**PHBREAK:**

**RISC ISP architecture**
**the MIPS ISP**

you read:

text Chapter 3

**Summary of main points:**

**Two objectives;**

1] Describe the MIPS ISP architecture

2] expose the Reduced Instruction Set Computer (RISC) approach to architecture

**RISC approach: what it is not:**

**CISC a la S/360, VAX (1970s)**

$M_p$ is slow

( no caches,
cycle times of 1-6 microseconds
[vs. 100 nsec = 0.1 microsec today] )

so instruction fetches are expensive,
so let's make every instruction do a lot

let's mimic higher-level contructs, eg

¶ loop control (S/360 BXLE)

¶ stack push/pop (Burroughs B-5000, VAX)

¶ procedure call instruction (VAX)

"wired macroinstructions"

in general, lots of side-effects per instruction\

{ we can implement these easily (for free?),
by writing long microroutines in vertical microstore}

What happened?

seemed OK thru the 1970s, but in the 80s

¶ $M_p$ got a lot faster, esp. with caches

- Microstore became as slow as Mp
- People increasingly needed to use compilers
  - but compilers couldn’t always generate efficient CISC code

- Programmers spent pages setting up a killer effect, so

- code was hard to understand or modify

- solution: a form of KISS:
- Reduced Instruction Set Computer

**RISC approach: what it is:**
**Rationale**

Reduced (small) set of simple instructions

¶ able to be used effectively by compilers

get rid of the slow microprogram store

i.e. instructions implemented by wired-logic controls

wired-logic decoders will be feasible and fast,
as the instructions are simple and few in number

programs will have more instructions, but
$M_p$ is now big (>1 Mbyte) and fast (<100 nsec)

**RISC Empirical result:**

In executing (e.g.) compiled C code

the product

(# of instrs executed) * (mean execution time per instruction)

is usually smaller for RISC than for CISC

the simpler control design was amenable to VLSI (single-chip cpus) so

the microprocessor world (MIPS, SPARC, PowerPC) is now all RISC

except(!) Intel . . . and Motorola 68X00

but it could all change tomorrow.

**MIPS architecture**

**(note simplicity w r to S/360, VAX)**

ALL instrs have exactly 3 operands (KISS)

there are just 32 fast registers, \$0 - \$31.

c(\$0) = 0, always.

$2^{30}$ memory cells,

4 bytes wide and byte addressed.

¶ *Aligned* word data begin at byte adresses of form 4n.

¶ Index registers must be incremented by 4 when addressing word data.

ALL instructions 32 bits (1 word) long

## MIPS Instruction Formats

some instructions (eg binary ops) are RR format:

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 | 5 | 5 | 5 | 5 | 6 |

c(rs) bop c(rt) -> rd (Register Transfer)

```
bop     $d, $s, $t              #Assembler
```

Example: If

```
val(f) is in  $16
      g          $17
      h          $18
      i          $19
      j          $20
```

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

compiles into

```
add $8,   $17, $18      # g+h in $8
add $9,   $19, $20      # i+j in  $9
sub $16,  $8,   $9      # f gets answer
```

# I-type (Register-Storage) format:

- some instructions (eg mem -> reg, reg -> mem) are I-type (Register-Storage) format:

op rs rt address
6 5 5 16

or, more naturally,

op ri rd address (op, index, dest'n, addr)

c[c(ri) + *address* ] <-> rd (RTL)

```
op      $d, address($i)         #comments

op      $t, address($s)

lw      $8,Astart($19)
                        #r8 <- c(Astart + c(r19))
```

Note: address is only 16 bits but addresses are 30 bits

Example:

swap(v[k], v[k+1}) is

in C:

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

in MIPS asm: if

v is in $4

k is in $5 (MIPS parameter passing convention)

then

```
muli    $2,   $5,4  #$2 has k*4, needed for
                    #word addressing

add     $2,   $4,   $2    # v +k*4 in $2, DIY
                          indexing, to form
                          # address of v[k]

lw      $15, 0($2) #$15 has temp = v[k]
lw      $16, 4($2) #$16 has v[k+1]
sw      $16, 0($2)
sw      $15, 4($2)
```

c[c(ri) + *address* ] <-> rd

is the only memory addressing mode, the assembler provides several others (later)

