Course Notes

"Introductory Computer Architecture"

Prof. Rob Marano <robmarano@gmail.com>

based upon text:

Computer Organization & Design, Revised 4th Edition

by D. Patterson & J. Hennessy.

Special Topics on Combinational Logic

Expanded Combinational Logic ⇒ Binary addition

assume unsigned binary

$$\begin{array}{rccc}
 & \text{MSB} & \varnothing\leftarrow & \text{LSB} \\
 & 1 & 1 & \varnothing \\
+ & \varnothing & 1 & 1 \\
\hline
 & & & 1 \leftarrow \varnothing+1, \cancel{\varnothing}\ \text{carry}
\end{array}$$

– add right to left

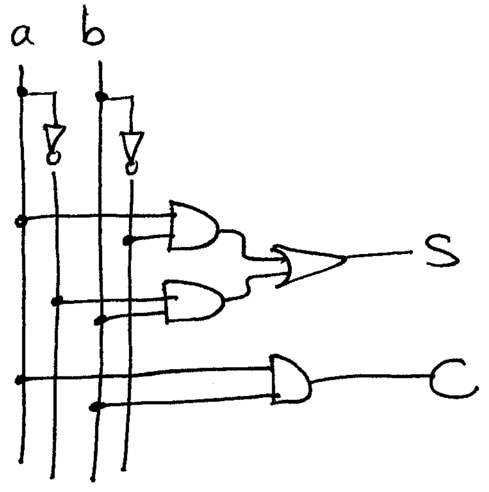
$$\begin{array}{ccc}
1\leftarrow & \varnothing & \\
1 & 1 & \varnothing \\
\varnothing & 1 & 1 \\
\hline
 & \varnothing & 1
\end{array}$$

$$\begin{array}{cccc}
 & 1 & & \\
 & 1 & 1 & \varnothing \\
 & \varnothing & 1 & 1 \\
\hline
(1) & \varnothing & \varnothing & 1
\end{array}$$

in unsigned binary; result can be in n+1 bits

Half Adders

a	b	Sum	Carry
0	0	0	0
0	1	1	0
1	1	0	1
1	0	1	0

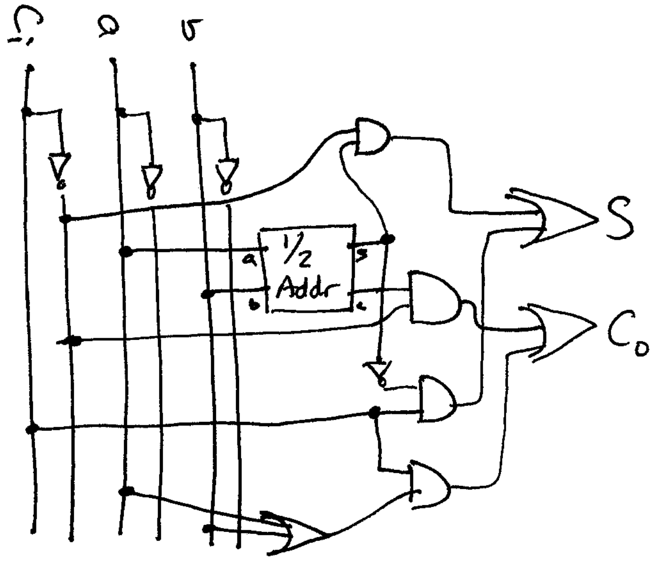


Full Adder : same as ½ adder except that it accepts a carry bit as input w/ 2 bits

C_{in}	a	b	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	1	1	0
0	1	0	0	1
1	0	0	0	1
1	0	1	1	0
1	1	1	1	1
1	1	0	1	0

$C_{in} = \emptyset$
$C_o = a \cdot b$
$S = a \oplus b$

$C_{in} = 1$
$C_o = a + b$
$S = \overline{a \oplus b}$



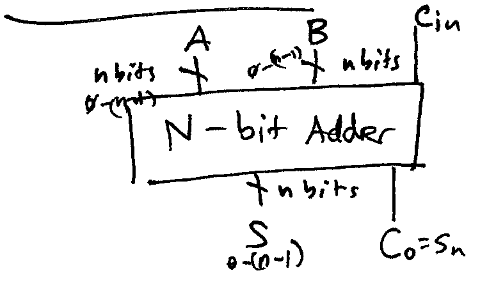
$\boxed{S}$

C_i \ ab	00	01	10	10
0		1		1
1	1		1	

checker board pattern ⇒ characteristic of XOR

$s = a \oplus b \oplus C_i$

N-bit Adder

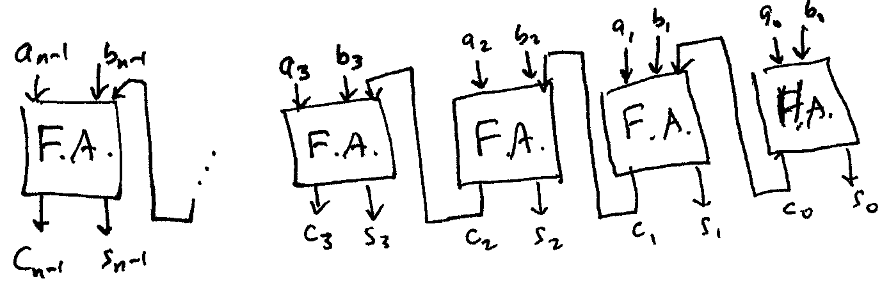


when an n-bit adder includes a carry-in (C_{in}) & a carry-out (S_n), they can be cascaded to generate an adder for a longer bit string.

2 common designs
- Ripple carry Adder
- Carry Look-Ahead Adder

Ripple-Carry Adder

- uses a ½ adder & n-1 full adders to implement an n-bit adder



n-bit ripple-carry adder

* this is a non-cascading version since it does not include a carry-in into the ½ adder block.

* worst-case time delay = # bits × gate delay × depth of circuit of full adder

∴ this delay makes ripple-carry adder impractical for most applications

Carry Look-Ahead Adder

- reduces the time delay for generating the sum of 2 n-bit binary encoded values
- increase in HW logic to reduce depth of circuit : 1958 Weinberger & Smith
  - Consider 2 ways that the ith bit position can produce a carry bit
    1. a carry may be generated locally by the ith pair of inputs (a_i, b_i)

    or

    2. a carry may be propagated through the ith bit from an earlier bit in the sequence

    ∴ we define

$g_i = a_i b_i \longrightarrow$ carry generated locally when $a_i = b_i = 1$

$p_i = a_i + b_i \longrightarrow$ carry will be propagated thru ith position if either a_i or b_i or both $= 1$

$C_{i+1} = g_i + p_i C_i \longrightarrow$ there will be a carry-out from ith position if it is generated locally or carry is propagated thru the ith position.

$C_0 = C_0$

$C_1 = g_0 + p_0 C_0$

$C_2 = g_1 + p_1 C_1 = g_1 + p_1(g_0 + p_0 C_0)$

$= g_1 + g_0 p_1 + p_1 p_0 C_0$

Chap 2.1 Intro

- instruction set ⇒ words of a computer language
- in MIPS ⇒ "one word" = 32 bits
- secret of computing ⇒ stored-program concept
  "instructions & data of many types can be stored in memory as numbers"
- we will use MIPS Technologies instruction set (RISC-based)

2.2 - Ops of the Computer HW

- perform arithmetic ⟶ sidebar: Half adders, Full Adders, & Ripple Carry Adders

```
add a, b, c # a = b+c
            ↳ comment marker.
```

- See Figure 2.1
- HW for a variable # of operands is more complicated than HW for a fixed #

(ex)
```
add a,b,c   # a = b+c
add a,a,d   # a = b+c+d
add a,a,e   # a = b+c+d+e
```

- Underlying principles of HW Design (1st 4)

• $\boxed{\text{Design Principle 1 : SIMPLICITY FAVORS REGULARITY}}$

(ex) a = b+c; d = a-e; } C lang ⇒ ASM { add a, b, c sub d, a, e

(ex) f = (g+h) - (i+j); } C lang

⇓ ASM : must break into several asm instructions b/c only 1 op per MIPS instruction (RISC)

```
add t0, g, h   # ~~ft0~~ t0 is ~~temp~~ variable
add t1, i, j
sub f, t0, t1
```

2.3 - Operands of Computer HW

- operations of arithmetic occur on operands, which are restricted in ASM
- they must be from a limited # of special locations built directly in HW called REGISTERS
  ↳ primitives used in HW design

registers · $\boxed{\text{Design Principle 2 : SMALLER IS FASTER}}$

- use 32 registers for 32 bits ; byte = 8 bits ∴ 1 word = 4 bytes
- designers must balance the craving of programs for more registers with his desire to keep the clock cycle fast.

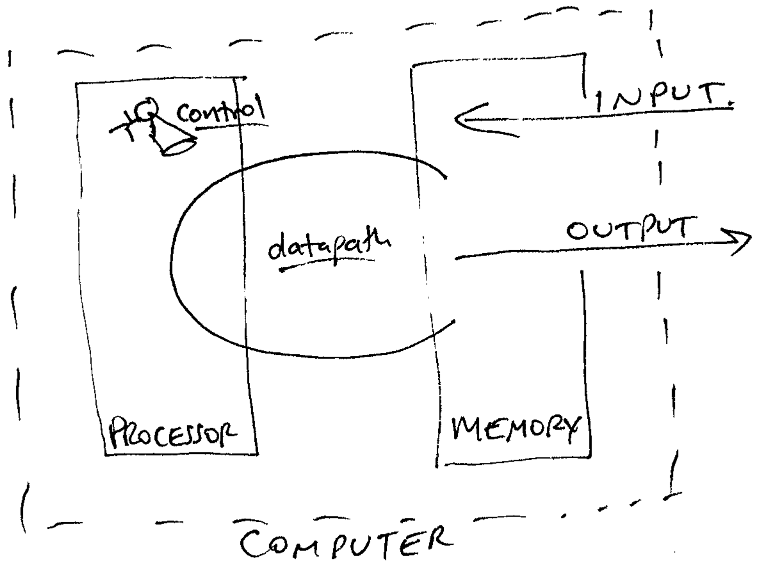
- MIPS convention ⇒ 2 character names following $ sign to represent a register

Memory operands

- complex data structures in a higher lang ⇒ represent in addressable memory

"billions" of items $\xrightarrow{\text{xfr}}$ registers "32"

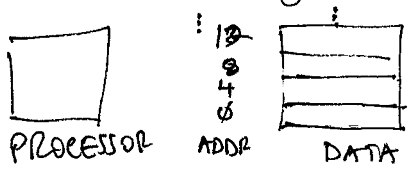
BIG PICTURE OF A COMPUTER



- arithmetic ops occur only on registers in MIPS instructions
- ∴ MIPS includes asm instructions that xfr data btwn MEMORY & REGISTERS → "data xfr instructions"

- to access a word in memory ⇒ need "memory address"

- MEMORY ≡ large single-dimension array with the ADDRESS acting as INDEX to that array, starting @ ∅.



- data xfr instr. that copies data from MEMORY to REGISTER is "load" ⟹ actual MIPS instr: lw ≡ load word
  sw ≡ store word
- (ex) compile this C statement: g = h + A[8];

```
lw  $t0, 8($s3)     # temp reg $t0 = A[8]
add $s1, $s2, $t0   # g = h + A[8]
                    #$s1  $s2  $t0
```

* called OFFSET
* called BASE REGISTER

- In MIPS ⟹ words MUST start @ addresses that are multiples of 4.
  ⟹ called ALIGNMENT RESTRICTION

BYTE ORDER
- BIG END : use addr of the $\underbrace{\text{leftmost byte}}_{\text{big-endian}}$ as the word addr

BYTE #			
0	1	2	3

- LITTLE END : use the $\underbrace{\text{right most byte}}_{\text{little-endian}}$ as the word addr

3	2	1	0

- byte addressing also effects the array index.

