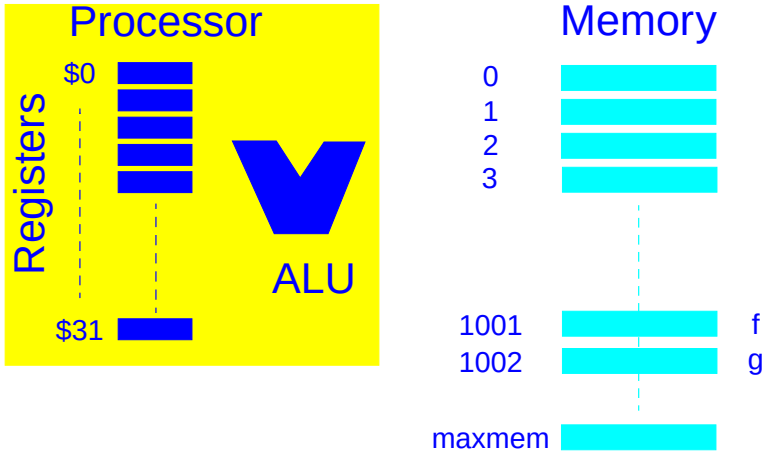
But need more than 32 words of storage

An array of locations M[addr], indexed by addr (figure next slide)

Data movement (on words or integers)

- load word for reg <-- memory
- `lw $17, 1002` # get input `g`
- store word for reg --> memory
- `sw $16, 1001` # save output `f`; Note: destination last!

Memory and load/store

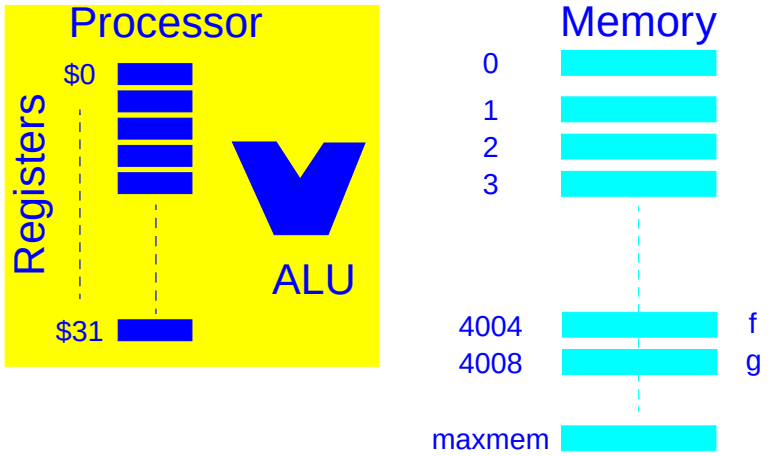


Memory and load/store

I lied again!

- We need address bytes for character strings
- Words take 4 bytes
- Therefore, word addresses must be multiples of 4
- Figure next slide

Memory and load/store



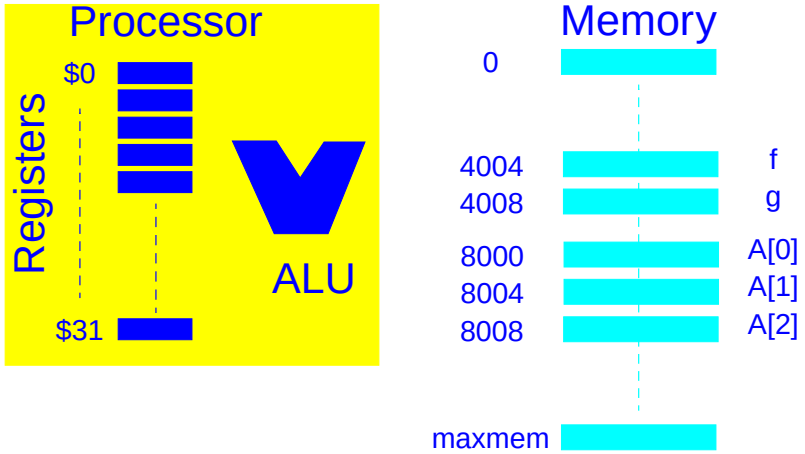
Memory and load/store

Important for arrays

- `A[i] = A[i] + h` (figure next slide)
- # `$8` - `temp`, `$18` - `h`, `$21` - `(i x 4)`
- `Astart` is `8000`
- `lw $8, Astart($21)  # or 8000($21)`
- `add $8, $18, $8`
- `sw $8, Astart($21)`

