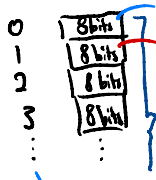
Memory Organisation (RAM)

- viewed as a large, single dimensional array of memory locations
- addr – content



data: instr, var, etc.

1 byte of data. ⇒ byte addressing. – distinct mem. location for each byte.

vs Word addressing distinct mem location for each word

2^n byte

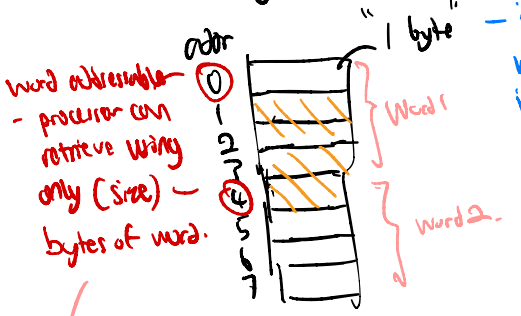
eg. 2/4/8 bytes.

Design Simplicity
- ⇒ common unit of transfer between processor and memory
- ⇒ In MIPS is 4 bytes = 32 bits.
- ⇒ Same size as register, integer and instruction size

binary no. of k-bits → 2^k mem. locations.

3-bits → 8 locations

- Word alignment



– in byte accessible mem, we are able to retrieve individual bytes in the memory

Word addressable – processor can retrieve word only (size) – bytes of word.

(4B) B – Adder – A+B (4B)
(4B) A –

problem: misalignment of word.

eg. word stored in across 2 words.

⇒ processor needs to retrieve 2 words and discard bytes appropriately. and then still need to align them properly.

⇒ use proper address of each word.

↳ check if address is divisible by 4.

↳ word size

Memory Instructions

- MIPS is a load-store registers architecture
  - 32 registers, each 32 bit long (4 byte)
  - each word contains 32 bits (4 bytes)
  - memory address are 32 bit long.
    (i.e indexes of memory.)
  - In MIPS, data must be in registers to perform arithmetic.
  - 2^{30} memory words

$\frac{2^{32} \text{ bytes}}{2^2 \text{ (size of Word)}}$

Mem[0], 0-3 bytes
Mem[4], 4-7 bytes.
⋮
Mem[4294967292]. — Memory holds data structures eg. arrays. and spilled registers such as those saved on procedure calls.

⇓ "memory at the address 4294967292"

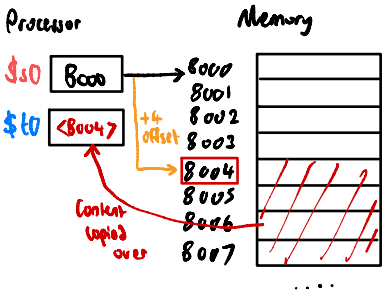
  - accessed only by data transfer instructions
  - MIPS uses byte addresses, so consecutive words differ by 4.

- Memory Instruction: Load Word.

```
lw $t0, 4($s0)
```

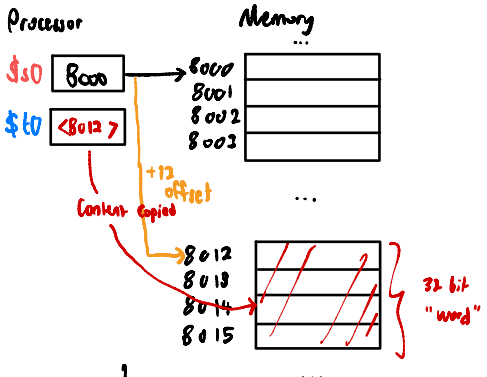
"Sum" up and give effective address in memory

$t0 — destination register (addr. then stored here)
4 — displacement or offset
$s0 — source register