- instruction to store ⟹ SW ≡ store word

(ex) A[12] = h + A[8]; # in C

```
lw  $t0, 32($s3)   # temp reg = A[8] => 32 = 8x4   (alignment)
add $t0, $s2, $t0  # temp reg = h + A[8]
sw  $t0, 48($s3)   # stores h + A[8] = A[12] => 48 = 12x4
```

- Constants or immediate operands
  2 methods ⟹ ① load constants, placed when program loaded)
  ② quick add instruction "add immediate" – addi

(ex) addi $s3, $s3, 4 # $s3 = $s3 + 4 (constant)

$\boxed{\text{Design Principle 3 : MAKE THE COMMON CASE FAST}}$

- Constant operands occur frequently, & by including constants inside arithmetic instructions, ops are much faster & use less energy than if constants were loaded from memory

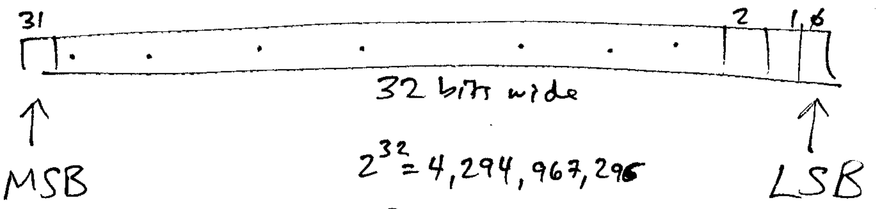
2.4 Signed & Unsigned Numbers

- humans → think in base 10
- computers → in binary digits ⇒ bits

} $d \times Base^i$

ex $1011_2 = (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{10}$
$= 11_{10}$

- # bits 0–31 from RIGHT to LEFT ⇒ MIPS word



$2^{32} = 4,294,967,296$

represent 2^{32} different 32-bit patterns
$0 \rightarrow 2^{32} - 1$

- <u>TWO'S COMPLEMENT</u> ⇒ to represent negative #'s : Beware programmer, Logical for HW designer
  * leading 0's ⇒ positive #
  * leading 1's ⇒ negative #

  32 bits ⇒ $-(2^{31}) \leq x \leq (2^{31} - 1)$

  $-2,147,483,648$ ⋮ $2,147,483,647$
  ⇓
  DOES <u>NOT</u> have a corresponding (+) #

- in MIPS : 32 bits ; the MSB ≡ sign bit

* Program want to deal sometimes w/ (+) #'s OR (−) #'s & sometimes only (+) #'s

  in C lang ⇒ unsigned int ⇒ $0 \rightarrow 2^{32}$
  int ⇒ $-2^{31} \leq x \leq (2^{31} - 1)$

– quick way to negate 2's complement binary #
- invert every digit
- add 1 to result

why? ∴ $s + \bar{s} = -1$

$s + \bar{s} + 1 = \emptyset$

$\bar{s} + 1 = -s$

(ex) 2_{10} = 0000 0000 0000 0000 0000 0000 0000 0010

negate ⇒ 1111 1101

add 1 ⇒ 1

1111 . . . . 1110 ⇒ where ∅ ⇒ −(value if 1) shift 1 left

= −2

– Sign Extension : Convert a 16-bit binary version of 2 and −2 to 32 bit binary #'s

(a) 2 = 0000 0000 0000 0010 (MSB)

↓ convert by making 16 copies of the value in the MSB, here ∅, and placing that in the LEFT HAND HALF of the word; the right half gets the old value:

×16

= 0000 0000 0000 0000 ⁞ 0000 0000 0000 0010

(b) Let's negate the 16-bit version of 2

0000 0000 0000 0010 → 1111 1111 1111 1101
+ 1

1111 1111 1111 1110 (MSB)

(c) convert from 16- to 32-bits

1111 11111111 1111 1111 1111 1111 1110

✱ This trick works b/c (+) 2's complements really have an infinite # of ∅'s on the left & (−) 2's complements have an ∞ # of 1's on the left.

✱ 2's complement gets its name from the rule that unsigned sum of an n-bit # & its negative is 2^n.

∴ the complement or negation of a 2's complement # X is $2^n - X$

- inaddition to sign/magnitude $\overset{-\emptyset ? +\emptyset ?}{}$ & 2's complement representation of negative #'s ⟹ 1's Complement

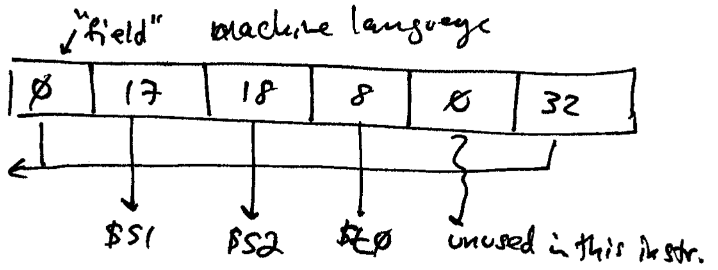
  * the negative of a one's complement ⟹ complement of X
    (a) invert each bit $= 2^n - x - 1$

  - attempt to be a better soln than sign & magnitude
  - like sign/magnitude ⇒ 1's complement has 2 zeros : (+) ∅
    (−) ∅
  - (+) & (−) #'s balanced
  - requires an extra step to subtract a # ⇒ hence 2's complement better

2.5 Representing Instructions in the Computer

- instructions are kept in computer as series of 1's & ∅'s } physical representation
  $\overset{\downarrow}{+5V}$ $+3V$ $\overset{\downarrow}{0V}$
- each instr ≡ unique #
- pattern of instr ≡ forms execution
- convention to map register names to #'s
- in MIPS $S∅ - $S7 ≡ registers 16 - 23
  $t∅ - $t7 = " 8 - 15

(ex) add $t∅, $s1, $s2 ⇒



"R-Type" instruction format

OP	RS	RT	RD	SHAMT	FUNCT
6bits	5bits	5bits	5bits	5bits	6bits
opcode	1st reg source	2nd reg source	reg dest	shift amt	function code

* Since all computer data sizes are multiples of 4, HEXADECIMAL (base 16) #'s are popular.

$\boxed{\text{HEX:}}$ ∅−9 ; A = 10
B = 11
C = 12
D = 13
E = 14
F = 15

"R-Type" {
op = "opcode" — basic operation of instruction
rs = 1st reg source operand
rt = 2nd reg source operand
rd = reg destination → gets result
shamt = shift amount
funct = function code → selects specific variant in opcode

· $\boxed{\text{Design Principle 4 : GOOD DESIGN DEMANDS GOOD COMPROMISE}}$

- MIPS : Kept all instr same length
  ∴ differing instruction formats for different kinds of instr.

  R-type (register)
  I-type (immediate) ⇒ immediate & data xfr instruction

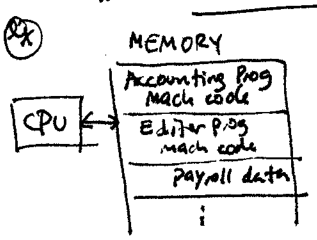
"I-format/type"

op	rs	rt	const or addr
opcode	1st reg src operand	2nd reg src operand	
6 bits	5 bits	5 bits	16 bits

↳ 16-bit addr ⇒ lw can load any word within a region of $\pm 2^{15}$ (32,768 bytes)
or $\pm 2^{13}$ words (8192) & constants limited to $\pm 2^{15}$
↳ $4 = 2^2$ bytes/word

Ex) Convert C lang to ASM to machine lang < p 98-99 >

* STORED-PROGRAM CONCEPT of a computer



- instructions represented as #'s ⇒ combinational logic
- programs are stored in memory ⇒ sequential logic to be read or written

2.6 Logical Operations

— Operations to simplify the packing & unpacking of bits into words

	Logical operations	C	Java	MIPS asm
shifts	shift left	<<	<<	sll - shift logical left
shifts	shift right	>>	>>>	srl - shift logical right
	bit-by-bit AND	&	&	and, andi
	bit-by-bit OR	|	|	or, ori
	bit-by-bit NOT	~	~	nor

(ex) $\underbrace{\text{0000 ...5... 0000 1001}}_{\text{in } \$s0} = 9 \xrightarrow[\text{by 4 bits}]{\text{srl}} \underbrace{\text{0000 ...5... 1001 0000}}_{\text{in } \$t2} = 144$ $144 = 9 \times 16$, 2^4

⇓ MIPS

sll \$t2, \$s0, 4 # reg \$t2 = reg \$s0 << 4 bits

"SHAMT" - "shift amount"
- field in the R format
- used in shift instructions

machine language version of sll \$t2, \$s0, 4

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

Review

MIPS instruction encoding

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{10}	n.a.
sub	R	0	reg	reg	reg	0	34_{10}	n.a.
add immediate	I	8_{10}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{10}	reg	reg	n.a.	n.a.	n.a.	address (16-bit)
sw (store word)	I	43_{10}	reg	reg	n.a.	n.a.	n.a.	address (16-bit)

0-31

(ex) A[300] = h + A[300]; \$t1 has base of array "A"
\$s2 has "h"

↓ compiled into MIPS asm

```
lw $t0,1200($t1)   # temp reg $t0 gets A[300]
add $t0,$s2,$t0    # temp reg $t0 gets h + A[300]
sw $t0,1200($t1)   # store h + A[300] back into A[300]
```

↓ machine lang

	op	rs	rt	rd	address/shamt	funct
I-type	35	9	8	1200		
R-type	0	18	8	8	0	32
I-type	43	9	8	1200		

→ offset to select A[300]
$= 300 \times 4 = 1200$
↑ for alignment

Shift left → by i bits gives same result ~~of~~ $* 2^i$

just like shifting a decimal # by i digits is $* 10^i$

logical AND ⇒ isolates fields; "MASKING"

logical OR ⇒ dual of AND

"NOT" as NOR in MIPS ⇒ in keeping w/ 3-operand format, MIPS designers decided to include NOR (not or) instead of NOT specifically.

⇒ if one operand is ∅, then that is equivalent to

= NOT : A NOR ∅

⇒ NOT (A OR ∅)

= NOT (A)

2.7 - Instructions for Making Decisions

* a computer differs from a calculator b/c it can make decisions.

MIPS: 2 decision-making instructions, similar to an if with a goto.

Conditional branches

(a) beq reg1, reg2, L1 — "go to statement labeled "L1" IF the value in reg1 is EQUAL to value in reg2"

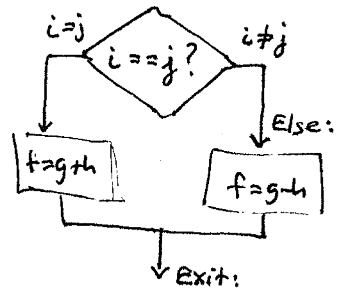
"beq" = BRANCH IF EQUAL

(b) bne reg1, reg2, L1 — "go to statement labeled "L1" IF value reg1 NOT EQUAL to value in reg2"

"bne" = BRANCH IF NOT EQUAL

ex

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



f → \$s0
g → \$s1
h → \$s2
i → \$s3
j → \$s4

MIPS:

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

MIPS: j ≡ "jump" to statement label

$\boxed{\text{Loops}}$ - iteration a computation

WHILE LOOP

C: while (save[i] == k)
     i += 1;

i ⇒ $s3
k ⇒ $s5
save[] ⇒ $s6

MIPS:

```
Loop: sll $t1, $s3, 2          # $t1 = i * 2^2      (shamt ... 4)
      add $t1, $t1, $s6        # $t1 = addr of save[i] ; add $t1 & base save[]
      lw $t0, 0($t1)           # $t0 = save[i] ; load save[i] into $t0
      bne $t0, $s5, Exit       # goto Exit if save[i] ≠ k
      addi $s3, $s3, 1         # i = i + 1
      j Loop                   # goto Loop label
Exit: ...
```

BASIC BLOCK ≡ sequence of instructions w/o branches, except possibly @ the end, & w/o branch targets or branch labels, except possibly @ the beginning.

* one of the early phases of compilation is breaking the program into basic blocks.

TEST for EQUALITY / INEQUALITY

slt ≡ SET on less than } Compares 2 regs & sets the 3rd reg to 1 if first < second

ex slt $t0, $s3, $s4 # $t0 = 1 IF $s3 < $s4

↓ IMMEDIATE version for working w/ constants

slti $t0, $s2, 10 # $t0 = 1 IF $s2 < 10

~~***~~ register $zero ≡ fixed value of zero

*~~xw~~ extra reading ⇒ von Neumann's warning about the simplicity of the "equipment"

$\boxed{\text{Comparisons}}$ with SIGNED & UNSIGNED #'s
* 1 in MSB ⇒ most likely a NEGATIVE #. b/c POSITIVE #'s have 0 in MSB
* for unsigned #'s → 1 in MSB is ALWAYS > 0

} slt and slti ⇒ work w/ SIGNED integers

sltu ⇒ works w/ UNSIGNED integers
sltiu

* Treating signed #s as if they were unsigned gives us a low cost way of checking $0 \le x < y$, which matches the index out-of-bounds checks for arrays.
  Key*: NEGATIVE integers in 2's complement notation look like LARGE #'s in unsigned notation

(ex) Bounds check shortcut: — use this shortcut to reduce an index-out-~~of~~-bounds check: jump to IndexOutOfBounds if $s1 ≥ $t2 OR $s1 < 0
— uses sltu to do both checks

```
sltu $t0, $s1, $t2   # $t0 = 0 if $s1 ≥ length OR $s1 < 0
beq $t0, $zero, IndexOutOfBounds   # if bad, go to ERROR
```

Case/Switch Statement:

select 1 of many depending upon a single value.

- can either sequence of conditional ~~t~~ests (chain of if-then-else)
  OR
- encode as a table of addresses of alternative instruction sequences
  $\boxed{\text{JUMP ADDRESS TABLE / JUMP TABLE}}$
  ↳ prog needs only to index into the table & then jump to appropriate sequence

JUMP ^address TABLE ≡ array of words containing addresses that correspond to labels in code.