# Tests & Branches:

```
beq     $1,     $2, label       #goto label iff
                                # c($1) = c($2)

bne     $1,     $2, label       #goto label iff
                                #c($1) .ne. c($2)
```

so
if f,g,h,i,j live in $16 - $20 then

```
        if (i == j) goto L1;
        f = g + h;
  L1:   f = f - h;
```

compiles to

```
        beq   $19,$20, L1  #goto L1 if i=j
        add   $16,$17,$18 # f = g + h
L1:     sub   $16,$16,$19 #f = f - i
```

the format: i format like lw, sw:

| op | rs | rt | addr |
|---|---|---|---|
| 6 5 | 5 | 16 | |

The C code

```
if (i == j) f = g+h; else f = g -h:
```

compiles to

```
            bne     $19,$20,Else          #if i .NE. j
            add     $16,$17,$18 # f = g+h
            j       Exit
Else:       sub $16,$17,$18      #f = g-h
Exit:
```

note the labels created by the compiler

Less-than test:

```
slt     $r,$d, $t    #c($r) = 1 iff
                             #c($d) < c($t);
                     #else c($r) = 0

. . .

blt     $r, $d, label  #cf s/360 setting condition
             code
```

implemented as

```
slt     $1, $r,$d    #$1 gets 1 if a<b
bne     $1,$0, Less          #if $1 != $0 ie a<b
```

- assembler instruction nonexistent in hardware

In fact

blt $d, $t , label is *assembled into*

```
slt   $1,   $d, $t
bne   $1,   $0, label     #always use $1 - convention
```

(blt in hardware would take 2 clock cycles or stretch the clo
interval)

# Case Statement:
# (C's switch)

format:

| slt | $r | $t | $d | shamt | funct |
| --- | --- | --- | --- | --- | --- |
| 6 | 5 | 5 | 5 | 5 | 6 |

Now

The address field L1 is 16 bits.
So, to avoid limiting programs to $2^{16}$ bytes,

branch target is *PC relative addressed*. Thus if

c(PC) = 1000

beq 19 20 100

branches to 100 + c(PC) = 1100 if c($19) = c($20)

where PC is 32 bits

leading us to the . . .

# Digression

# Addressability

- definition: generating jump addresses in a space big enougl span the $M_p$ address space

- Why care?

- Sometimes, we don't:

IBM 7090 (1962-66)

- 36 bit word

- one-word instructions comprising

op code | index reg | address
---------------------------
15 bits

- $2^{15}$ = 32K was maximum primary memory size, so



- no addressability problem

- recent trends to larger $M_p$ :

  $2^{24}$ bytes (IBM System 360/370/390)

  $2^{30}$ bytes (MIPS 3000 & 4000 chip)

  and to no increase in instruction size:

  32 bits (IBM 360/370/390)

  32 bits (MIPS chip)

  16 bits (DEC PDP-11/VAX,
  for some instructions)

  mean that an $M_p$ address can't fit into an instruction

# Addressability - some solutions

1] addressing relative to the Program Counter or PC (Instruc
Address Register or IAR)

DEC PDP-11 (inventor?) and now
MIPS chip

2] memory organized as 4 *banks* and the contents of a 2-bi
*bank register* always prepended to the PC value

Control Data CDC 3600

*write bank register* instruction

a serious source of programming bugs
(forgot to change banks)

3] base-displacement addressing (IBM S/360)

address = disp field (12 bits) + $c(R_j)$ (32 bits)
*displacement* *base register*

achievement: only 12 bits (disp) + 4 bits (select BR)
16 bits of instruction space used to address $2^{32}$ by
of $M_p$

assembler has to calculate displacements
from symbolic labels
( Jump Foo ) so

assembler must know $c(R_j)$ *at assembly time*

solution: assembler statements

USING $R_j$ /*directive to the assembler;
does not create an instruction
for the hardware
followed immediately by */

LA Rj, ADC /* Load Address is a machine
instruction,
adcon is an address
constant */

ADC: (beginning of code)

• the above must appear at least every $2^{12}$ = 4K bytes throughout the source

( called, "establishing addressability")

## End of Digression on Addressability

Loops:

the C fragment

```
Loop:   g = g + A[i];
        i = i + j;
        if (i != h) goto Loop;
```

if

A[100] and if

g,h,i,j -> $17 , $18, $19, $20

4 -> $10

then

```
Loop:   mult    $9, $19,$10        #c($9) = i*4
        lw      $8, Astart($9)     #c($8) = A[i]
        add     $17, $17, $8       # g = g + A[i]
        add     $19, $19, $20      # i = i + j
        bne     $19,$18, Loop
```

will do it

# Case statement:

in C:

```
switch (k)  {

        case 0: f = i + j; break; /* if k=0 */

        case 1: f = g + h; break; /* if k =1 */

        case 2: f = g - h; break;

        case 3: f = i - j; break

        }                    /* here after  break */
```