- Memory Instruction: Store Word

```
sw $t0, 12($s0)
```



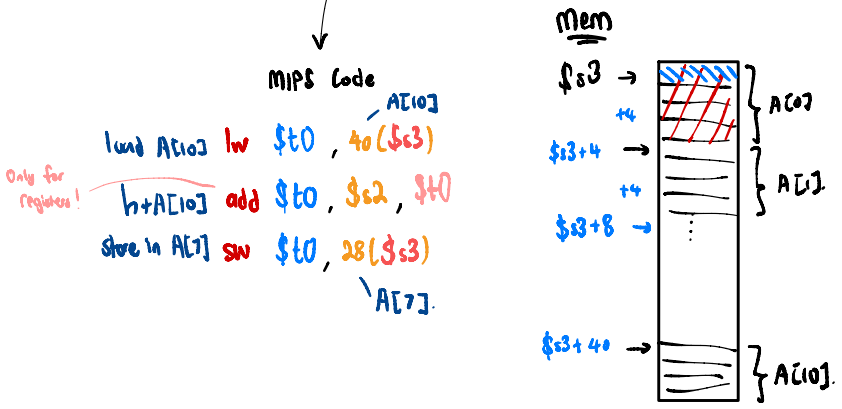
- Only load and store instructions can access data in memory.

- eg. assuming each array element occupies a word (4 bytes)

A[7] = h + A[10];

↓ mapped ↓

$s2 $s3 : contains the base address i.e. addr. of first element, A[0] of array A.



- load byte (lb) and store byte (sb)
  - same format.
  - only load/store that particular byte
  - useful for character type.
  - offset no longer needs to be multiple of 4!

- eg. $t2 can store 32 bits of data,
  if we want to store 1 byte of data,
  will store at lowest 8 bits of data in $t2.

- Memory instruction works if address is divisible by 4 (aligned word)
  - MIPS disallowed loading & storing unaligned word. (ie lw/sw)
  - pseudo-instructions unaligned load word (ulw), unaligned store word (usw)
    - translated into some other instructions.
    - avoid using pseudo-instr.

- Register do not have types
  - can hold any 32-bit number
  - number has no implicit data type, it is interpreted according to the instruction that uses it.
  - eg. add $t2, $t1, $t0 — 2's comp. signed int.

    lw $t2, 0($t0) — "address"

* Consecutive word addresses in machines with byte addressing do not differ by 1.

i.e. if incr. address by 1, will not point to next word, will only point to next byte.

⇒ lw & sw – sum of base address & offset must be multiple of 4 (adhere to word boundary).

- Eg.

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

Variables mapping:

k → $5

Base addr. of v[] → $4

temp → $15.

⇓ Corresponding Assembly Code

```
swap:
Computes effective addr. of v[k]
    sll  $2, $5, 2      shift k by 2 bits on LHS ⇒ k * 2^2
    add  $2, $4, $2
v[k] → $15 ←  lw  $15, 0($2)
v[k+1] → $16 ←  lw  $16, 4($2)
              sw  $16, 0($2)
              sw  $15, 4($2)
```

Making Decisions

- eg. if and goto statements.
  - goto: avoid in C.

- Alter control flow of the program.
- Change the next instruction to be executed

- 2 types:

1. Conditional (branch)

```
bne $t0, $t1, label
```

- bne: "branch if not equal"
- label: points to one of the instructions in your program.
- ⇒ if (a != b) goto L1

check both value if equal

- ↳ next instruction after bne.

else

- ↳ branch to label.

```
beq $t0, $t1, label
```

- beq: "branch if equal"
- ⇒ if (a == b) goto L1
- ↳ reverse of bne.

eg.

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

- beq: if equals.
- j: if not equals.

2. Unconditional (jump)

```
j label
```

- ⇒ goto L1
- ↳ next instruction pointed at label.
- Equivalent to: beq $s0, $s0, label.

"label" is an "anchor" in the assembly code to indicate point of interest, usually as branch target

- ↳ labels are not instructions! not variable! } no memory allocated for label.

eg. If (i == j)

f = g+h;

⇓ invert the condition

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

if (i != j) goto Exit

bne $s3 (i), $s4 (j), Exit

else (i == j)

add $s0 (f), $s1 (g), $s2 (h)

less code vs

beq $s3 (i), $s4 (j), L1

j Exit (Extra instruction)

add $s0 (f), $s1 (g), $s2 (h)

- While loop:

eg. while (j == k)

i = i+1;

if ... goto

↓ ↓

bne/beq j

⇒ goto

Loop: if (j != k)

goto Exit;

i = i+1;

goto Loop;

Exit :

i → s3
j → s4
k → s5

- Any form of loop can be written w the help of conditional branches & jumps.

Loop: bne $s4, $s5, Exit ⇒ if (j != k) Exit

addi $s3, $s3, 1

j Loop ⇒ repeat loop.

Exit :

address of bne instruction

all these instr. are stored in memory.