* MIPS → jump register instruction (jr)
  jr ≡ unconditional jump to address specified in reg.

2.8 - Supporting Procedures in Computer HW

"procedure" or "function" ≡ tool used to structure programs & allow code to be reused + also easier to understand flow

SIX STEPS program MUST follow in execution of a procedure

1 - Place parameters in a place where the procedure can access them.
2 - Transfer control to the procedure.
3 - Acquire the storage resources needed for the procedure.
4 - Perform the desired task.
5 - Place the result value in a place where the calling program can access it.
6 - Return control to the point of origin, since the procedure can be called from several points in a program.

* registers are FASTEST place to hold data.

MIPS: following convention for procedure calling in allocating its 32 registers

$a0 - $a3 : 4 argument registers in which to pass parameters

$v0 - $v1 : 2 value registers in which to return values

$ra : one return address register to return to point of origin
↳ register 31 (last reg on MIPS)

JUMP-AND-LINK instruction "jal" ≡ jumps to an address & simultaneously saves the address of the following instruction in register $ra

ex jal ProcedureAddress

$ra ≡ return address

JUMP REGISTER instruction "jr" ≡ unconditional jump to the address in reg.

ex jr $ra

Caller (calling prog) puts parameter values in $a0-$a3 and uses jal X to jump to procedure X (called "callee").
The Callee then performs the calculations, places results in $v0 & $v1, and returns control to caller by jr $ra

* PROGRAM COUNTER (PC): Implicit in the stored-program idea is the need to have a register to hold the address of the current instruction being executed. For historical reasons this reg called PROGRAM COUNTER, abbreviated PC.
jal instruction actually saves PC + 4 in reg $ra to link to the following instr to set up the procedure return

Using More Registers in Procedures
* SPILL regs to MEMORY.
STACK: ideal data structure to spill regs (LIFO)
STACK POINTER: – points to most recently allocated address in the stack to show where the next procedure should place the regs to be spilled OR where old reg values found.
– adjusted by one word for each reg saved or restored
– MIPS: reg 29 ≡ $sp
STACKS grow from higher addr to lower addrs ⟹ adding to $sp SHRINKS the stack, popping values off. push values onto stack by subtracting from the $sp

(ex) Compiling a C procedure that does NOT call another procedure → called: "leaf procedures"

"C" lang

```
int leaf_example(int g, int h, int i, int j) {
    int f;
    f = (g+h) - (i+j);
    return f;
}
```

g → \$a0   f → \$s0
h → \$a1
i → \$a2
j → \$a3

⇓ compile into MIPS

```
"proc label" leaf_example:
                          addi $sp, $sp, -12     # adjust stack to make room for 3 items (12 = 3×4)
"save orig values of    • sw $t1, 8($sp)         # save reg $t1 for later
 specific regs you will • sw $t0, 4($sp)         # save $t0       "
 use so as to preserve    sw $s0, 0($sp)         # save $s0       "
 for caller"

"body of procedure"       add $t0, $a0, $a1      # $t0 = g+h
                          add $t1, $a2, $a3      # $t1 = i+j
                          sub $s0, $t0, $t1      # $s0 = f = $t0 - $t1

"prep return value        add $v0, $s0, $zero    # return f
 into $v0"

"restore reg for caller"  lw $s0, 0($sp)         # restore $s0 for caller
                        • lw $t0, 4($sp)         # restore $t0 for caller
                        • lw $t1, 8($sp)         # restore $t1 for caller
"adjust stack for caller" addi $sp, $sp, 12      # adjust stack to "delete" 3 items
"jump back" ⇒ to caller   jr $ra                 # jump back to calling routine
```

"•" not necessary because

*** : To avoid saving & restoring a reg whose value is NEVER USED, which might happen w/ a temp reg, MIPS separates 18 of the regs into 2 groups

① \$t0 - \$t9 : 10 temp regs that are NOT preserved by the callee (called procedure) on a procedure call

② \$s0 - \$s7 : 8 saved regs that MUST be preserved on a procedural call, if used, the callee saves & restores them.

* THIS CONVENTION → reduces reg spilling.

Nested Procedures - ex) recursive procedures

"C" Code

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

parameter "n" ⇒ argument reg $a0

⇓ compile into MIPS asm

```
fact:
    addi $sp, $sp, -8      # adjust stack to save/preserve 2 items
    sw   $ra, 4($sp)       # save return address
    sw   $a0, 0($sp)       # save argument "n"

    slti $t0, $a0, 1       # test for n<1
    beq  $t0, $zero, L1    # if n>=1, goto L1

    addi $v0, $zero, 1     # return 1 (end of recursion)
    addi $sp, $sp, 8       # pop 2 items off stack
    jr   $ra               # return to caller
L1:
    addi $a0, $a0, -1      # n>1 → argument reg gets (n-1)
    jal  fact              # call fact with $a0 = (n-1)
    lw   $a0, 0($sp)       # return from jal, restore arg n
    lw   $ra, 4($sp)       # restore return addr
    addi $sp, $sp, 8       # adjust stack ptr to pop 2 items

    mul  $v0, $a0, $v0     # return n * fact (n-1)
    jr   $ra               # return to caller
```

saves/preserves 2 regs on stack

1st time fact called store addr that called fact - test for (n<1), go to L1 if n≥1

if n<1, fact returns 1 by → in $v0. Then pops 2 saved values off stack & returns

since $a0 & $ra don't Δ to here, skip loading $a0 & $ra.

if n not <1, n then call fact

fact returns here: restore old ra & old argument, along w/ sp

next, value reg $v0 gets product of old arg $a0 & current value of $v0

fact returns

* "C" variables → has type & storage

type: int, char ...

storage:
- automatic : local to a procedure & then are discarded when proc exits.
- static : exist across exits from & entries to procedures

∴ C variables declared outside all procedures ⟹ considered static, as are any declared using "static" keyword.

* In <u>MIPS</u> : to simplify access to static data, MIPS reserves register "$gp"
GLOBAL POINTER

PRESERVED ACROSS A PROCEDURE CALL

Preserved	Not preserved
Saved regs : $s0 - $s7	Temporary regs : $t0-$t9
Stack pointer reg : $sp	Argument regs : $a0 - $a3
Return address reg : $ra	Return value regs : $v0 - $v1
Stack above $sp	Stack below the $sp

* If the SW relies on the frame pointer register OR on the global pointer register, they are also preserved.

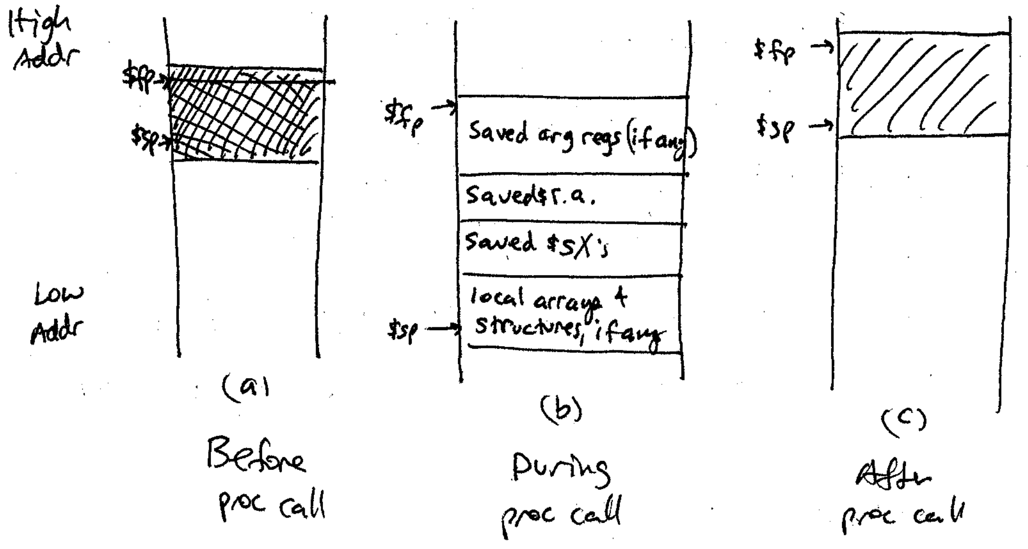
Allocating Space for New Data on the Stack

* final complexity ⟹ stack is also used to store variables that are local to the procedure but do not fit in registers, e.g. local arrays or other data structures.

"PROCEDURE FRAME" or "ACTIVATION RECORD" ≡ the segment of the stack containing a procedure's saved registers & local variables

"FRAME POINTER" ($fp) ≡
- points to the first word of the frame of a procedure
- $fp offers a stable base register within a procedure for local memory references
- a $sp may change during the procedure, & so references to a local variable in memory might have different offsets depending on where they are in the procedure, making proc harder to understand

Stack Allocation



— The frame pointer ($fp) points to the first word of the frame, often a saved argument reg, and the stack pointer ($sp) points to the top of the stack.

— The stack is adjusted to make room for all the saved registers & any memory-resident local variables.

— Since the $sp may change during program execution, it's easier for the programmer to reference variables via the stable $fp, although it could be done just with the stack pointer & a little address arithmetic.

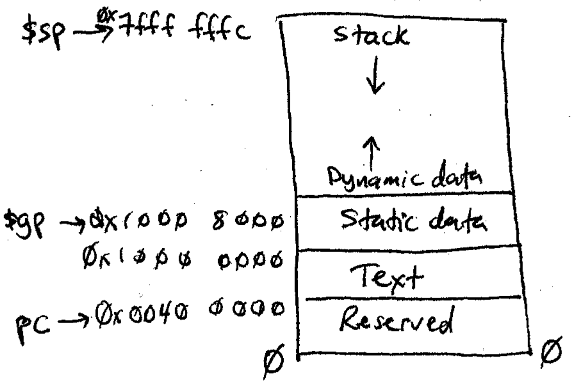
— If there are no local variables on the stack within a procedure, the compiler will save time by NOT setting & restoring the $fp.

— when a $fp is used, it is initialized using the address in $sp on a call, & $sp is restored using $fp.

Allocating Space for New Data on the Heap

* in addition to automatic variables (in procedures) → C programmers need space in memory for static variables & for dynamic data structures.

MIPS convention for memory allocation:



– Stack starts in high end of memory & grows down.

– 1st part of low end of memory is RESERVED, followed by the home of the MIPS machine code ~~program code~~ → (TEXT segment)

– Above TEXT segment is STATIC DATA SEGMENT, which is the place for constants & other static variables.

– HEAP ≡ above STATIC, & it's where dynamic structures like linked lists "live".

* STACK & HEAP grow toward each other.

C allocates & frees memory space on the heap : malloc()
free()

"bugs"
- "Memory leak" → forgetting to free space
  → eventually uses up so much memory that the OS may crash.
- "dangling pointers" → freeing space too early
  → causes pointers to point to items that program never intended.

* Java/C# ⇒ uses automatic memory allocation & garbage collection to avoid the bugs.

MIPS Register Convention

Name	Reg #	Usage	Preserved on call?
$zero	Ø	Constant value Ø	not applicable (n.a.)
$vØ - $v1	2-3	values for results & expression evaluation	no
$aØ - $a3	4-7	Arguments	no
$tØ - $t7	8-15	Temporaries	no
$sØ - $s7	16-23	Saved	Yes
$t8 - $t9	24-25	More Temporaries	no
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	3Ø	Frame Pointer	Yes
$ra	31	Return Address	Yes

* Register 1 ⇒ $at ≡ reserved for assembler (section 2.12)

* Registers 26-27 ⇒ $kØ - $k1 ≡ reserved for OS

NOTE : what if more than 4 parameters for a procedure?

- MIPS convention ≡ place extra params on the stack just above the frame pointer.
  - ∴ The procedure expects 1st 4 parameters in $aØ - $a3 AND the rest in memory addressable via the $fp.
- As mentioned in caption of Fig 2.12, $fp is convenient because all references to variables in the stack within a procedure will have the same offset. $fp is NOT necessary. ex GNU MIPS C compiler uses a $fp, but the MIPS C compiler from MIPS the company does NOT. it treats reg 3Ø as another save reg ($s8)

2.9 Communicating with People

- computers invented to crunch numbers. BUT when they became commercially viable → used to process text.
  "human"
  8-bit bytes to represent strings : ASCII
  "American Standard Code for Info Interchange"

Due to popularity of text in programs:

- MIPS provides instructions to move bytes

  lb ⇒ Load Byte from memory, placing it in the <u>rightmost</u> 8 bits of a reg
  *treated as <u>signed</u>

  sb ⇒ Store byte into memory from <u>rightmost</u> 8 bits of a reg.
  * treated as <u>signed</u>

  * signed vs. unsigned applies to loads as well as to arithmetic.
  - the <u>function</u> of a signed load is to copy the sign repeatedly to fill the rest of the reg
    → called "signed extension"
    → but the <u>purpose</u> is to place a correct representation of the number in that reg
  - "unsigned loads" simply fill with ∅'s to the <u>left</u> of the data
  - when loading a 32-bit word into a 32-bit register, the point is moot - signed & unsigned loads are identical.

