Five classic components

I am like a control tower

I am like a pack of file folders

I am like a conveyor belt + service stations

I exchange information with outside world
MIPS operations and operands

- Operation specifies **what function** to perform by the instruction
- Operand specifies **what quantity** to use with the instruction

- MIPS operations
  - Arithmetic (integer/floating-point)
  - Logical (AND, OR, etc)
  - Shift (moves bits around)
  - Compare (equality test)
  - Load/store (get/put stuff in memory)
  - Branch/jump (make decisions)
  - System control and coprocessor

- MIPS operands
  - Registers (one of 32 general-purpose regs)
  - Fixed registers (e.g., HI/LO)
  - Memory location (place in memory)
  - Immediate value (constant)

**Examples**
- addi $t0, $t1, 10
- $t0 = $t1 + 10

MIPS arithmetic

- `<op> <r_target>, <r_source1>, <r_source2>`

- All arithmetic instructions have 3 operands
  - Operand order in notation is fixed; target (destination) first
  - Two source registers and one target (destination) register
  - Operands are either 2 registers or 1 register + 1 immediate (constant)
  - Destination is **always** a register

- Examples
  - add $s1, $s2, $s3  # $s1 ← $s2 + $s3
  - sub $s4, $s5, $s6  # $s4 ← $s5 − $s6
### MIPS registers

<table>
<thead>
<tr>
<th>General-Purpose Registers</th>
<th>Special-Purpose Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>r0</td>
</tr>
<tr>
<td>$at</td>
<td>r1</td>
</tr>
<tr>
<td>$v0</td>
<td>r2</td>
</tr>
<tr>
<td>$v1</td>
<td>r3</td>
</tr>
<tr>
<td>$a0</td>
<td>r4</td>
</tr>
<tr>
<td>$a1</td>
<td>r5</td>
</tr>
<tr>
<td>$a2</td>
<td>r6</td>
</tr>
<tr>
<td>$a3</td>
<td>r7</td>
</tr>
<tr>
<td>$t0</td>
<td>r8</td>
</tr>
<tr>
<td>$t1</td>
<td>r9</td>
</tr>
<tr>
<td>$t2</td>
<td>r10</td>
</tr>
<tr>
<td>$t3</td>
<td>r11</td>
</tr>
<tr>
<td>$t4</td>
<td>r12</td>
</tr>
<tr>
<td>$t5</td>
<td>r13</td>
</tr>
<tr>
<td>$t6</td>
<td>r14</td>
</tr>
<tr>
<td>$t7</td>
<td>r15</td>
</tr>
</tbody>
</table>

**General-purpose registers (GPRs)**

- The name GPR implies that **all these registers can be used as operands in instructions**
- Still, **conventions and limitations** exist to keep GPRs from being used arbitrarily (from the PRM)
  - $0, termed $zero, always has a value of “0”
  - $31, termed $ra (return address), is reserved for storing the return address for subroutine call/return
  - Register usage and related software conventions are typically summarized in “application binary interface” (ABI) – important when writing system software such as an assembler or a compiler

- 32 GPRs in MIPS
  - Are they sufficient?
Special-purpose registers

- **HI/LO registers** used to store result from multiplication
- **PC register** (program counter)
  - Always keeps the pointer to the current program execution point; instruction fetching occurs at the address in PC
  - Not directly visible and manipulated by programmers in MIPS
- **Other instruction set architectures**
  - May not have HI/LO; use GPRs to store the result of multiplication
  - May allow storing to PC to make a jump

Instruction encoding

- Instructions are **encoded** in **binary numbers**
  - Assembler translates assembly programs into binary numbers
  - Machine (processor) decodes binary numbers to figure out what the original instruction is
  - MIPS has a fixed, 32-bit instruction encoding
- Encoding should be done in a way that decoding is easy
- **MIPS instruction formats**
  - **R-format**: arithmetic instructions
  - **I-format**: data transfer/arithmetic/branch instructions
  - **J-format**: jump instruction format (changes program counter)
  - (**FI-/FR-format**: floating-point instruction format)
## MIPS instruction formats

<table>
<thead>
<tr>
<th>Name</th>
<th>bit 31</th>
<th>Fields</th>
<th>bit 0</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Size</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
<tr>
<td>R-format</td>
<td></td>
<td>op (opcode)</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>I-format</td>
<td></td>
<td>op (opcode)</td>
<td>rs</td>
<td>rt</td>
</tr>
<tr>
<td>J-format</td>
<td></td>
<td>op (opcode)</td>
<td></td>
<td>target address</td>
</tr>
</tbody>
</table>

### Instruction encoding example

```
add $t0,$t1,$t8
```

- **$t8 is register 24**
  - rt = 11000 (5 bits)
- **$t1 is register 9**
  - rs = 01001 (5 bits)
- **$t0 is register 8**
  - rd = 01000 (5 bits)
- **operation is “addition”**
  - opcode = 000000 (6 bits)
  - funct = 100000 (6 bits)
- **shamt is unused**
  - shamt = 00000 (5 bits)

Resulting encoded instruction:

```
00000000100110000100000000100000
```

---

**CS/CoE0447: Computer Organization and Assembly Language**

**University of Pittsburgh**
Dealing with immediate

- Many operations involve small “immediate” value
  - a = a + 1
  - b = b – 4
  - c = d & 0xff

