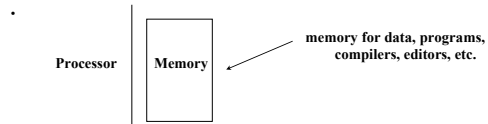


Stored Program Concept

- Instructions are bits
- Programs are stored in memory
 - to be read or written just like data



- Fetch & Execute Cycle
 - Instructions are fetched and put into a special register
 - Bits in the register "control" the subsequent actions
 - Fetch the "next" instruction and continue

1

Instructions:

- Language of the Machine
- More primitive than higher level languages
 - e.g., no sophisticated control flow
- Very restrictive
 - e.g., MIPS Arithmetic Instructions
- We'll be working with the MIPS instruction set architecture
 - similar to other architectures developed since the 1980's

2

Architecture Specification

- Data types:
 - bit, byte, bit field, signed/unsigned integers logical, floating point, character
- Operations:
 - data movement, arithmetic, logical, shift/rotate, conversion, input/output, control, and system calls
- # of operands:
 - 3, 2, 1, or 0 operands
- Registers:
 - integer, floating point, control
- Instruction representation as bit strings

3

Characteristics of Instruction Set

- Complete
 - Can be used for a variety of application
- Efficient
 - Useful in code generation
- Regular
 - Expected instruction should exist
- Compatible
 - Programs written for previous versions of machines need it
- Primitive
 - Basic operations
- Simple
 - Easy to implement
- Smaller
 - Implementation

4

Example of multiple operands

- Instructions may have 3, 2, 1, or 0 operands
- Number of operands may affect instruction length
- Operand order is fixed (destination first, but need not that way)

add \$s0, \$s1, \$s2 ; Add \$s2 and \$s1 and store result in \$s0

add \$s0, \$s1 ; Add \$s1 and \$s0 and store result in \$s0

add \$s0 ; Add contents of a fixed location to \$s0

add ; Add two fixed locations and store result

5

Where operands are stored

- Memory locations
 - Instruction include address of location
- Registers
 - Instruction include register number
- Stack location
 - Instruction opcode implies that the operand is in stack
- Fixed register
 - Like accumulator, or depends on inst
 - Hi and Lo register in MIPS
- Fixed location
 - Default operands like interrupt vectors

6

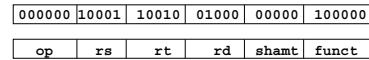
Addressing

- Memory address for load and store has two parts
 - A register whose content are known
 - An offset stored in 16 bits
- The offset can be positive or negative
 - It is written in terms of number of bytes
 - It is but in instruction in terms of number of words
 - 32 byte offset is written as 32 but stored as 8
- Address is content of register + offset
- All address has both these components
- If no register needs to be used then use register 0
 - Register 0 always stores value 0
- If no offset, then offset is 0

7

Machine Language

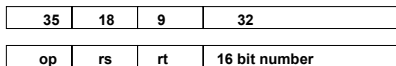
- Instructions, like registers and words of data, are also 32 bits long
 - Example: `add $t0, $s1, $s2`
 - registers have numbers, `$t0=9, $s1=17, $s2=18`
- Instruction Format:



8

Machine Language

- Consider the load-word and store-word instructions,
 - What would the regularity principle have us do?
 - New principle: Good design demands a compromise
- Introduce a new type of instruction format
 - I-type for data transfer instructions
 - other format was R-type for register
- Example: `lw $t0, 32($s2)`



- Where's the compromise?

9

Control

- Decision making instructions
 - alter the control flow,
 - i.e., change the "next" instruction to be executed

- MIPS conditional branch instructions:

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

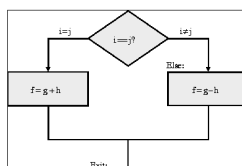
- Example: `if (i==j) h = i + j;`

```
bne $s0, $s1, Label
add $s3, $s0, $s1
Label: ....
```

10

Conditional Execution

- A simple conditional execution
- Depending on `i==j` or `i!=j`, result is different



11

Control Flow

- We have: `beq`, `bne`, what about Branch-if-less-than?
- New instruction:

```

if $s1 < $s2 then
    $t0 = 1
else
    $t0 = 0

```

- Can use this instruction to build "`blt $s1, $s2, Label`"
 - can now build general control structures
- Note that the assembler needs a register to do this,
 - there are policy of use conventions for registers

12

Constants

- Small constants are used quite frequently (50% of operands)
e.g., `A = A + 5;`
`B = B + 1;`
`C = C - 18;`
- Solutions? Why not?
 - put 'typical constants' in memory and load them.
 - create hard-wired registers (like \$zero) for constants like one.
- MIPS Instructions:


```
addi $29, $29, 4
slli $8, $19, 10
andi $29, $29, 6
ori $29, $29, 4
```
- How do we make this work?

13

Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

| | | | | | | |
|---|----|----------------|----|----------------|-------|-------|
| R | op | rs | rt | rd | shamt | funct |
| I | op | rs | rt | 16 bit address | | |
| J | op | 26 bit address | | | | |

- rely on compiler to achieve performance
 - what are the compiler's goals?
- help compiler where we can

14

Addresses in Branches and Jumps

- Instructions:


```
bne $t4, $t5, Label    Next instruction is at Label if $t4 ≠ $t5
beq $t4, $t5, Label    Next instruction is at Label if $t4 = $t5
j Label                Next instruction is at Label
```
- Formats:

| | | | | |
|---|----|----------------|----|----------------|
| I | op | rs | rt | 16 bit address |
| J | op | 26 bit address | | |
- Addresses are not 32 bits
 - How do we handle this with load and store instructions?