∴ MIPS offers 2 flavors of loads for bytes (a) <u>load byte</u> (lb) - treats byte as signed & thus sign-extends to fill the 24-left most bits of the reg

(b) <u>load byte unsigned</u> (lbu) - works w/ unsigned integers

- Characters (represented by unsigned integers) for text processing are normally combined into strings, which have a variable # of chars.

  ∴ 3 possible ways to represent a string:
  (a) 1st position reserved to give length of string
  (b) accompanying variable has the length of string (as in a structure)
  (c) the last position of the string is indicated by a ~~character~~ to mark end.

* C language ⇒ use (c) → "null" in ASCII '~~\0~~∅' to mark end of string.
→ called "null byte termination"

Compiling a string copy procedure (using C strings)

C-code
using arrays NOT pointers

```
void strcpy (char x[], char y[])
{  int i;
   i = Ø;
   while ((x[i] = y[i]) != '\Ø')   /* copy & test byte */
   {
      i += 1;
   }
}
```

"copies string y to string x, using null byte termination convention in C"

⇓ compile into assembly for MIPS:

* Assume that { · base addresses for arrays x and y are found in $aØ and $a1
· i is in $sØ

* strcpy → adjusts $sp and then saves the saved reg $sØ on the stack:

MIPS assembly code - using arrays NOT pointers in C

```
strcpy:
    addi $sp, $sp, -4       # adjust stack for one more item
    sw   $sØ, Ø($sp)        # save $sØ

                            # to initialize i to Ø, the next instruction sets $sØ to Ø
                            # by adding Ø to Ø & placing that sum in $sØ
    add $sØ, $zero, $zero   # i = Ø + Ø

                            # This is the start of the loop.
                            # The address of y[i] is first formed by adding i to y[]
L1: add $t1, $sØ, $a1       # address of y[i] in $t1

                            # NOTE: we don't have to multiply i by 4 since y is an array of bytes
                                    not of words, as in prior examples.

                            # To load the char in y[i], we use load byte unsigned, which puts
                            # the char into $t2
    lbu $t2, Ø($t1)         # $t2 = y[i]

                            # Similar addr calc puts addr of x[i] into $t3, & then char in $t2 stored
                                                                                  in that addr.
    add $t3, $sØ, $aØ       # addr of x[i] in $t3
    sb  $t2, Ø($t3)         # x[i] = y[i]

    beq $t2, $zero, L2      # if y[i] == Ø, go to L2 (exit loop if char is Ø → end of string)
```

→

```
                        # if not, increment i & loop back:
  addi $s0, $s0, 1      # i = i+1
  j L1                  # go to L1

                        # if we don't loop back, we reached last char of string; we restore
                        # $s0 and the $sp, and then return
L2: lw $s0, 0($sp)      # y[i] == 0 : end of string; restore old $s0
  addi, $sp, $sp, 4     # pop one word off stack
  jr $ra                # return
```

* see Section 2.14 for implementation using pointers in C language ⟹ that would avoid ops on i

* Since the procedure above for strcpy is a leaf procedure, the compiler could allocate i to a temporary reg and avoid saving & restoring $s0.

* ∴ Instead of thinking of $t regs as being just for temporaries, you can think of them as regs that the callee should use whenever convenient.

* when a compiler finds a leaf procedure, it exhausts all $t regs before using regs it must save

* Characters & Strings in Java (UNICODE) → 16-bits to represent a character (2-bytes) unlike ASCII for English lang that does NOT have special lang chars.

* For supporting 2-byte chars (unicode), MIPS has explicit instructions to load & store 2-byte chars/quantities called HALFWORDS. (a) load half (lh) ⟹ loads a halfword from mem, placing it in the rightmost 16 bits of a reg, sign-extends to fill the 16 leftmost bits

(b) load half unsigned (lhu) → works w/ unsigned integers, like working w/ unicode.

(c) store half (sh) → takes halfword from the 16 rightmost bits of a reg & writes to memory

NOTE: * MIPS tries to keep the stack aligned to word addresses, allowing the program to always use lw and sw (which must be aligned) to access the stack.

* This convention means that a char variable allocated on the stack occupies 4 bytes, even though it needs less. HOWEVER, a C string variable or an array of bytes will pack 4 bytes per word, and a Java string variable or array of shorts packs 2 halfwords per word.

2.10 - MIPS addressing for 32-bit immediates and addresses

* general solution for large constants & optimizations for instruction addresses used in branches/jumps
* Keeping all MIPS instrs 32-bits long → simplifies HW.

32-bit immediate operands

- Some constants are short & fit into the 16-bit field ⟹ Some constants are bigger
- MIPS provides instr: load upper immediate (lui) → sets lower 16 bits to Ø.
    - specifically to set upper 16 bits of a constant integer in a register, allowing a subsequent instr to specify the lower 16 bits of the constant

(ex) load this 32-bit constant into $sØ:

$$C = \underbrace{0000\ 0000\ 0011\ 1101}_{61_{10}}\ \underbrace{0000\ 1001\ 0000\ 0000}_{2304_{10}}$$

```
lui $sØ, 61          # load upper 16 bits as decimal
ori $sØ, $sØ, 2304   # insert by or'ing the lower 16 bit value
```

P.129 elab
MUST do lui first b/c lui sets lower 16 bits to Ø, & addi copies leftmost bit of 16-bit immediate of instr into upper 16-bits of a word (logic immediate load Ø into upper 16.

NOTE: either compiler OR assembler must break large constants into pieces & then reassemble them into a register. ∴ assembler must have a temp reg to create long values.
$at → does this & is reserved for the assembler.

Addressing in Branches & Jumps

- MIPS jump instructions use J-type instr. format: 6 bits for op (code) field, 26 bits for addr field
- unlike jump instr, conditional branch instr. must specify 2 operands & branch address. (16 bits)
- ∴ 16-bits available for memory addressing → 2^{16} limit on programs ⟹ too small
  SO ⟹ need to specify reg used to add to branch address? calc PC = Reg + Branch Addr
  this would allow 2^{32} for prog size and still be able to use conditional branching.
- QUESTION IS: WHICH REG? ⟹ Program Counter (PC) ≡ contains the address of the current instr.
- 50% of all conditional branches in SPEC benchmarks go to locations < 16 instr away.
- You can branch within $\pm 2^{15}$ words of current instr IF we use PC as the register to be added to the addr.
- Almost all loops & IF statements are much smaller than 2^{16} words → so PC an ideal choice

"PC-Relative Addressing" ⟹ in MIPS used b/c all conditional branches' destinations is likely to be close to the branch.

* MIPS architecture → offers long addr for procedures calls by use J-type format for both jump and jump-and-link instr.

- Since all MIPS instr = 4 bytes long $\longrightarrow$ MIPS stretches the distance of the branch by having PC-relative addr refer to the # of <u>WORDS</u> to the next instr instead of the # of bytes.
  - ∴ the 16-bit field can branch 4x further
  - ∴ the 26-bit field in jump instructions is also a word addr $\rightarrow$ represents a 28-bit byte addr.

* - Since the PC is 32 bits, 4 bits must come from somewhere else for jumps.
  - The MIPS jump instruction replaces only the lower 28 bits of the PC, leaving the upper 4 bits of the PC unchanged.
  - The loader & linker (sec 2.12) must be careful to avoid placing a program across an address boundary of 256 MB (64 million instr)
  - Otherwise, a jump must be replaced by a jump reg instr preceded by other instrs to load the full 32-bit addr into a reg.

(ex) <u>Showing Branch Offset in Machine Lang</u>

$\boxed{\text{C-lang}}$

```
while (save[i] == k)
    i += 1;
```

$\Longrightarrow$

$\boxed{\text{assembly lang}}$

```
Loop: sll  $t1, $s3, 2     # Temp reg $t1 = 4*i
      add  $t1, $t1, $s6   # $t1 = addr of save[i]
      lw   $t0, 0($t1)     # temp reg $t0 = save[i]
      bne  $t0, $s5, Exit  # goto Exit if save[i] != k
      addi $s3, $s3, 1     # i = i+1
      j    Loop            # goto Loop
Exit:                      # Exit while Loop.
```

$\boxed{\text{Machine code}}$

assume place loop at mem loc 80000

addr	instructions in decimal						
80000	0	0	19	9	2	0	R-type
80004	0	9	22	9	0	32	R-type
80008	35	9	8	0			I-type
80012	5	8	21	2			I-type
80016	8	19	19	1			I-type
80020	2	20000					J-type
80024	▨						

instructions

2 words down from = 4x2 = 8 down from <u>NEXT</u> instr @80016, leading to 80024, representing "Exit"

x 4 = 80000

Most cond branches are to a nearby location, but occassionally they branch far away, farther than can be represented in the 16 bits of the conditional branch instr. $\Rightarrow$ Assembler inserts an unconditional jump to the branch target, & inverts the cond so that the branch decides ~~whether~~ whether to skip the jump.

(ex) beq \$s0, \$s1, L1

$\Downarrow$ replace w/ a pair of instrs that offer a much greater branching dist.

```
bne $s0, $s1, L2
j   L1
L2:
```

MIPS Addressing Mode Summary

Addressing mode $\equiv$ one of many addressing regimes delimited by their varied use of operands &/or addrs.

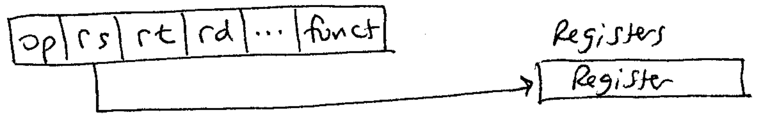
MIPS addr modes:

(1) IMMEDIATE ADDRESSING — where operand is a constant w/i instr itself

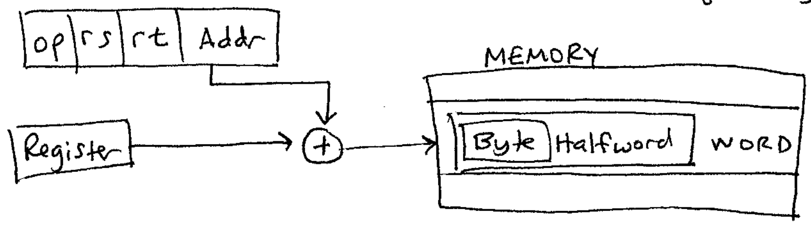
op	rs	rt	immediate

$\rightarrow$ 16 bits

(2) REGISTER ADDRESSING — where operand is a reg

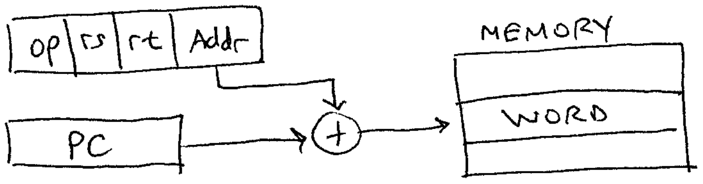


(3) BASE or DISPLACEMENT ADDRESSING — where operand is at the memory location whose addr is the sum of a reg & a constant in the instr



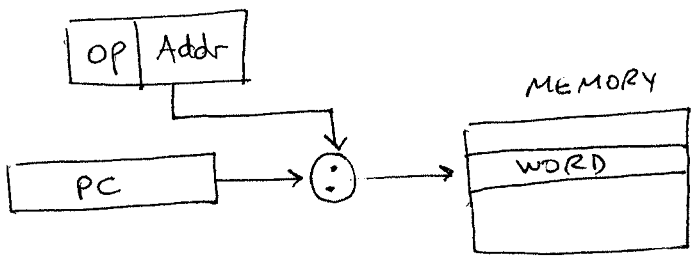
NOTE: versions of load & store access bytes, halfwords, or words

(4) PC-RELATIVE ADDRESSING — where the branch addr is the sum of the PC & a constant in instr.



Note: adding a 16-bit addr, shifted left 2 bits, to the PC

(5) PSEUDODIRECT ADDRESSING - where the jump addr is the 26 bits of the instr concatenated with the upper bits of the PC



NOTE: concatenating 26-bit addr, that is shifted left 2 bits, with the upper 4 bits of PC

NOTE ✱ : Although we show MIPS as having 32-bit addresses, nearly all μprocs have 64-bit addr extensions → support larger programs
→ also supports upward compatibility to next generation architecture.

Decoding Machine Lang - see pp 134-135
- useful for diciphering a "core dump"

2.11 Parallelism & Instructions : SYNCHRONIZATION

- parallel execution is easier when tasks (& their data) are independent, but often they need to cooperate
- Cooperation usually means some tasks are writing new values that others must read
- Synchronize tasks } to know when a task is finished writing so that it is safe for another to read
- If they don't synchronize, there is a danger of a data race, where the results of the program can change depending on how events happen to occur.