MIPS has other load/store for bytes and halfwords

Memory and load/stor4



Aside on “Endian”

Big endian: MSB at address xxxxxx00

- e.g., IBM, SPARC

Little endian: MSB at address xxxxxx11

- e.g., Intel x86 (Windows NT requires it)

Mode selectable

- e.g., PowerPC, MIPS

Branches and Jumps

```
While ( i != j) {
   j= j + i;
   i= i + 1;
}     # $8 is i, $9 is j, $10 is k
Loop: beq $8, $9, Exit          # not !=
         add $9, $9, $8
         addi $8, $8 , 1
         j     Loop
Exit:
```

Branches and Jumps

Better yet:

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

Let compilers worry about such optimizations

Branches and Jumps

What does **bne** do really?

read **$8**; read **$9**, compare

set **PC = PC + 4** or **PC = Target**

To do compares other than **=** or **!=**

- e.g., **blt $8, $9, Target** # assembler pseudo-instr
- expanded to
- **slt $1, $8, $9** # **$1** == (**$8**<**$9**) == (**$8**-**$9** < 0)
- **bne $1, $0, Target** # **$0** is always **0**

Branches and Jumps

Other MIPS branches/jumps

beq $8, $9, imm # if ($8 == $9) PC = PC + imm<<2 else PC += 4

bne . . .

slt, sle, sgt, sge

- with immediate, unsigned

j addr # PC = addr

jr $12 # PC = $12

jal addr # $31 = PC+4; PC = addr; used for procedure calls

Layers of Software

Notation - program: input data -> output data

- executable: input data -> output data
- loader: executable file -> executable in memory
- linker: object files -> executable file
- assembler: assembly file -> object file
- compiler: HLL file -> assembly file
- editor: editor commands -> HLL file

Only possible because programs can be manipulated as data

MIPS Machine Language

All 32-bit instructions

A.L. **`add $1, $2, $3`**

```
3322222222221111111111000000000
10987654321098765432109876543210
```

M.L. 00000000010000110000100000010000

000000 00010 00011 00001 00000 010000

alu-rr 2 3 1 zero add/signed

Instruction Format

R-format

opcode rs rt rd shamt funct

6 5 5 5 5 6

Digression:

How do you store the number **`4,392,976`**?

- Same as **`add $1, $2, $3`**

Stored program: instructions are represented as numbers

- programs can be read/written in memory like numbers

Instruction Format

Other R-format: addu, sub, subi, etc

A.L. **`lw $1, 100($2)`**

M.L. 100011 00010 00001 0000000001100100

lw 2 1 100 (in binary)

I-format

opcode rs rt address/immediate

6 5 5 16

Instruction Format

I-format also used for ALU ops with immediates

`addi $1, $2, 100`

001000 00010 00001 0000000001100100

What about constants larger than 16 bits = [-32768, 32767]

1100 0000 0000 0000 1111?

```
lui $4,12      # $4 == 0000 0000 1100 0000 0000 0000 0000
ori $4,$4,15   # $4 == 0000 0000 1100 0000 0000 0000 1111
```

