MIPS 1

- "Microprocessor without Interlocked Pipeline Stages"

ISA

- Instruction Set Architecture → Contains set of instructions that processor can support. 'manual' for hardware list of functionalities.
- C++/Java : A + B

⇓ Compiler
- translates this into assembly language statements

add A, B

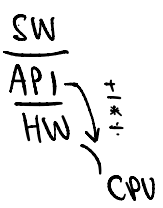
↓ Assembler
- translates this into machine language instructions that the processor can execute.

1000 1100 1010 1000

↓

MIPS

- An abstraction on the interface between the hardware and the low-level software



- eg IA-64, IA-32 ISA.

Assembler

Machine Code vs Assembly Language

- Instructions in binary
- hard & tedious to code
- may also be written in hexadecimal

- human readable
- easier to write, symbolic version of machine code.
- may provide "pseudo-instructions" as synthetic sugar
- when considering performance, only real instructions are counted.

set of real instructions ⇓ Macro in C

2 Major Components in CPU

- Processor ✓ → Performs computation (+, −, *, /)
- Memory ✓ (RAM) → Store code & data

Bus

bridge between the 2 components (wires / circuits).

- processor is much faster than memory

⇓

provide temporary storage for values in the processor (avoid freq. access of memory)

⇓

"memory instruction" registers ↔ same speed as processor's ALU

1. Instruction to move data into register ((load) ⇒ variable mapping.
2. Arithmetic operations can now work directly on registers only
   - replaces all variables w/ corresponding mapped registers in memory.
   - able to use a constant value instead of register value.
3. Instruction is executed sequentially by default
   - control flow instructions ⇒ changes control flow based on condition
     - loop (repetition) & if-else (selection) are supported.
4. Move data from register into memory (store)

General Purpose Registers

- fast memories in processor
  - data are transfered from memory to registers for faster processing

- limited in number (MIPS)
  - a typical architecture has 16 to 32 registers
  - Compiler associate variables in a program with registers (mapping)

- Registers have no data type (only stores 32-bit numbers)
  - Unlike program variables.
  - machine assembly instruction assumes the data stored in register is of the correct type. (eg. add ⇒ 32 bit 2's)

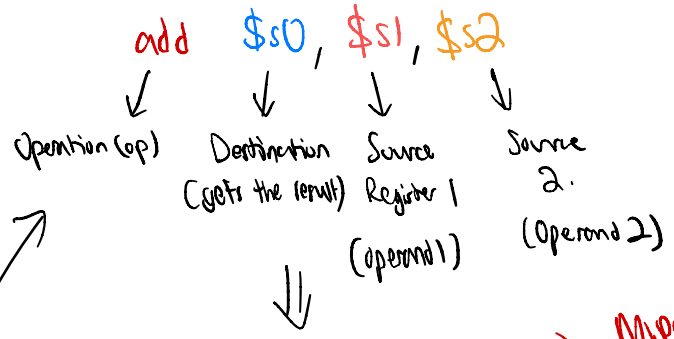
- MIPS register.
  - Can be referred by number $1, $2 or a name eg. $a0, $t1 (addu ⇒ us bin no., fadd ⇒ IEE 754)
  - each number has a corresponding name.
  - register 1 $at is reserved for assembler
    register 26-27 $k0-$k1 is reserved for OS.

Name	Register number	Usage
$zero	0	Constant value 0
$v0-$v1	2-3	Values for results and expression evaluation
$a0-$a3	4-7	Arguments (pass params of funcs)
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Program variables

Name	Register number	Usage
$t8-$t9	24-25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

MIPS Assembly Language

- Each instruction executes a simple command
  - has a counterpart in high-level language.
- Each line of assembly code contains at most 1 instruction.
- # is used for comments.
- General Syntax:



→ MIPS arithmetic operation are mainly register-to-register

- Every instruction have ~~at most~~ 3 operands

"Variable mapping" by compiler ⇒ memory instructions

a = b + c

Complex Instructions)

a = b + c - d.

⇒ t0 = b + c
a = t0 - d.

temporary register.

```
add $t0, $s1, $s2
sub $s0, $t0, $s3
```

Arithmetic Operation – views the content of a register as a single quantity (signed or unsigned integer)
↳ i.e. 32 bit number

Addition	`add $s0, $s1, $s2`	$s0 = $s1 + $s2
Subtraction	`sub $s0, $s1, $s2`	$s0 = $s1 − $s2
Addition (constant)	`addi $s0, $s1, 4` ("immediate operand"; number $-2^{15} \to 2^{15}-1$; 16 bit 2's)	$s0 = $s1 + 4

↳ no subi since can use addi with negative constant.

Assignment Operation — appears very often in code

f=g	`addi $s0, $s1, $zero`	$s0 = $s1 + 0
pseudo-instruction	`move $s0, $s1`	avoided in CS2100

↳ "Fake" instruction that get translated to the corresponding MIPS instruction(s)
↳ For convenience in coding.

Logical Operations – view register as 32 raw bits

Logical operation	C operator	Java operator	MIPS instruction
Shift Left	<<	<<	sll
Shift right	>>**	>>, >>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT*	~	~	nor
Bitwise XOR	^	^	xor xori

*with some tricks **this is "arithmetic" shift in both GCC and Clang (compiler dependent)

shift left logical
shift right logical
→ If all 1, then 1, else 0.
→ If all 0, then 0, else 1. → force certain bits to 1s.
→ Not (or) → If all 0, then 1, else 0.
→ Exclusive or → Not Equal, then 1, else 0.

and $t0, $t1, $t2
"mask"
- masking the interested bits to be copied to destination register.
- Wherever the bits are set to 1, the corresponding bits in $t1 will be copied over.

NOT operation?
Make second operand to be 0.
`nor $t0, $t0, $zero` (32 bit, all zero.)
∴ keep instruction set small

no nori cause not much used.
Can get NOT by making second operand all 1s.

Shift Left Logical (reverse for right)

- move all the bits in a word to the left by a number of positions.
- fill the emptied positions with zeroes.
- 1011 1000 0000 0000 0000 0000 0000 1001

`sll $t2, $s0, 4` ↔ $t2 = $s0 << 4
(4: 0-31 (since 32-bit register))

1000 0000 0000 0000 0000 0000 1001 0000

- equivalent math operation for shifting n bits)

0001 → 0010 → 0100 → 1000
1 → 2 → 4 → 8

Multiplication by 2^n (left)
Division by 2^n (right)

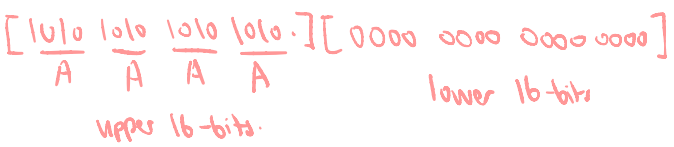
- Much faster than multiplication / division.

eg. a = a*8 → `sll $s0, $s0, 3`
($2^n = 8 \Rightarrow n = 3$)

Load large constant into a register

- 32 bit constant into 32 bit register.
- Use "load upper immediate" to set the upper 16-bit:

① `lui $t0, 0xAAAA` (max is 16-bits)



- Use "or immediate" to set the lower-order bits.

② `ori $t0, $t0, 0xF0F0` (max is 16-bits)