DATA RACE : Two memory accesses form a data race if they are from different threads to same location, at least one is a write, and they occur one after another.

- Synchronization mechanisms typically built with user-level SW routines that rely on HW-supplied synch instrs.
  - we focus on lock & unlock sync ops in this chp.
- Lock/unlock - create regions where only one proc can operate ⇒ called MUTUAL EXCLUSION
- critical ability to implement synchronization in a multi-processor
  ↳ set of HW primitives to ATOMICALLY READ & MODIFY a memory location

First key HW primitive to be used to build a basic sync primitive:

ATOMIC EXCHANGE / ATOMIC SWAP

- interchanges a value in a register for a value in memory.
- simple lock: $\emptyset$ = free
  1 = unavailable
  - a proc tries to set the lock by doing an exchange of 1, which is in a reg, with a memory addr corresponding to the lock.
  - the value returned from the exchange instr is 1 IF some other processor had already claimed access & is $\emptyset$ otherwise.
    - In the latter case, the value is immediately changed to 1, preventing any competing exchange in another proc from also retrieving a $\emptyset$.
  - the key to using the atomic exchange/swap primitive to implement synchronization is that the operation is ATOMIC.
  - the exchange is indivisible & 2 simultaneous exchanges are ordered by the HW
    ∴ it is impossible for 2 proc trying to set the sync variable in this manner to both think they have simultaneously set the variable.
- implementing a single atomic memory operation introduces some challenges in the design of the proc, since it requires both a memory read & a write in a single, uninterruptible instruction.
- Alternative to that single, uninterruptible instr ⟹ a pair of instr in which the 2nd instr returns a value showing whether the pair of instr was executed as if the pair were atomic.
  - the pair is effectively atomic if it appears as if ~~all other~~ ops executed by any proc occurred before or after the pair.
  - thus, when an instr pair is effectively atomic, no other proc can change the value btwn the instruction pair.
- In MIPS this pair of instr includes a special load called LOAD LINKED (ll) and a special store called STORE CONDITIONAL (sc)
- These instr used in sequence — if the contents of the memory location specified by the load linked are changed before the store conditional to the same addr occurs, then the SC fails.

- The sc is defined to both (a) store the value of a register in memory
  AND
  (b) change the value of that register to a 1 if it succeeds & to a Ø if it fails.
- Since the ll returns ~~the~~ the initial value, & the sc returns 1 ONLY if it succeeds, the following sequence implements an atomic exchange on the memory location specified by the contents of $s1 :

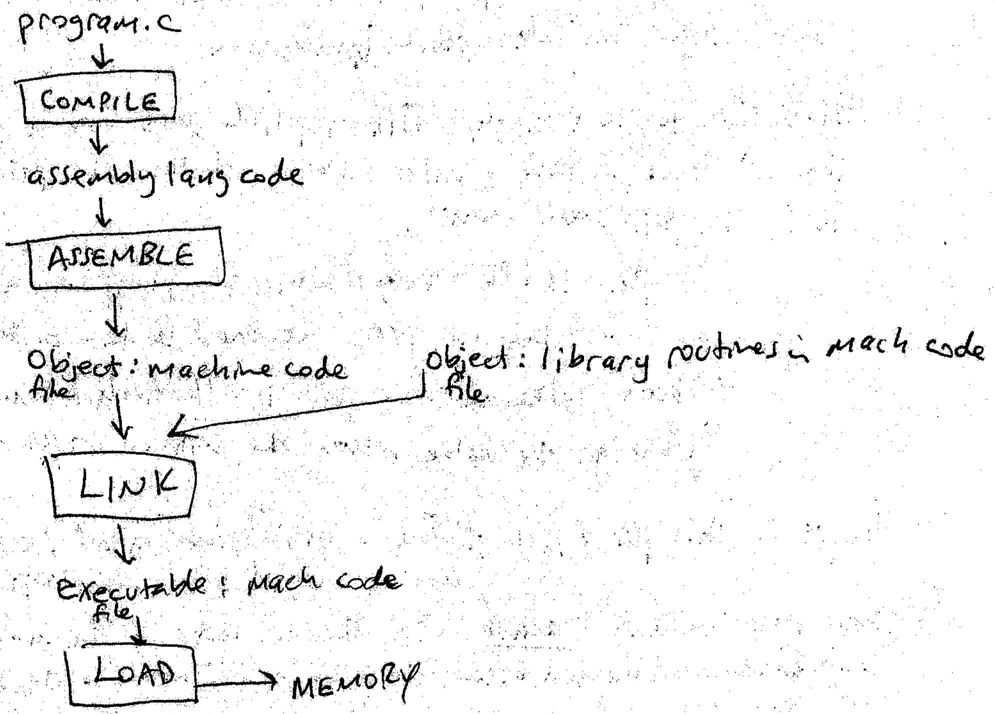
```
try: add $tØ, $zero, $s4    # copy exchange value
     ll  $t1, Ø($s1)        # load linked
     sc  $tØ, Ø($s1)        # store conditional
     beq $tØ, $zero, try    # branch store fails
     add $s4, $zero, $t1    # put load value in $s4
```

* An advantage of the load linked/store conditional mechanism is that it can be used to build other sync primitives, e.g.,
  ATOMIC COMPARE & SWAP,
  ATOMIC FETCH-AND-INCREMENT } these, & more, require more code between ll & sc.

2.12 Translating & Starting a Program

4 steps in transforming a C program in a file on a disk into a program running on a computer:

(a) compile
(b) assemble
(c) link
(d) load


program.c
COMPILE
assembly lang code
ASSEMBLE
Object: machine code file
object: library routines in mach code file
LINK
executable: mach code file
LOAD
MEMORY

$\boxed{\text{ASSEMBLY}}$

Assembly lang for any processor $\Rightarrow$ have pseudoinstructions

↓ common variation of the processor's true assembly lang instr often treated as if it were an instr in its own right

} give a richer set of asm instrs than those implemented in HW.

⇓ ONLY cost is reserving $at reg for use by assemble.

(ex) move $t0, $t1

⇓ pseudoinstr for

add $t0, $zero, $t1

(ex) blt $\xrightarrow[\text{pseudo}]{}$ slt and bne / as well as bgt, bge, ble

(ex) converts branches to faraway locations into branch & jump

(ex) allows 32-bit constants to be loaded into a register despite the 16-bit limit of the immediate instrs.

* MIPS assemblers use ~~hex~~ decimal

* Assembler converts assembly code into an <u>object file</u>

↳ combination of:
- machine lang instr
- data
- info needed to place instr properly in memory

* To produce the binary version of each instr, the assembler must determine the addrs corresponding to all labels

* Assemblers keep track of labels used in branches & data transfer instrs in a <u>SYMBOL TABLE</u> ⟶ contains pairs of symbols & addrs

(ex) an object file for a UNIX system contains 6 distinct pieces:

- <u>object file header</u> : describes the size & position of the other pieces of the obj file
- <u>text segment</u> : contains machine lang code
- <u>static data segment</u> : contains data allocated for the life of the program
  - can be static data — allocated throughout the program and dynamic data — grow & shrink as needed.
- <u>relocation info</u> : id's instrs & data words that depend on absolute addrs when the prog is loaded into memory

— <u>symbol table</u> – contains the remaining labels that are not defined, such as external references

— <u>debugging info</u> – contains a concise descr of how the modules were compiled so that a debugger can associate machine instrs with C source files & make data structures readable.

LINKER

Instead of recompiling & re-assembling the whole program, libraries too, is a piece has changed, one compiles & assembles each procedure independently, so that a change in one file does not require a full compile & assembly.

↳ this requires a <u>LINK EDITOR</u> or <u>LINKER</u>

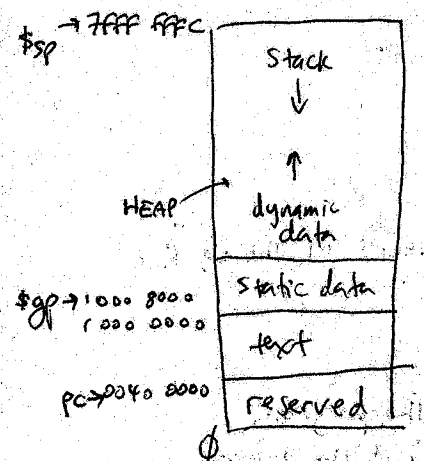
↳ takes all independently assembled machine lang programs & "Stiches" them together:

<u>LINKER STEPS</u>

(1) Place code & data modules symbolically in memory
(2) Determine the addrs of data & instr labels
(3) Patch both the internal & external references

* linker uses the relocation info & symbol table in each object module to resolve all undefined labels.

* if all external references are resolved, the linker next determines the memory locations each module will occupy.



MIPS memory alloc for prog & data

- these addrs are only sw convention, not part of MIPS arch.
- $sp initialized to 7fff fffc & grows downward toward data segment
- @ other end, the program code ("text") starts @ 0x0040 0000
- the static data starts @ 0x1000 0000
- dynamic data, allocated by malloc - C, & new in Java, is next, growing up toward the stack in the area called the HEAP
- $gp is set to an addr to make it easy to access data; it's initialized to 0x1000 8000 so that it can access from 0x1000 0000 to 0x1000 ffff using the (+) & (−) 16-bit offsets from $gp

* Since the files were assembled in isolation, the assembler could not know where a module's instrs & data would be placed relative to other modules.

* When the linker places a module in memory, all ABSOLUTE references, that is, memory addrs that are not relative to a register, must be relocated to reflect its true location.

- The linker produces an <u>EXECUTABLE FILE</u> that can be run on a computer.
  ↳ same file format as an object file except it contains no unresolved references.

$\boxed{\text{LOADER}}$

- After the executable file is on disk, the user prompts the OS to "run" the program
- The OS reads the program into memory & starts it
- In UNIX, the loader follows these steps:

(1) Reads the executable file header to determine the size of the text & data segments
(2) Creates an address space large enough for the text & data
(3) Copies the instr & data from the ~~exec~~ file into memory
(4) Copies parameters (if any) to the main prog onto the stack
(5) Initializes the machine registers & sets the $sp to the first free location
(6) Jumps to a start-up routine that copies the parameters into the argument registers & calls the main routine of the program. When the main routine returns, the start-up routine terminates the program with an <u>exit</u> system call.

<u>Dynamically Linked Libraries</u>

The first part of this section describes traditional approach to linking libraries before the program runs. Although this static approach is the fastest way to call library routines, there are a few disadvantages:

- the library routines become part of the ~~executable~~ code.
  The statically linked program uses original library code, no updates.

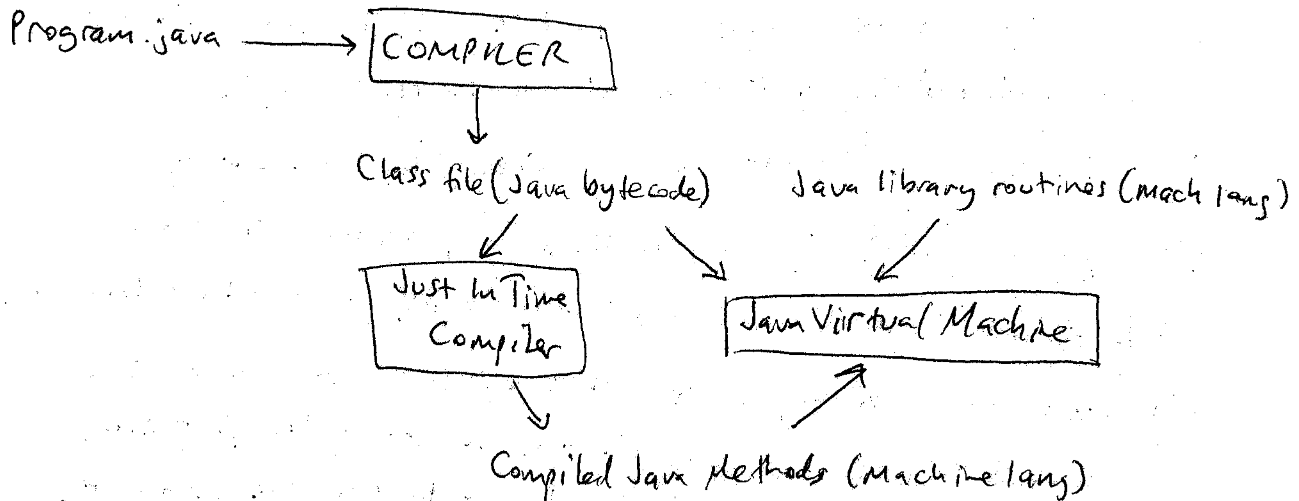
- It loads all routines in the library that are called anywhere in the exec, even if those calls ~~are~~ are not executed. The library can be large relative to the program; for example, the standard C lib is 2.5 MB.