All loads and stores use I-format

Instruction Format

`beq $1, $2, 7`

000100 00001 00010 0000 0000 0000 0111

PC = PC + (0000 0111 << 2) # word offset

Finally, J-format

`j address`

opcode addr

6 26

addr is weird in MIPS: 4 MSB of PC // addr // 00

Summary: Instruction Formats

R-format: opcode rs rt rd shamt funct

6 5 5 5 5 6

I-format: opcode rs rt address/immediate

6 5 5 16

J-format: opcode addr

6 26

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

Practice: Instruction bits -> control signals

Procedure Calls

See section 3.6 for more details

• save registers	Proc: save more registers
• set up parameters	do function
• call procedure	set up results
• get results	restore more registers
• restore registers	return

jal is only special instruction: the rest is software convention

Procedure Calls

An important data structure is the stack

Stack grows from larger to smaller addresses

`$29` is the stack pointer, it points just beyond valid data

Push **`$2`**:	Pop **`$2`**:
• **`addi $29, $29, -4`**	**`lw $2, 4($29)`**
• **`sw $2, 4($29)`**	**`addi $29, $29, 4`**

• the order cannot be changed. why? interrupts

Procedure Example

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

$4 is v[] & $5 is k -- 1st & 2nd incoming argument
$8, $9 & $10 are temporaties that callee can use w/o saving

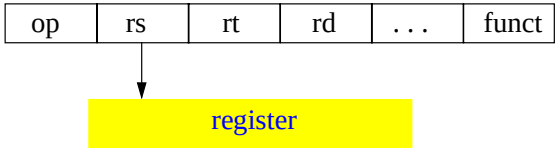
```
swap: add $9,$5,$5  # $9 = k+k
      add $9,$9,$9  # $9 = k*4
      add $9,$4,$9  # $9 = v + k*4 = &(v[k])
      lw  $8,0($9)  # $8 = temp = v[k]
      lw  $10,4($9) # $10 = v[k+1]
      sw  $10,0($9) # v[k] = v[k+1]
      sw  $8,4($9)  # v[k+1] = temp
      jr  $31       # return
```


Addressing modes

There are many ways of getting data

Register addressing

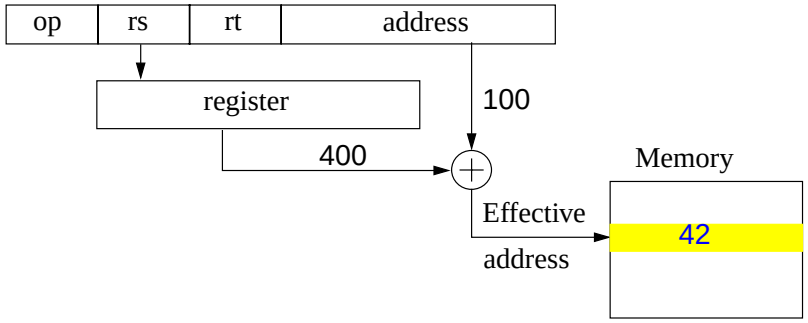
```
add $1, $2, $3
```



Addressing Modes

Base addressing (aka displacement)

```
lw $1, 100($2)    # $2 == 400, M[500] == 42
```



Addressing Modes

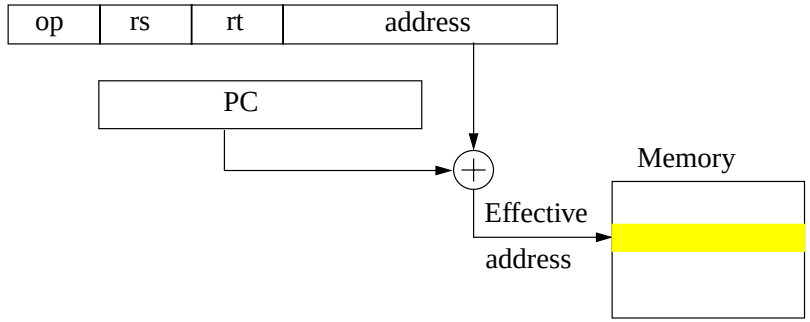
Immediate addressing

```
addi $1, $2, 100
```

