MIPS Architecture

- Multiplication and division instructions, which run asynchronously from other instructions.
- A pair of 32-bit registers, **HI** and **LO**, are provided.
- The program counter has 32 bits.
- The two low-order bits always contain zero. Why?
- MIPS I instructions are 32 bits long and are aligned to their natural word boundaries.

MIPS Architecture

- Values must be fetched from memory before (add and sub) instructions can operate on them
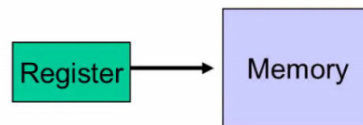
Load word:
lw $t0, memory-address
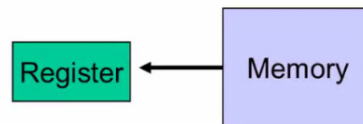
Store word:
sw $t0, memory-address

MIPS Architecture – Memory Organization

- Viewed as a large single-dimension array with access by address
- A memory address is an index into the memory array
- Byte addressing means that the index points to a byte of memory, and that the unit of memory accessed by a load/store is a byte
- Bytes are load/store units, but most data items use larger words
- For MIPS, a word is 32 bits or 4 bytes.
- 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$
  - words are aligned
  - what are the least 2 significant bits of a word address?

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
	...

MIPS Architecture

- ❑ $gp points to the area in memory that saves global variables.
- ❑ The space allocated on stack by a procedure is termed the activation record (includes saved values and data local to the procedure).

$Sp\downarrow$ block–64K 2^{32} 2^{16} 64K

Stack → Activation record
Dynamic data (heap)
Static data (globals)
Text (instructions)

- ❑ Frame pointer points to the start of the record. $gp
- ❑ Stack pointer points to the end.
- ❑ Variable addresses are specified relative to $fp as $sp may change during the execution of the procedure
- ❑ Dynamically allocated storage (with malloc()) is placed on the heap
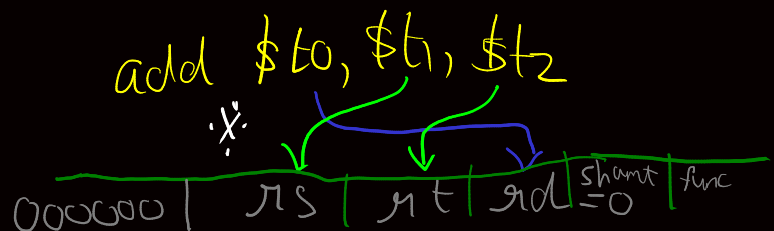
RUPSA DHAR has left the meeting

R | J formats

Regs ⟋ immediate ⟍ jump

Type

Type	opcode					
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	immediate (16)			
J	opcode (6)	address (26)				

rs – register source | rt – reg. target | rd = reg. destination

shamt = Shift amount | funct – function field (select variant)

(function code #)

All R type instructions have 000000 as opcode

Operand order is fixed

C – Code   a = b+c;

add $t0, $t1, $t2

000000	rs	rt	rd	shamt=0	func

I type: Opc (6)   Src reg (5)   dest (5)   imm (16)

add — R type
addi = I-type   X!   addi $17, $2, 1

| OPC | 00010 | 10001 |

J type | Opcode (6) | Jump addr (26) |

→ To convert 26 bits to 32 bit
→ Pad the end with **00** (alignment)   each instruction is word aligned

first 4 bits of
→ The first 4 are the PC ^ → Why (?)   why specifically PC?
                                          form of segmentation

4 ($2) :— 4 bytes offset from value in $2   (256 megs)

(goes to shamt i think for R-type)

32 bit immediate value in a register →

li $t0, 0x12345678

li is a pseudo instruction
lui $t0, 0x1234 -> load upper immediate
ori $t0, 0x5678

unconditional jumps : 26 - bits
conditional jumps - 16 bits -> program must fit into 16 bits
Use PC-Relative then

Conditional value: (address-PC)/4
Unconditional value : Address/4

bne is at 80012 , Exit is at 800024
so the address is supposed to be 12/4 = 3

jump is unconditional however -> it's jumpting to
80000 -> address is 80000/4 = 20000

Example

```
80000  Loop:sll $t1,$s3,2      # Temp reg $t1 = 4 * i
80004       add  $t1,$t1,$s6   # $t1 = address of save[i]
80008       lw   $t0,0($t1)    # Temp reg $t0 = save[i]
80012       bne  $t0,$s5, Exit # go to Exit if save[i] ≠ k
80016       addi $s3,$s3,1     # i = i + 1
80020       j    Loop          # go to Loop
80024  Exit:
```

12/4 = 3

PC -> 80012

P(+4)
80024 - 08 = 02
80016 —— 4

Assumption: program starts at address 80000 in memory.

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8		0	
80012	5	8	21		2	
80016	8	19	19		1	
80020	2		2 80000			
80024	...					

loop
add
lw
bne → I-type
addi
j
exit