⇓ To overcome these disadvantages, use dynamically linked libraries (DLLs)

- DLLs ≡ not linked & loaded until program is run.
  - lazy procedure linkage ⇒ each routine is linked only AFTER it is called
  - require extra space for the info needed for dynamic linking, but do not require the whole libraries ~~be~~ copied or linked.
  - they pay a good deal of overhead the first time a routine is ~~called~~, but only a single indirect jump thereafter.

Starting a Java Program

Translation hierarchy for Java



Program.java
COMPILER
Class file (Java bytecode)
Java library routines (mach lang)
Just In Time Compiler
Java Virtual Machine
Compiled Java Methods (Machine lang)

- upside for interpretation is ~~portability~~.
- downside of interpretation is performance → factor of 10 slowdown when compared to compiled C
- JIT compiler ⇒ preserve portability & improve execution speed
  → translate~~code~~ while program is running
  → profile the running program to find where the "hot" methods are and then compile them into native instr set
  → compiled portion is saved for ~~the~~ next time the program is run.

2.13 - A C Sort Example to Put It All Together

C-lang swap procedure

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

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

WHEN TRANSLATING FROM C to assembly lang, follow these general steps:

STEPS:
1 - allocate registers to program variables } MIPS convention on param passing - use $a{0,3}
2 - Produce code for the body of the procedure
3 - preserve regs across procedure invocation

Example steps

STEP 1: Reg alloc:
params: v, k ⇒ v → $a0 ; k → $a1
variables: temp ⇒ temp → $t0 since swap() is a leaf procedure (one that does not call other procedures)

STEP 2: Procedure call:

$\boxed{\text{NOTE}}$ ⇒ mem addr for MIPS refer to BYTE addr ∴ words are 4 bytes apart

∴ Multiply index k by 4 before adding it to addr > COMMON MISTAKE IN ASM PROGRAMMING ⇒ forgetting that sequential word addrs differ by 4 instead of 1

```
sll $t1, $a1, 2    # reg $t1 = k*4
add $t1, $a0, $t1  # reg $t1 = v + (k*4) → $t1 has addr of v[k]
lw  $t0, 0($t1)    # load v[k] using $t1 : reg $t0 (temp) = v[k]
lw  $t2, 4($t1)    # load v[k+1] by adding 4 to $t1 : reg $t2 = v[k+1]
                                                   ↳ refers to next element of v
sw  $t2, 0($t1)    # store $t2 to swapped addrs : v[k] = reg $t2
sw  $t0, 4($t1)    # store $t0 to swapped addr : v[k+1] = reg $t0 (temp)
```

STEP 3: Preserve
Since we're not using saved reg ~~in~~ this leaf procedure, nothing to preserve.

NEXT: build a routine that calls the swap procedure, demonstrating complexity of asm programming

```
void sort (int v[], int n)
{   int i, j;
    for (i=0; i<n; i+=1)
    {
      for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1)
      {
         swap(v, j);
      }
    }
}
```

STEP 1 : Reg alloc ⟶ v ⇒ \$a0 ; n ⇒ \$a1 ; i ⇒ \$s0 ; j ⇒ \$s1

don't forget to preserve

STEP 2: Procedure Code

2 nested for Loops & a call to swap, which includes params

has 3 parts : initialization, loop test, & iteration increment

FIRST LOOP :

for (i=0; i<n ; i+=1) ⇒

addi \$s0, \$s0, 1 # i+=1

pseudo instr of MIPS assembler ⟶ move \$s0, \$zero # i = 0

```
for1tst: slt $t0, $s0, $a1 # $t0 = 0 if $s0 >= $a1 (i>=n)
         beq $t0, $zero, exit1 # go to exit1 if $s0 >= $a1 (i>=n)
         : body of 1st loop.
         j for1tst # jump to test or the loop
exit1 :
```

SECOND LOOP :

for (j = i-1 ; j>=0 && v[j] > v[j+1]; j -= 1)

addi \$s1, \$s0, -1 # j = i-1

addi \$s1, \$s1, -1 # j = j-1

loop test has 2 parts (j>=0) && (v[j] > v[j+1])

exit if either fail ∴ first test must exit loop if it fails (j<0)

∴

```
for2tst : slti $t0, $s1, 0 # $t0 = 1 if $s1 < 0 (j<0)
          bne $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j<0)
```

this branch skips over 2nd condition test. If it doesn't skip, then j ≥ 0

∴ test v[j] > v[j+1] EXITS that is NOT TRUE OR $\underbrace{v[j]}_{\text{addr is } \$s1} <= \underbrace{v[j+1]}_{\text{addr is } \$t1 \text{ which must be } \$s1*4}$

or shift left 2 bits

```
sll $t1, $s1, 2     # $t1 = j * 4
add $t2, $a0, $t1   # $t2 = v + (j * 4)
```

now Load v[j]

```
lw $t3, 0($t2)   # $t3 = v[j]
```

Since we know the next element is just the following word, add 4 to addr in $t2 to get v[j+1]

```
lw $t4, 4($t2)   # $t4 = v[j+1]
```

Test of v[j] ≤ v[j+1] SAME AS v[j+1] ≥ v[j]

∴
```
slt $t0, $t4, $t3   # $t0 = 0 if $t4 ≥ $t3
beq $t0, $zero, exit2   # go to exit2 if $t4 ≥ $t3
```

Bottom of the 2nd loop jumps back to inner loop test

∴
```
j  for2tst  # jump to test of inner (2nd) loop
```

⟨Procedure call⟩

- Next step is body of the 2nd for loop ⇒ swap(v, j)
- calling swap is easy: jal swap

passing params from sort() to swap()

sort() needs values in regs $a0 & $a1 YET swap() needs to have its params placed in $a0, $a1

∴ one soln is to copy the parameters for sort into other regs earlier in procedure, making $a0 and $a1 available for call to swap() ⟹ this copy FASTER than saving & restoring on the stack!

∴
```
move $s2, $a0   # copy param $a0 to $s2
move $s3, $a1   # copy param $a1 to $s3
```

Then we pass the params to swap()

```
move $a0, $s2   # first swap() param is v
move $a1, $s1   # second swap() param is j
```

STEP 3:

Preserving regs

- must save the return addr in $ra, since sort() is a procedure & is called itself
- sort() also uses saved regs $s{0, 1, 2, 3} → so they must be saved

∴
```
addi $sp, $sp, -20   # make room on stack for 5 regs
sw $ra, 16($sp)      # save $ra on stack
sw $s3, 12($sp)      # save $s3 on stack
sw $s2, 8($sp)       # save $s2 on stack
sw $s1, 4($sp)       # save $s1 on stack
sw $s0, 0($sp)       # save $s0 on stack
```

THEN the end of sort() reverses these, then adds jr to return.

Full MIPS version in asm of C-lang sort() → see p155 or next page.

2.14 Arrays vs Pointers

- understanding pointers → challenge in C
- comparing asm code that uses arrays & array indices to the asm code using pointers
- use example procedure to clear a sequence of words in memory ⇉ one using array indices
  - one using pointers

Array version of clear()

params ⇒ array → $a0
size → $a1
i → $t0

```
        move $t0, $zero       # i=0
loop1:  sll  $t1, $t0, 2      # $t1 = i*4
        add  $t2, $a0, $t1    # $t2 = addr of array[i]
        sw   $zero, 0($t2)    # array[i] = 0
        addi $t0, $t0, 1      # i+=1
        slt  $t3, $t0, $a1    # $t3 = (i<size)
        bne  $t3, $zero, loop1  # if (i<size) go to loop1
```

* this works as long as size > 0.
ANSI C requires test of size before loop

C-lang

```
clear1 (int array[], int size)
{
  int i;
  for (i=0; i < size; i+=1)
    array[i] = 0;
}
```

```
clear2(int *array, int size)
{
  int *p;
  for (p = &array[0];
       p < &array[size];
       p = p+1)
    *p = 0;
}
```

Pointer version of clear()

params ⇒ array → $a0
size → $a1
p → $t0

```
        move $t0, $a0         # p = addr of array[0]
loop2:  sw   $zero, 0($t0)    # memory[p] = 0
        addi $t0, $t0, 4      # p=p+4 (increment ptr by 1, p to point to next word b/c p is a ptr to an integer
                                                                                      ↓
                                                                                  4 bytes long.
        sll  $t1, $a1, 2      # $t1 = size x4
        add  $t2, $a0, $t1    # $t2 = addr of array[size]
        slt  $t3, $t0, $t2    # $t3 = (p < &array[size])
        bne  $t3, $zero, loop2  # go to loop2 if (p < &array[size])
```

can move outside loop b/c addr of end of array does not change.

simplification of pointer version:

```
        move $t0, $a0        # p = addr of array[0]
        sll $t1, $a1, 2      # $t1 = size * 4
        add $t2, $a0, $t1    # $t2 = addr of array[size]
loop2:  sw $zero, 0($t0)     # Memory[p] = 0
        addi $t0, $t0, 4     # p = p + 4
        slt $t3, $t0, $t2    # $t3 = (p < &array[size])
        bne $t3, $zero, loop2 # if (p < &array[size]) go to loop 2
```

* Comparing the two versions
  - with arrays, multiply & add are inside the loop b/c i is incremented & each address must be recalculated from the new index
  - memory pointer version increments the pointer p directly & moves them outside loop, reducing instrs executed per iteration to 4 from 6

  → ** this manual optimization ≡ compiler optimization of <u>strength reduction</u> (shift instead of multiply) & of induction variable elimination (rid of array addr calc w/i loops)

Chap 3 - Arithmetic for Computers

3.2 Addition & Subtraction

* when adding operands w/ different signs, overflow cannot occur (b/c sum can't be > than one of operand
* when subtracting operands w/ SAME signs, overflow cannot occur ∵ $x - y = x + (-y)$
* lack of 33rd bit ⇒ so when overflow does occur the sign bit is set with the <u>value</u> of the result INSTEAD of the proper sign of the result.

Signed #'s

⇒ since we need just one extra bit, only the sign can be wrong

∴ overflow occurs when adding two positive numbers & the sum is negative, or vice-versa ⇒ this means a <u>carry out</u> has occurred into the sign bit

∴ overflow occurs in subtraction when subtract a (−) # from a (+) # & get a (−) result, OR when subtract a (+) # from a (−) # & get a (+) result ⇒ this means a <u>borrow occurred</u> from sign bit

*** what about OVERFLOW from unsigned #'s (integers)?
↳ commonly used for memory addr & overflow ignored

∴ Computer designer ⇒ recognize 2 choices

- add, addi, sub ⇒ cause exceptions on overflow
- addu, addiu, subu ⇒ do <u>not</u> cause exceptions on overflow.

<u>Overflow conditions for (+) and (−)</u>

operation	Operand A	Operand B	Result indicating overflow
A+B	$\geq \emptyset$	$\geq \emptyset$	$< \emptyset$
A+B	$< \emptyset$	$< \emptyset$	$\geq \emptyset$
A−B	$\geq \emptyset$	$< \emptyset$	$< \emptyset$
A−B	$< \emptyset$	$\geq \emptyset$	$\geq \emptyset$

ALU : arithmetic logic unit ⇒ HW performs add/subtract

<u>3.3 Multiplication</u>

(ex)

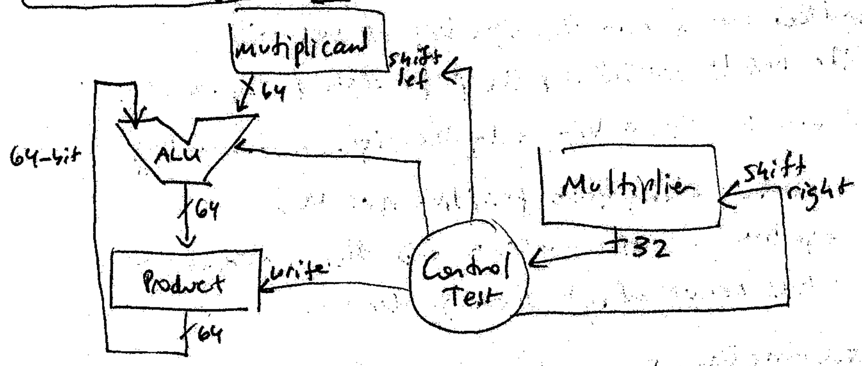
```
Multiplicand:    1000₁₀
Multiplier:    × 1001₁₀
               --------
                   1000
                  0000
                 0000
                1000
               --------
Product:       1,001,000₁₀
```

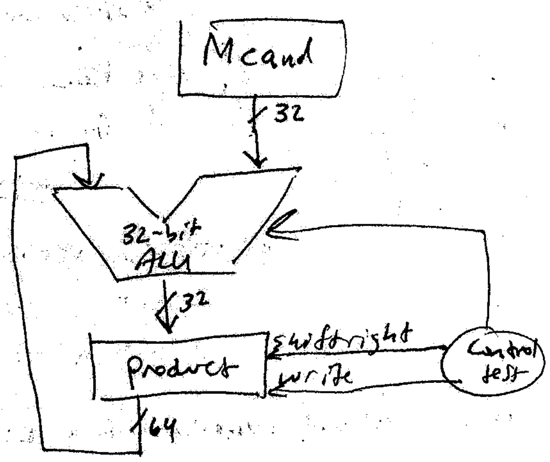
ignoring the sign bit, the length of mult of n-bit multiplicand and an m-bit multiplier results in a product (n+m) bits long.