Addressing Modes

PC relative addressing

```
beq $1, $2, 100 # if ($1 == $2) PC = PC + 100
```



Addressing Modes

Not found in MIPS

Indexed: add two registers - base + index

Indirect: M[M[addr]] - two memory references

Autoincrement/decrement: add data size

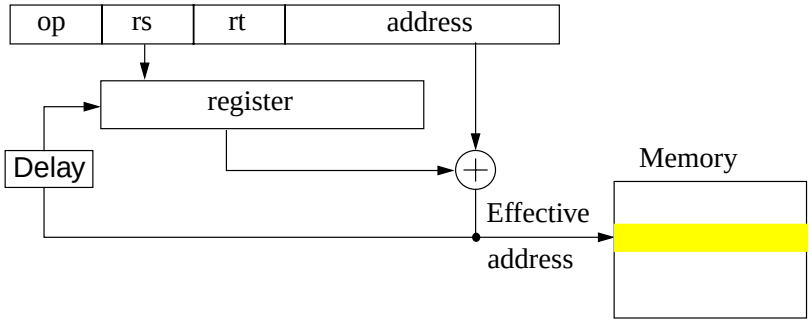
Autoupdate - found in IBM PowerPC, HP PA-RISC

- like displacement but update register

Addressing Modes

Autoupdate

```
lwupdate $1, 24[$2] # $1 = M[$2+24]; $2 = $2+24
```



Addressing Modes

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

$7 - sum, $8 - address of a[i], $9 - N, $2 - tmp, $3 - i*4

```
inner:

  lw $2, 0($8)
  addi $8, $8, 4
  add
```

```
new inner:

   lwupdate $2, 4($8)

   add $7, $7, $2
```

Any problems with new inner ? before the loop: sub $8, $8, 4

How to Choose ISA

Minimize what?

- Instrs/prog x cycles/instr x sec/cycle

In 1985-1995 technology, simple modes like MIPS good.

As technology changes, computer design options change

For memory is small, dense instructions important

For high speed, pipelining important

Intel x-86 (IA-32)

Year	CPU	Comments
1978	8086	16-bit with 8-bit bus from 8080; selected for IBM PC - golden handcuffs!
1980	8087	Floating Point Unit
1982	80286	24-bit addresses, memory-map, protection
1985	80386	32-bit registers and addresses, paging
1989	80486	
1992	Pentium	
1995	Pentium Pro	few changes; 1997 MMX

Intel 80386 Registers & Memory

Registers

- 8 32-bit registers (but backward 16 & 8b: EAX, AX, AH, AL)
- 4 special registers: stack (ESP) & frame (EBP) pointers, ...
- Condition codes: overflow, sign, zero, parity, & carry
- Floating point uses a 8-element stack (re-used by MMX)

Memory

- Flat 32-bits or segmented (rarely used, but must support)
- Effective address =
base_reg + (index_reg x scaling_factor) + displacement

Intel 80386 ISA

Two register machine: src1/dst src2

- reg - reg, reg - immed, reg - mem, mem - reg, mem - imm

Examples

```
mov EAX, 23      # places 32b 2SC imm 23 in reg EAX
neg [EAX+4]      # M[EAX+4] = -M[EAX+4]
faddp ST(7),ST   # ST = ST + ST(7)
jle label        # PC = label if Sign Flag or Zero Flag set
```

Intel 80386 ISA, cont.

Decoding nightmare

- instructions 1 to 17 bytes
- prefixes, postfixes
- crazy “formats” - e.g., register specifiers move around
- but key 32-b 386 instructions not terrible
- yet got to make all work correctly

Current Approach

Current technique in Pentium Pro

- Instruction decode logic translates into “RISCy Ops”
- Execution unit runs RISCy ops

+ Backward compatibility

– Complex decoding

+ Execution unit as fast as RISC

We work with MIPS to keep it simple and clean

Learn x86 on the job!

Complex Instructions

More powerful instructions not necessarily faster execution

E.g. - string copy

option 1: move with repeat prefix for memory to memory move

- special-purpose

option 2: use loads into register and then stores

- generic instructions

option 2 faster on the same machine!

Concluding Remarks

Simple and regular

- same length instructions, fields in same place

Small and fast

- Small number of registers

Compromises inevitable

- Pipelining (buffet concept) should not be hindered

Common case fast