ISA

- For loop

or convert to while loop.

eg. for (① i==0 ; ② i < 10 ; ③ i++)
a = a + 5; ④

i → $s0
a → $s2

"format"
1. initialise i
2. load const → reg.
3. beq / bne
4. i++ / i--
5. j

① initialise i variable — add $s0, $zero, $zero

if (i==10) — to check against — addi $s1, $zero, 10 } ①
Loop: beq $s0, $s1, Exit }

a = a+5 — addi $s2, $s2, 5 } ④
i = i+1 — addi $s0, $s0, 1 } ③
j Loop

Exit :

- Inequalities.
  - branch-if-less-than?
    blt → pseudo-instr.
  - use slt or slti — $s2 as constant
    Set-on-less-than

slt $t0, $s1, $s2
bne $t0, $zero, L
==
if ($s1 < $s2) goto L

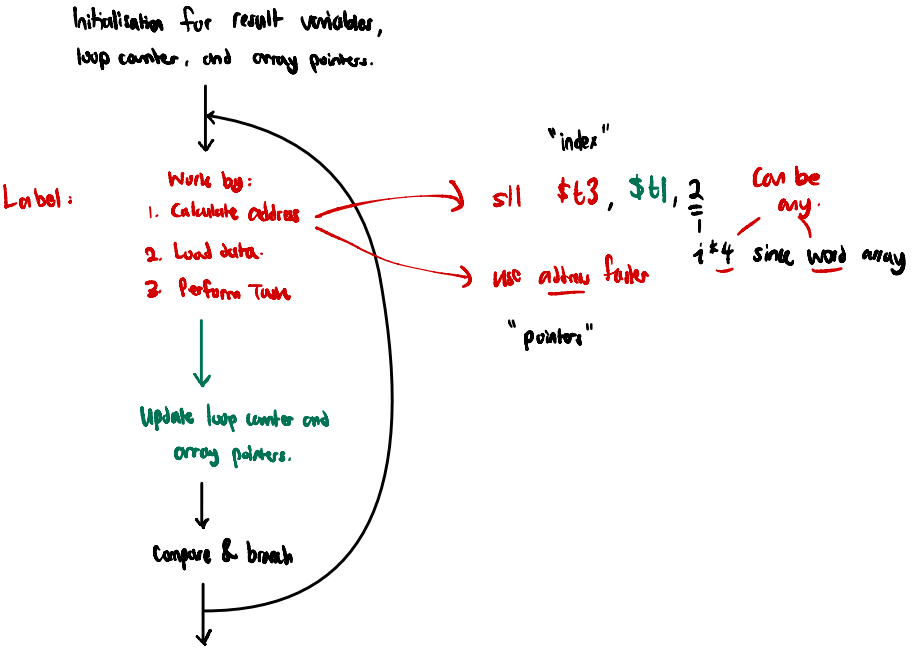
|||

blt $s1, $s2, L

slt $t0, $s1, $s2

if $s1 < $s2,
  $t0 will be set to 1,
else
  $t0 will be set to 0.

- Array and Loop.



```
sum = 0;
for (i=0; sum==0; i++) {
  switch (x[i]) {
    case  0: x[i] = -1;
    case -1: break;
    default: sum = x[i] + i + 5;
  }
}
```

Assume that you have the following variable-to-register mapping.

sum: $1	i: $2	base address of x: $3

Switch-case implementation:

```
        ADD $1, $0, $0, 0    # sum = 0
        ADD $2, $0, $0, 0    # i = 0
loop:   BEQ $1, $0, cont     # while (sum == 0) continue
        BEQ $0, $0, end      # otherwise, end

cont:   ADD $4, $2, $2, 0    # array element is 2 bytes, so 2i
        ADD $4, $4, $3, 0    #   $4 = addr of x[i]
        LW  $5, $4, 0        #   $5 = x[i]
        BEQ $5, $0, case0    # switch(x[i]) --> case  0:
        BEQ $5, $7, add      # switch(x[i]) --> case -1: (add)
        ADD $1, $5, $2, 5    # default: sum = x[i] + i + 5
        BEQ $0, $0, add      # unconditional branch to add
case0:  SW  $7, $4, 0        # x[i] = -1
add:    ADD $2, $2, $0, 1    # i = i + 1

break:  BEQ $0, $0, loop     # goto loop
end:
```