See p 231

1st version of mult. HW → Refined version of mult. HW

perform ops in parallel multiplier & multiplicand shifted while multiplicand added to product if multiplier bit is 1.





Chap 4 - The Processor

- ch1 - explains perf of computer is determined by 3 key factors
  - instr count
  - clock cycle time
  - ~~clocks~~ cycles per instr (CPI)
- ch2 - explains that the compiler & the instruction set arch (ISA) determine the instr count required for a given program

  However, implementation of the processor determines both the clock cycle time & the # of clock cycles per instr.
- ch3 - describes arithmetic for computers : addition
  - subtraction
  - multiplication
  - division
  - floating point ~~#~~ representation
- ch4 - explains & constructs the datapath & control unit for two different impls of the MIPS ISA.
  - explains the principles & techniques used in implementing a processor, starting w/ a highly abstract & simplied overview
  - next, builds up a datapath & constructs a simple version of a processor sufficient to implement an instr set like MIPS
  - then, covers a more realistic pipelined MIPS implementation

$\boxed{\text{A Basic MIPS implementation}}$ key principles used in creating a <u>datapath</u> & designing the <u>control</u>

- lw and sw ⟸ <u>memory-reference instructions</u>

  load word   store word
- add, sub, AND, OR, slt ⟸ arithmetic - logical instructions

  set less than
- beq and j

  branch equal   jump

* does not include rest integer instrs NOR any floating point

- choice of ISA determines many aspects of the impl
- choice of various impl strategies effects clock rate & CPI

Review design principles : ① simplicity favors regularity
② smaller is faster
③ make the common case fast
④ good design demands good compromise

Overview of the Implementation

— for every instr, the first 2 steps are identical

① Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.

② Read one or two registers, using fields of the instruction to select the registers to read. For the Load word instr, we need to read only one register, but most other instrs require that we read two regs.

* After these 2 steps, actions required to complete the instr depend on the instr class : (a) memory-reference
(b) arithmetic-logical
(c) branches

for MIPS ISA, simplicity + regularity simplifies impl by making the execution of many of the instr classes similar. <RISC>

ex _ all instr classes (except jump) use the ALU after reading the regs.
→ mem-ref instrs use ALU for addr calc
→ arith-Log instrs use ALU for op execution
→ branches instrs use ALU for comparison

After using the ALU, actions required to complete various instr classes differ.

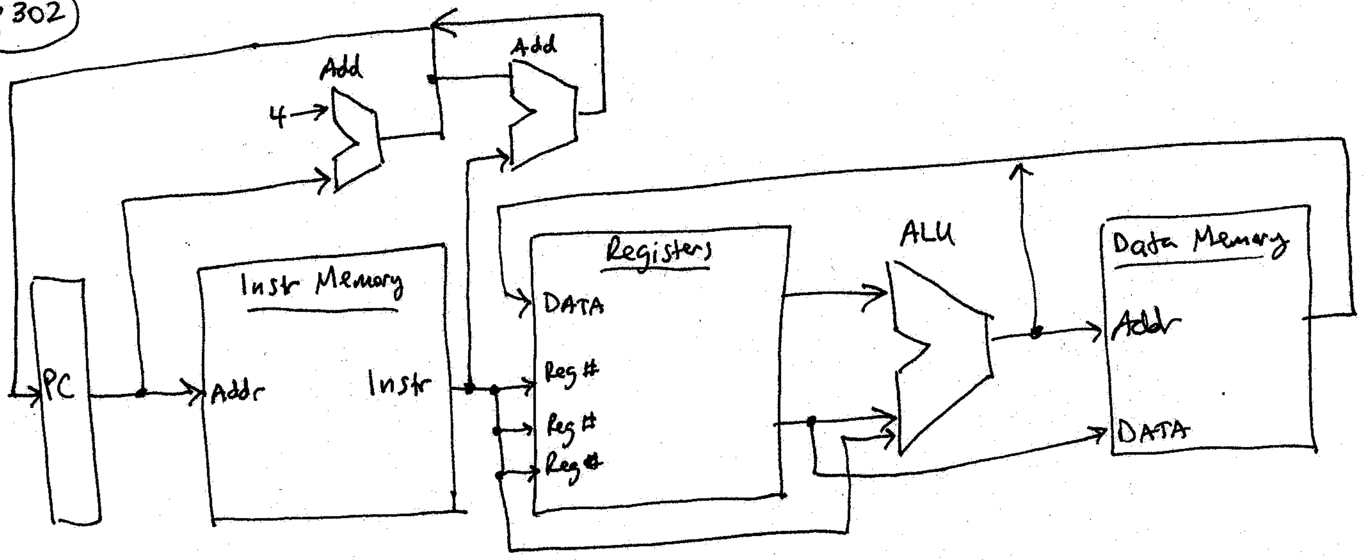
→ mem-ref instr ⇒ access memory to either read data for a Load or write data for a store

→ arith-Log instr or load instr ⇒ write data from ALU or memory back into a reg.

→ branch instr ⇒ may need to change the next instr addr based on the comparison.
⇒ otherwise, PC incremented by 4 to get addr of next instr.

Abstract View of the Impl of MIPS subset : major functional units 4 major connections

(302)



- All instructions ~~start~~ by using the PC to supply the instr addr to instr memory
- After the instr is fetched, the register operands used by an instr are specified by fields of that instr.
- Once reg operands have been fetched, they can be operated on to compute a memory addr (for a load or store), to compute an arithmetic result (for an integer arith-log instr), or a compare (for a branch)
- If the instr is an arith-log instr, the result from the ALU must be written to a reg.
- If the operation is a load/store, the ALU result is used as an addr to either store a value from the registers or load a value from memory into the regs.
- The result from the ALU or memory is written back into the reg file.
- Branches require the use of the ALU output to determine the next instr addr, which comes either from the ALU (where the PC and branch offset are summed) or from an addr that increments the current PC by 4
- The thick lines interconnecting the functional units represent buses, which consist of multiple signals.
- The arrows used to guide the reader in knowing how information flows
- Since signal lines may cross, explicitly show when crossing lines are connected by the presence of a dot where lines actually cross.

NOTE: every instruction begins execution on one clock edge and completes execution on ~~the~~ next clock edge.

- data going to a perdicular unit as coming from 2 differnt units/sources
  - ∴ since they are signal lines, a CONTROL UNIT needs to select which source goes to which unit at the junction ⇒ USE MULTIPLEXOR or data selector.
- also, several of the units must be controlled depending on the TYPE of instr.
  - (ex). data memory must read on a load OR write on a store
    - · reg file must be written on a load and on an arith-log instr.
    - · ALU must perform one of many ops
  - ∴ like the multiplexors, these ops are directed by a CONTROL UNIT

Control Unit $\boxed{\text{SEE pg 304 - Fig 4.2}}$

- has input : instr ⇒ used to determine how to set the control lines for the functional units and 2 of the multiplexors

* the 3rd multiplexor ⇒ determines whether PC+4 OR branch dest addr is written into the PC
  ⇒ is set based upon the ZERO output of the ALU ↓ used to perform the comparison of the BEQ instr.

* Regularity & simplicity of MIPS ISA means that a simple decoding process can be used to determine how to set the control lines

* NOTE: in this first ISA design, every instr begins exec on one clk edge & completes on the next clk edge.

- Remainder of this chapter ⇒ fill in details of MIPS basic ISA
  - add more functional units
  - increase # of conns btwn units
  - enhance CONTROL UNIT to control what actions are taken for different instr classes
- Sections 4.3 & 4.4 ⇒ describe simple impl that uses a SINGLE LONG CLOCK CYCLE for every instr
- * THIS APPROACH IS NOT practical SINCE the clock cycle must be stretched to accomodate the longest instruction ⇒ SOLN → pipelining (section 4.5)

Section 4.2 - Logic Design Conventions

- design of a computer → must decide how logic implementing the computer will operate
  AND
  - how the computer is clocked.

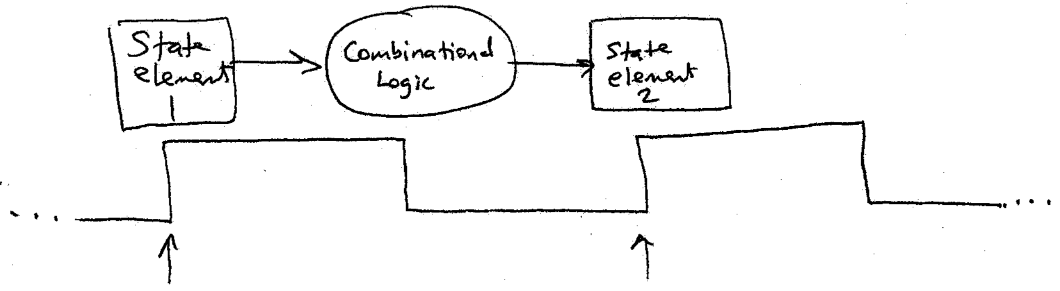
- datapath elements in MIPS consists of 2 types of logic:
  - combinational logic ⇒ data values → depend upon only the inputs
  - state/sequential logic ⇒ state values → depend upon state element (mem)
    ↳ depend on inputs & internal state.

- MIPS ISA uses 3 types of state elements:
  - D-type FF (value + clk)
  - registers
  - memories

* NOTE: assertion → logically high / true
deassertion → logically low / false

Clocking Methodology - approach used to determine when data is valid and stable relative to the clock
- when signals can be written or read.
- ensures predictability

* For simplicity, assume "edge-triggered" clocking methodology
↳ all state changes occur on a clock-edge (+) or (-)



Control signal: a signal used for a multiplexor selection OR for directing the operation of a functional unit

data signal: a signal containing information that is operated on by a functional unit.

* An edge-triggered clocking methodology allows us to <u>read</u> the contents of a register, <u>send</u> the value thru some combinational Logic, AND <u>write</u> that register IN THE SAME CLOCK CYCLE.

* <u>data bus</u> : signals wider than 1 bit.

Section 4.3 Building a Data Path

Design a datapath design — start by examining major components required to execute each class of MIPS inst.

* datapath element ≡ a unit used to operate on ~~OR~~ hold data within a processor
  * IN MIPS ISA ⇒ datapath elements include : instr
    - data memories
    - reg file
    - ALU
    - adders

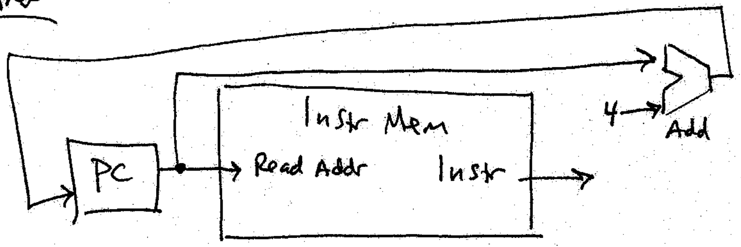
  ≡ also have control signals

— look at datapath elements each MIPS instr needs:

* memory unit — to store instrs of a program & supply instrs at a given addr
* PC — a reg that holds addr of the current instr
* adder — to increment PC to addr of next instr

  ⟶ hardwired "add" of ALU (combo Logic)

Putting it together



A portion of the datapath used for fetching instrs & incrementing PC

* To execute any instr, we must start by fetching the instr from mem.
* To prepare for executing next instr, we must also increment PC so that it points at next instr, 4 bytes later.

R-format instrs :

6 bits	5bits	5bits	5bits	5bits	6bits
op	rs	rt	rd	shamt	funct

- all read 2 regs
- perform an ALU op on regs
- write result to a reg

R-type or arith-log } instrs. add sub AND OR slt

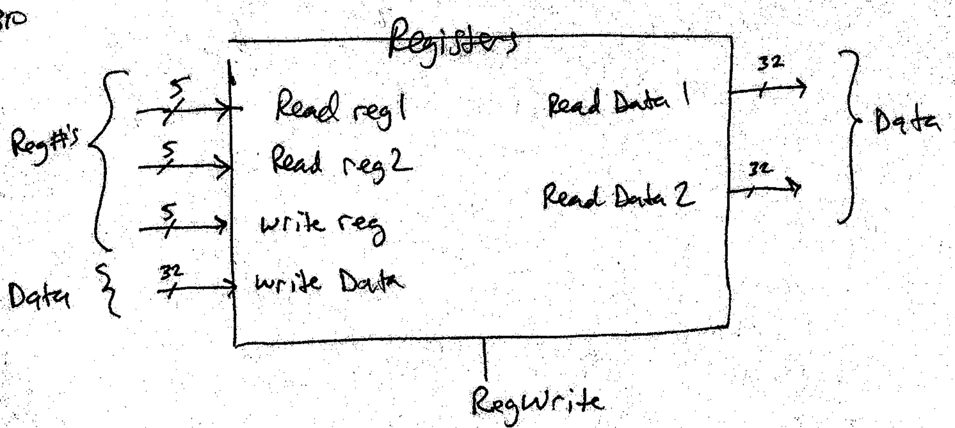
* Register file ≡ collection of regs ~~in~~ which any reg can be read or written by specifying the # of the reg in the file
≡ Contains the reg state of the computer
* need an ALU to operate on the values read from the regs.