15

Address Handling

- Instructions:


```
bne $t4, $t5, Label    Next instruction is at Label if $t4 ≠ $t5
beq $t4, $t5, Label    Next instruction is at Label if $t4 = $t5
```
- Formats:

| | | | | |
|---|----|----|----|----------------|
| I | op | rs | rt | 16 bit address |
|---|----|----|----|----------------|
- Could specify a register (like lw and sw) and add it to address
 - use Instruction Address Register (PC = program counter)
 - most branches are local (principle of locality)
- Jump instructions just use high order bits of PC
 - address boundaries of 256 MB

16

MIPS Instruction Format

| | | | | | | | | | | | | |
|-----------|----|-------|----|----|-------|----|---------|----------------|--------------|----------------|---|--|
| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 | |
| LW | | REG 1 | | | REG 2 | | | LOAD ADDRESS | | OFFSET | | |
| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 | |
| SW | | REG 1 | | | REG 2 | | | STORE ADDRESS | | OFFSET | | |
| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 | |
| BEQ/BNE/J | | REG 1 | | | REG 2 | | | BRANCH ADDRESS | | OFFSET | | |
| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 | |
| R-TYPE | | REG 1 | | | REG 2 | | | DST | SHIFT AMOUNT | ADD/AND/OR/SLT | | |
| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 | |
| I-TYPE | | REG 1 | | | REG 2 | | | IMMEDIATE DATA | | | | |
| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 | |
| JUMP | | JUMP | | | | | ADDRESS | | | | | |

17

Our Example Machine Specification

- 16-bit data path (can be 4, 8, 12, 16, 24, 32)
- 16-bit instruction (can be any number of them)
- 16-bit PC (can be 16, 24, 32 bits)
- 16 registers (can be 1, 4, 8, 16, 32)
- With m register, log m bits for each register
- Offset depends on expected offset from registers
- Branch offset depends on expected jump address
- Many compromise are made based on number of bits in instruction

18

Instruction

- LW R2, #v(R1) ; Load memory from address (R1) + v
- SW R2, #v(R1) ; Store memory to address (R1) + v
- R-Type – OPER R3, R2, R1 ; Perform $R3 \leftarrow R2 \text{ OP } R1$
 - Five operations ADD, AND, OR, SLT, SUB
- I-Type – OPER R2, R1, V ; Perform $R2 \leftarrow R1 \text{ OP } V$
 - Four operation ADDI, ANDI, ORI, SLTI
- B-Type – BC R2, R1, V ; Branch if condition met to address PC+V
 - Two operation BNE, BEQ
- Shift class – SHIFT TYPE R2, R1 ; Shift R1 of type and result to R2
 - One operation
- Jump Class -- JAL and JR (JAL can be used for Jump)
 - What are th implications of J vs JAL
 - Two instructions

19

Instruction Encoding

- LW/SW/BC – Requires opcode, R2, R1, and V values
- R-Type – Requires opcode, R3, R2, and R1 values
- I-Type – Requires opcode, R2, R1, and V values
- Shift class – Requires opcode, R2, R1, and shift type value
- JAL requires opcode and jump address
- JR requires opcode and register address
- Opcode – can be fixed number or variable number of bits
- Register address – 4 bits if 16 registers
- How many bits in V?
- How many bits in shift type?
 - 4 for 16 types, assume one bit shift at a time
- How many bits in jump address?

20

Encoding Selection

- Two fields Opcode
 - Class of function and function in that class, may require more bits as in each class functions needs to be encoded
- One level opcode
 - In our example it is more optimal, 16 op codes are sufficient
- Each register takes 4 bits to encode
- Shift type requires four bits
- To pack instructions in 16 bits
 - V is 4 bits
 - Branch offset 4 bits
 - How many bits in jump address?
 - Only 12 bits jump address required

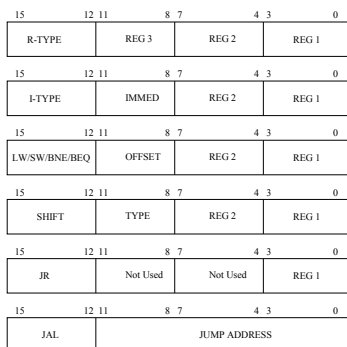
21

Trade Offs

- Only 4 bit immediate value
 - It is ok as constants are usually small
- Only 4-bit LW/SW address offset
 - This is real small
 - Good for small programs
- 12-bit jump address
 - Not a real limitation
- Branch offset 4 bits
 - Has constraints, but can be managed with jump
 - Depends on types of program
- Instructions are few
 - It is a quite a complete instruction set
- The instruction set is slightly redundant

22

Instruction Format



23

Operation for Each Instruction

| LW: | SW: | R/I/S-Type: | BR-Type: | JMP-Type: |
|-----------------------|-----------------------|-----------------------------|-------------------------|--------------|
| 1. READ INST | 1. READ INST | 1. READ INST | 1. READ INST | 1. READ INST |
| 2. READ REG 1 | 2. READ REG 1 | 2. READ REG 1 | 2. READ REG 1 | 2. |
| READ REG 2 | READ REG 2 | READ REG 2 | READ REG 2 | |
| 3. ADD REG 1 + OFFSET | 3. ADD REG 1 + OFFSET | 3. OPERATE on REG 1 & REG 2 | 3. SUB REG 2 from REG 1 | 3. |
| 4. READ MEM | 4. WRITE MEM | 4. | 4. | 4. |
| 5. WRITE REG2 | 5. | 5. WRITE DST | 5. | 5. |

24