Assume:

```
Jmptbl:Lbl0     #address of Lbl0 in  jmptbl
        Lbl1    # jmptbl + 4 (byte-addressed!)
        Lbl2    # jmptbl + 8
        Lbl3    # jmptbl + 12
```

f,g,h,i,j,k in $16, $17, . . ., $21
4 in $10

```
Switch:mult     $9, $10, $21 # $9 has k*4

        lw      $8, Jmptbl($9)      # $8 has jump addr

        jr      $8                  # go there

Lbl0:   add     $16, $19, $20       # k=0 so f <- i + j
        j       Exit                # the break

Lbl1:   add     $16, $17, $18       # k = 1 case
        j       Exit                # the break

Lbl2:   sub     $16, $17, $18       # k = 2
        j       Exit

Lbl3:   sub     $16, $19, $20
Exit:
```

# Procedure Call

## need to

1] jump to *Proc* and remember where we came from so we
can do the *return*

2] change scope of variables (procedural languages), or

3] switch contexts (same concept, operating
systems jargon) or

4] save and reload a bunch of registers
(ISP jargon)

Minimal RISC MIPS only does the minimum - 1]

Maximal CISC VAX does it all! (Appendix E)

Jump and Link (JAL) procaddr

1] save where we are in $31

c(PC) + 4 -> $31

2] jump to procaddr

$c(c(PC)_{[\text{bits } 15\text{-}31]}) = \text{this instr}_{[\text{bits } 15\text{-}31]}$

-> PC

How to switch context/ save registers? (eg $31)

- a stack, of course

MIPS conventions:

$29 is stack pointer SP

stack grows into lower addresses
(subtract 4 from SP to push a word
add 4 to SP to pop a word)

to minimize proc call overhead due to register saves restores:

proc params are in $4 -$7, extras on stack

callee saves (preserves) values in
$16 -$23, used by compiler for
long-lived values

nobody but caller saves $$8-$15
and $24-$25 ( PH page A-23)

# Immediate operands
# (efficient access to little constants)

- concept: put the constant itself, not its address in memory, the instruction
- pro:

  saves one memory cell
  saves one cycle to get the operand
- con

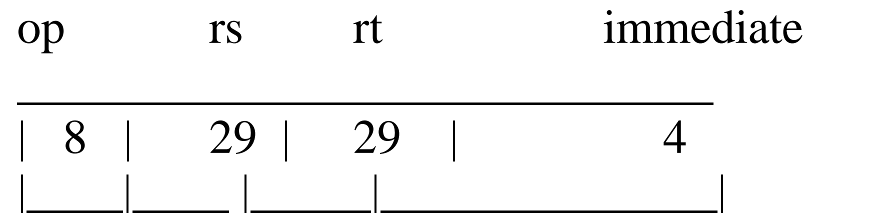
  only room for 16-bit (halfword) constants
  in the instruction
- example:

```
# add 4 to c($29)
        lw      $24, Four($0)       #c(Four) = 4
        add     $29, $29, $24

# using immediate operands
        addi    $29, $29, 4
```



which assembles to




#compare c($18) to 12

```
        slti    $8, $18, 12
```

# load 0000 0000 0011 1101
# 0000 1001 0000 0000 into $16

```
        lui     $16, 61
        addi    $16, $16, 2304
```

# Summary
# MIPS Addressing modes
# (p 131 PH)

## Register addressing:

rs, rt, rd are 5 bit fields pointing to registers

## base-displacement addressing:

c($rs) + disp points to a memory cell.

```
lw      $1, 100($2)
```

## immediate addressing:

low-order 16 bits of instruction *is* the data

```
addi    $1,$1, 224
```

# PC - relative

low-order 16 bits of instruction is the (branch target) address interpreted as relative to the PC

| | |
|---|---|
| 80008 | bne$8,$21, Exit |
| 80012 | |
| 80016 | |
| 80020 Exit | |

assembles to

| | | | | |
|---|---|---|---|---|
| 80008 | 5 | 8 | 21 | 8 |
| 80012 | | | | |
| 80016 | | | | |
| 80020 Exit | | | | |

# Common addressing modes
# Missing from MIPS

## auto-increment, auto decrement (DEC

```
mov ($6)+, ($3)+      # c(c($6)) -> c($3)
                                # increment c($6), c($3)

mov (SP)+, stacktop # pop stack
mov stacktop, -(SP)  # push onto stack
```

## storage-to-storage (IBM, DEC)

```
mov A,B             #A & B  mem addresses

mov ($6), ($3)      # $6 & $3 point to
                         # mem addresses
```

## super loop control (IBM)

```
BXLE     R1,R3, braddr

braddr -> PC iff c(R1) < c(R3).
                 Else R1 <- c(R1) + 1
```

# Arrays vs. Pointers

- Please study PH Section 3.11 carefully
- NB comparison of array and pointer versions of the little program