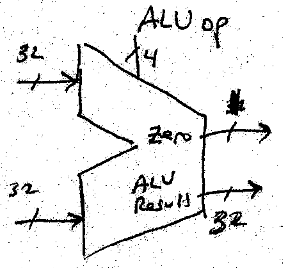
32 bits long
* addr in instr for reg is 5 bits
∴ $2^5 = 32$ possible regs

$\boxed{\text{reg file}}$: a STATE ~~E~~ELEMENT consisting of a set of regs that can be read or written by supplying a reg # to be accessed.

* reg file always outputs the contents of whatever reg #s are on the Read reg inputs.
⇒ Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge

p 310





* Sign Extend ≡ to increase

4.5 - Overview of Pipelining

* pipelining : an implementation technique in which multiple instructions are overlapped in execution.
  - nearly universal today.

Analogy : "laundry activities"

Non-pipelined approach

1. Place one dirty load of clothes in the washer
2. When the washer is finished, place the wet load in the dryer.
3. When dryer finished, place the dry Load on table & fold.
4. When folding done, put clothes away (another person), & when done, start over with next load.

pipelined approach
- takes much less time → ∴ more efficient, less idling.

* as soon a washer finished with 1st load & placed in dryer, load washer with 2nd load
* when 1st load dry, place it on table to start folding, move wet Load to dryer, & next dirty Load into washer
* Next have another person put first completed load away, you start folding 2nd Load, dryer has 3rd Load, & you put 4th Load into washer.
* At this point, all steps (stages in pipelining) are operating CONCURRENTLY

NOTE
As long as we have seperate resources for each stage, we can pipeline the tasks.

* Pipelining improves throughput
  BUT time taken per step/stage remains fixed & is NOT shorter.
  (paradox)
* reason pipelining is faster for many loads ⇒ all working in parallel, so more loads are finished per hour.
* Pipeline speed-up : approaches 1/(# stages)
  But is NOT exact due to start-up & wind-down!

5 stage-

Pipelining instructions execution

1. Fetch instruction from memory
2. Read registers while decoding the instruction
   - The regular format of MIPS instrs allows reading & decoding to occur simultaneously.
3. Execute the operation OR calculate address
4. Access an operand in data memory.
5. Write the result into a register.

~~***~~ Pipeline stage times of a computer are limited by slowest resource (ALU op or mem access)

(ex) Single-cycle vs. Pipelined Performance

** Assumptions - single-cycle impl : - all instrs take 1 clk cycle
∴ clk cycle stretched to accomodate slowest instr.
- op times : - 200ps - mem access
200ps - ALU op
100ps - reg read/write
- write to reg file occurs in 1st half of clock cycle ; read in 2nd half.

Instr class	Instr fetch	Reg read	ALU op	Data Access	Reg write	Total Time
lw	200ps	100ps	200ps	200ps	100ps	800ps ← slowest
sw	200ps	100ps	200ps	200ps	–	700ps
R-format (add, sub, AND, OR, slt)	200ps	100ps	200ps	–	100ps	600ps
Branch (beq)	200ps	100ps	200ps	–	–	500ps

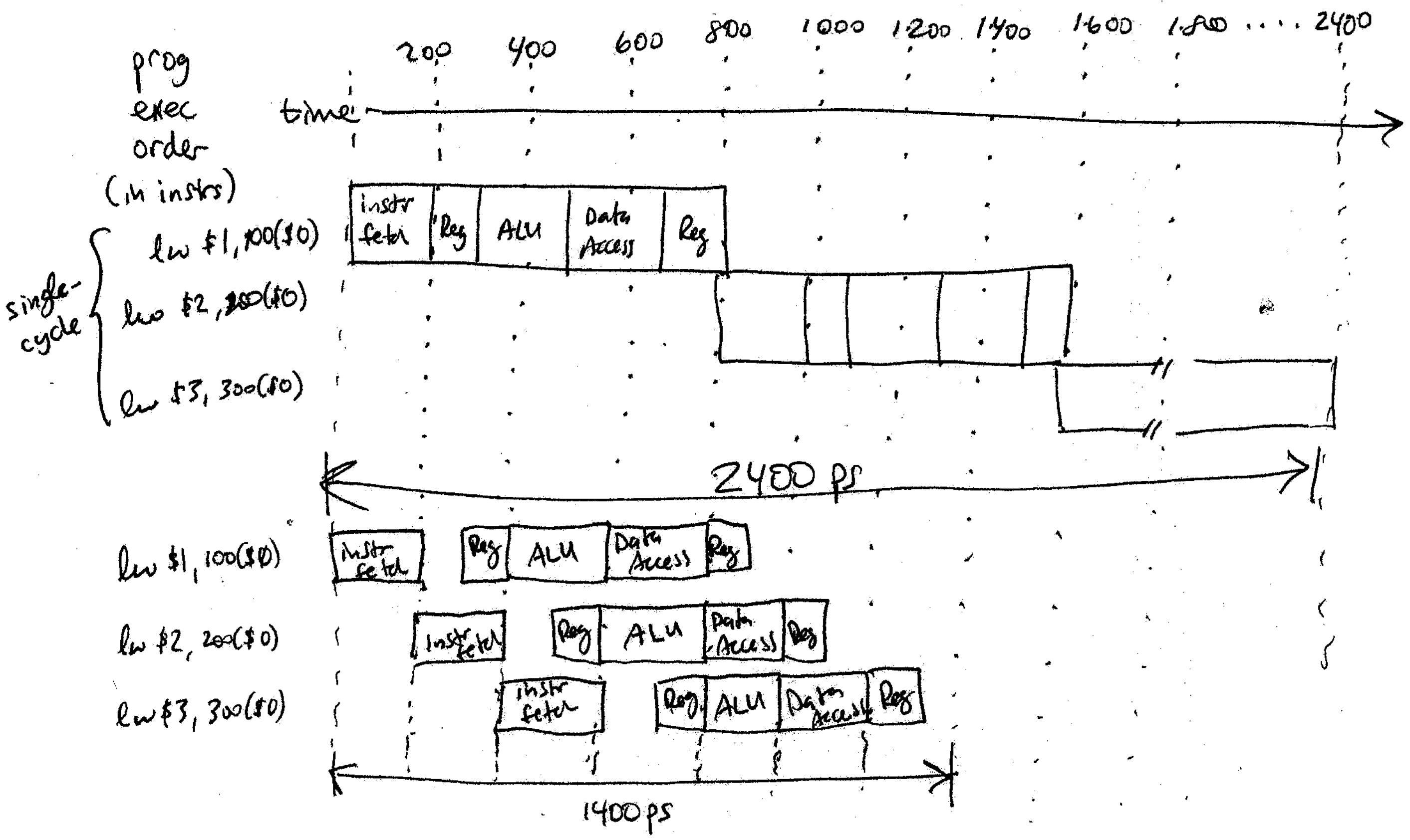
∴ so time required for every instr becomes 800ps.

now for example, take 3 lw instrs: for single-cycle vs pipeline:

lw \$1, 100(\$0)

lw \$2, 200(\$0)

lw \$3, 300(\$0)



~~pipeline~~ time btwn 1st & 4th instr = 3 × 200 ps = 600ps
single-cycle time btwn " " " " " = 3 × 800 ps = 2400ps } 4x performance improvement.

b/c clock cycle of single-cycle = 800ps & pipeline = 200 ps } 4x.

NOTE As # instr >> 1, then speedup approaches 4 : right now w/ 3 instrs 2400/1400 < 4
(ex) add 1M instrs : 1M × 200 ps = 200M
1M × 800 ps = 800M ⇒ 4x !

Pipeline Speed-up Formula

* if stages perfectly balanced, then time btwn instrs on the pipelined processor (assuming ideal conditions):

$$\text{Time btwn instr (pipelined)} = \frac{\text{Time btwn instrs (non-pipelined)}}{\text{\# pipe stages}}$$

* under ideal conditions & w/ a large # of instrs, speed-up ≅ # pipe stages
so w/ 5-stages | Instr fetch | Reg | ALU | Data Access | Reg | ⇒ we should have $\frac{800 \text{ ps}}{5} = 160$ ps clock cycle, BUT as we see in our diagram, the stages are not balanced.
Also, pipelining involves overhead.
∴ time/instr (pipelined) > min possible & speedup < # of pipeline stages

*pipelining improves performance by

INCREASING INSTR THROUGHPUT

as opposed to decreasing execution time of an individual instr.

Designing Instr Sets for Pipelining

(a) - all instr same length → easier to fetch in 1st stage & to decode in 2nd stage
↓ in MIPS true; but in x86 not true ⇒ instr length vary 1-17 bytes

(b) few instr formats
→ source reg fields located in same place in each instr. ⇒ symmetry.
∴ 2nd stage can begin reading the reg file at the same time the HW determines what type of instr was fetched.
⇒ if NOT true ⇒ need to split 2nd stage, resulting in 6 pipeline stages (downside)

(c) Memory operands only appear in loads & stores
∴ use exec stage to calc an addr & then access memory in following stage

(d) operands aligned in memory
∴ don't need to be concerned with a single data transfer instruction requiring 2 data memory addrs ⇒ requested data can be transferred btwn proc & memory in a single pipeline stage.

(e) writes at most one result
& does this in the last stage of the pipeline

Pipeline hazards

hazard ≡ a situation in pipelining when the next instr cannot execute in the next, following clock cycle

3 types of hazards
- structural hazard
- data hazard
- Control hazard

Structural Hazard

- HW cannot support combo of instr in same clock cycle

(ex) in MIPS w/o two memories
(ex) in laundry → have a washer-dryer or roommate busy to fold clothes.

Data Hazard

- when pipeline must be stalled b/c one step must wait for another to complete.
- when pipelined planned instruction cannot execute in the proper clock cycle b/c the data that is needed to execute the instrs is not yet available.
  - arises from dependence of one instr on an earlier one that is still in the pipeline

(ex)
```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

the add instr doesn't write its result until the 5th stage
∴ need to wait/waste 3 clock cycles in pipeline

One soln ⇒ have compiler id & remove all such hazards → and not too satisfactory
* these dependencies happen too often, delay too long

Primary soln : "forwarding" or "bypassing"
- observe that we don't have to wait for the instr to complete before trying to resolve the hazard
- for above code, as soon as ALU creates the sum, supply the sum as the input for the sub

"forwarding" or "bypassing" ⇒ extra HW to retrieve missing item early from the internal resources } Cannot prevent ALL pipeline stalls ↓ data hazards

see (ex) on pg 336 : Fig 4.28 & Fig 4.29

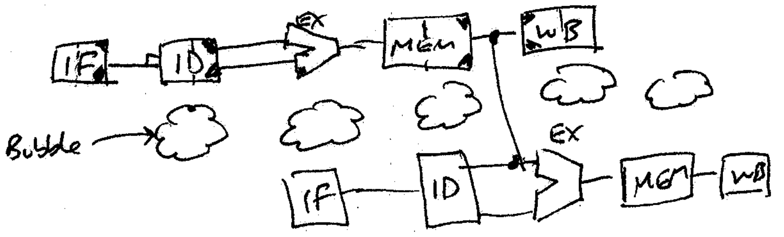
* load-use data hazard

- a specific form of data hazard in which the data being loaded by a load instr has not yet become available when it is needed by the proc by another instr.

(ex) Fig 4.30

lw \$SØ, 20Ø(\$t1)

sub \$t2, \$SØ, \$t3



bubble ≡ pipeline stall

- a stall initiated in order to resolve a hazard

SOLN: HW detection & stalls OR SW that reorders code to avoid load-use pipeline stalls

see (ex) on p 338-339

Control Hazard (branch hazard)

- when the proper instr cannot execute in the proper clock cycle for the pipeline b/c the fetch instr is NOT the one that is needed; that is, the flow of instruction addrs is NOT what the pipeline expected.

- arises from the need to make a decision based on the results of one instruction while others are executing

"branch prediction" ≡ method of resolving a branch hazard that
- assumes a given outcome for the branch & proceeds from that assumption rather than waiting to ascertain the actual outcome.

* most likely in a loop, you return to "start"

* when the guess is wrong, the pipeline control must ensure that the instructions following the wrongly guessed branch have no effect & must restart the pipeline from the proper branch addr

* in case of longer pipelines (stages) → raises "cost" of mis-prediction