- Example instructions
  - `addi` $s3, $s2, 1  
    # $s3 ← $s2 + 1
  - `addi` $s4, $s1, -4  
    # $s4 ← $s1 + (-4)
  - `andi` $s5, $s0, 0xff  
    # $s5 ← $s0 & 0x000000ff

- Immediate is pos/neg up to 15 bits (15 bit value with 1 bit “sign”)
- `li` $reg,immediate  
  # $s3 ← 0xFDECBA98 (up to 32 bits)

<table>
<thead>
<tr>
<th>Name</th>
<th>Fields</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-format</td>
<td>op</td>
<td>rs</td>
</tr>
</tbody>
</table>

Interacting with the OS

- We need the OS’s help!!!
  - How to print a number? (output)
  - How to read a number? (input)
  - How to terminate (halt) a program?
  - How to open, close, read, write a file?
  - These are operating system “services”

- Special instruction: `syscall`
  - A "software interrupt" to invoke OS for an action (to do a service)
  - Need to indicate the service to perform (e.g., print vs. terminate)
  - May also need to pass an argument value (e.g., number to print)
A few useful syscalls

- syscall takes a service ID (number) sent to OS in $v0
  <load arguments>
  <set service id in $v0>
  syscall

- Print integer
  - $v0=1, $a0=integer to print

- Read integer
  - $v0=5, after syscall, $v0 holds the integer read from keyboard

- Print string
  - $v0=4, $a0=memory address of string to print (null terminated)

- Exit (halt)
  - $v0=10, no argument

- See MARS docs for more!!! Also, attend recitation.

Example: Print 100d

```assembly
li $a0,100    # value to print
li $v0,10     # print int service
syscall        # call OS
```

Example: First Asm. Program!

Program should do the following:

1. Ask user for a number, X
2. Add 100 to X
3. Print the result (X+100)
4. Exit

What do we need?

- syscall to input, output number, exit program
- add instruction for X + 100
- load immediate
Example: First Asm. Program!

```assembler
li $v0,5          # read integer, X
syscall          # returns X in $v0
addi $a0,$v0,100 # $a0 = $v0 + 100
li $v0,1         # print integer in $a0
syscall          # invoke OS
li $v0,10        # exit program
syscall
```

Example: Second Asm. Program!

- Let’s clean this up a bit.
  - We should prompt the user to ask for a number.
  - We should print a prompt with the output.

- We need to use **strings** in the assembly program.
  - The strings are data!
  - Specify string name, string type, and string value

- Data is specified in special part of program: “data section”
- Data has general format:
  ```
  name: .type data-values
  allowed types are: asciiz, word, byte, etc.
  ```
.data
msg1: .asciiz  "Enter a value:\n"
msg2: .asciiz  "Sum of value and 100:\n"

.text
li $v0,4          # prompt user (print string)
la $a0,msg1      # indicate the message
syscall
li $v0,5          # read integer, X
syscall
addi $s0,$v0,100  # $s0 = X + 100
li $v0,4          # output message
la $a0,msg2      # indicate the message
syscall
li $v0,1          # print integer
move $a0,$s0      # value to print
syscall
li $v0,10        # exit program
syscall

Logic instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Fields</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>op</td>
<td>rs</td>
</tr>
<tr>
<td>R-format</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Bit-wise logic operations
  - `<op> <r_{target}>`, `<r_{source1}>`, `<r_{source2}>`
  - Examples
    - and $s3, $s2, $s1  # $s3 $eq $s2 & $s1
    - or $t3, $t2, $t1  # $t3 $eq $t2 | $t1
    - nor $s3, $t2, $s1  # $s3 $eq !$t2 | $s1
      - note: nor $s3,$t2,$0 is $s3 $eq !(t2) (not of $t2)
    - xor $s3, $s2, $s1  # $s3 $eq $s2 ^ $s1
      - note: xor produces 1 iff one of the operands is 1
Logic Instructions with Immediates

- Logic instructions have I-format (small immediate) versions
  - `andi` $s0,$s1,0xff00
  - `ori` $s0,$s1,0x0ff0
  - `xori` $s0,$s1,0xf00f
  - `nori` $s0,$s1,0xffff

- Upper bits (bits 31..16) are set to 0s by instruction
  - E.g., 0xff00 is really 0x0000ff00
  - This operation is known as “zero extension”

Handling long immediate number

- `li` allows loading large immediates (> 16 bits)
  - **Pseudo-operation**: Assembler “converts” to actual machine instructions
- Consider: `li $s3,0xAA55CC33`

- Converted to two instructions:
  - `lui $s3, 1010 1010 0101 0101b`
  - `ori $s3, $s3, 1100 1100 0011 0011b`

- Then we fill the low-order 16 bits
  - `ori $s3, $s3, 1100 1100 0011 0011b`
Shift instructions

- Bits change their positions inside a word
- `<op> <r_target> <r_source> <shift_amount>

Examples
- `sll $s3, $s4, 4`  \# $s3 $= \$s4 $\ll 4
- `srl $s6, $s5, 6`  \# $s6 $= \$s5 $\gg 6

- Shift amount can be in a register ("shamt" is not used)
- Shift right arithmetic (sra) keeps the sign of a number
  - `sra $s7, $s5, 4`

Let's try it in MARS!!! (mips6.asm)