CS 465 Lecture 2

MIPS ISA I: Introduction

*Slides adapted from Computer Organization and Design by Patterson and Henessey

Instruction Set Architecture

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- ISA also includes information needed to write a machine program that runs correctly
  - Instructions, resources, and usages
- Many modern computers have simple instruction sets
  - RISC and CISC

The MIPS ISA

- Used as the example throughout the book
- Stanford MIPS commercialized
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs : RISC
  - See MIPS Reference Data tear-out card

Road Map – MIPS ISA

- MIPS basic instructions ←
  - Arithmetic instructions: add, addi, sub
  - Logical operations
  - Data transfer instructions: lw, sw, lb, sb, lbu
  - Control instructions
- MIPS instruction format
  - ISA design principles
- Procedure calls
- Addressing modes
- Other misc.

Review: Number Representation

- Unsigned binary integer
- Signed integer: 2's-complement
  - Non-negative numbers have the same unsigned and 2's-complement representation
- Signed negation
  - Complement and add 1
  - Example:
    - +2 = 0000 0000 … 0010
    - –2 = 1111 1111 … 1101 + 1
      = 1111 1111 … 1110

Exercises

- Convert the number 35 to 8-bit binary notation.
- Convert the number -35 to 8-bit binary notation.

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

Binary to Hex ?

- 0001 0011 0101 0111 1001 1011 1101 1111

Road Map

- MIPS basic instructions
  - Arithmetic/logical instructions ←
  - Control instructions
  - Data transfer instructions
- ISA design principles
  - RISC philosophy: hardware implementation focuses on optimizing a small core set of instructions

Arithmetic/Logical Operations

- Syntax (only register operands)
  - **op dest, src1, src2**
- Example operators – work done in registers (dest = src1 op src2)
  - `add dest,src1,src2`
  - `sub dest,src1,src2 #dest = src1-src2`
  - `and dest,src1,src2 #bitwise and`
  - `or dest,src1,src2  #bitwise or`
- ***Design Principle 1*: Simplicity favors regularity**

Register Operands

- Many MIPS arithmetic instructions use register operands only
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
- Assembler names
  - $t0, $t1, …, $t7 for temporary values ($8-$15)
  - $s0, $s1, …, $s7 for saved variables ($16-$23)
- ***Design Principle 2:* Smaller is faster**

Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

  - Assume that f, …, j in $s0, …, $s4

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

MIPS Registers

Register Number	Assembly Name	Usage
$0	$zero	Hard-wired to 0
$1	$at	Assembler temporary (pseudo-instruction)
$2-$3	$v0, $v1	Return values
$4-$7	$a0-$a3	Arguments
$8-$15	$t0-$t7	Temporary values (caller save)
$16-$23	$s0-$s7	Saved values (callee save)
$24-$25	$t8, $t9	More temporary values (caller save)
$26-$27	$k0, $k1	Reserved for kernel (do not use)
$28	$gp	Global area base pointer
$29	$sp	Stack top pointer
$30	$fp	Frame pointer
$31	$ra	Return address

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., copy/move between registers

```
add $t2, $s1, $zero
```

    - Can be written as pseudocode:

```
move $t2, $s1
```

Constant Operand

- C code:

```
g = h + 8;
```

- So far arithmetic operations only take register operands
  - Possible approach: save constant in a memory location → load value into a register → compute ☹

Constant Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
andi $s3, $s3, 4
ori  $s3, $s3, 4
```

- No subtract immediate instruction
  - Just use a negative constant

```
addi $s2, $s1, -1
```

- ***Design Principle 3:* Make the common case fast**
  - Small constant operands are common
  - Allow immediate operand can avoid a load instruction

Shifting Operations

- Move (shift) all the bits in a word to the left or right by a number of bits
- Shift instruction syntax:

```
op  dest, src, amt
```

- MIPS shifting instructions (w/ amt as a constant)
  - sll (shift left logical): shift left and fill with 0s
    - sll by i bits multiplies by 2^i
  - srl (shift right logical): shift right and fills with 0s
    - srl by i bits divides by 2^i
  - sra (shift right arithmetic): shift right and fills with the sign bit

Shifting Examples

- Shift left by 8 bits

1001 0010 0011 0100 0101 0110 0111 1000

0011 0100 0101 0110 0111 1000 0000 0000

- Shift right logical by 8 bits

1001 0010 0011 0100 0101 0110 0111 1000

0000 0000 1001 0010 0011 0100 0101 0110

- Shift right arithmetic by 8 bits

1001 0010 0011 0100 0101 0110 0111 1000

1111 1111 1001 0010 0011 0100 0101 0110

Road Map

- MIPS basic format
  - Opcode dest, src1, src2
- MIPS basic instructions
  - Arithmetic/logical instructions
  - Control instructions ←
  - Data transfer instructions

Comparison Operators

- "Set" operations
  - Syntax: op dest, src1, src2
  - Semantic: set dest according to condition (src1 xxx src2), where xxx is a comparison (gt, ge, lt, le, eq)
    - Set dest register 0 for false, 1 for true
- Examples
  - `slt reg1,reg2,reg3` *(core instruction)*
  - `slti reg1,reg2,constant` *(core instruction)*
  - `sgt reg1,reg2,reg3` *(pseudo-instruction)*
  - `sle reg1,reg2,reg3` *(pseudo-instruction)*
  - …

Basic Branches

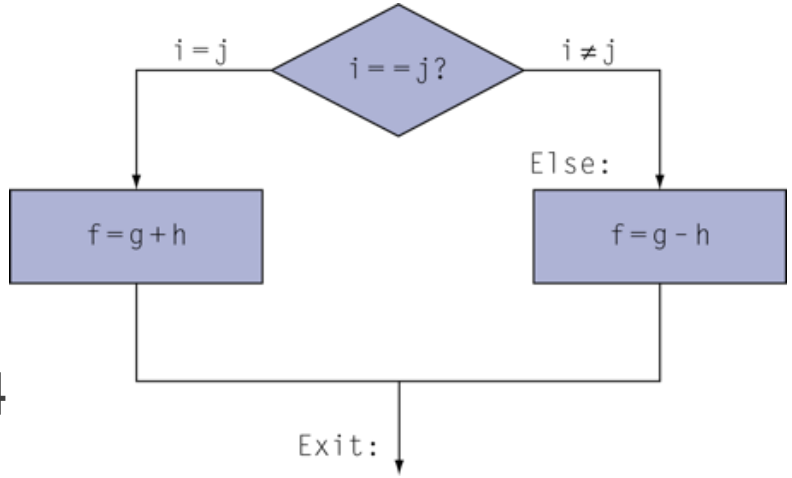
- Unconditional jumps
  - `j label` - jump to label
- Conditional branches
  - Compare on equality or inequality of two registers
    - `beq/bne rs, rt, label`
  - Compare on one register and zero
    - `bXXX reg, label` (**XXX** can be eqz, nez, gtz, …)
  - Lots of variants as *pseudo-instructions*

Example: If Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f: $s0, g: $s1, h: $s2, i: $s3, j: $s4



- Compiled MIPS code:

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

Assembler calculates addresses

Example: Loop

- Demo

Road Map

- MIPS basic instructions
  - Arithmetic/logical instructions
  - Control instructions
  - Data transfer instructions ←

Where Are Data?

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit storage (byte)

Memory Organization

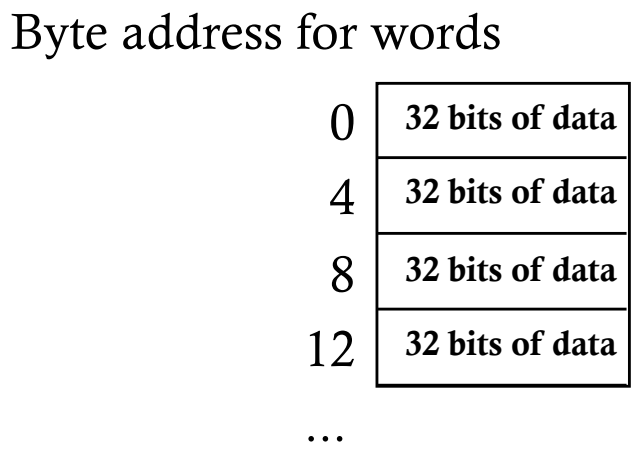
- Think of memory as a large, single-dimensional array
- Memory address is like index of the array
- Memory reference is at the byte level
- Retrieve a minimum of 1 byte from memory in each access

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
7	8 bits of data
8	8 bits of data

Byte address

Memory Operands

- Register length is 1 word = 4 bytes = 32 bits
- Words are aligned in memory
  - Address of a word must be a multiple of 4



  - If an array starts with address A and every element is one word long, then the byte address of its elements should be A, A+4, A+8, …

Big Endian vs Little Endian

- Most MIPS processors allow you to set endianness
  - Big endian: most-significant byte at least address
  - Little endian: least-significant byte at least address
- Example: $90AB12CD_{16}$

Big Endian

Address	Value
1000	90
1001	AB
1002	12
1003	CD

Little Endian

Address	Value
1000	CD
1001	12
1002	AB
1003	90

* MARS uses little endian
  - http://courses.missouristate.edu/KenVollmar/mars/Help/MarsHelpDebugging.html

Memory Load/Store

- MIPS need explicit load/store to access memory
- Loading/Storing
  - `lw register,addr` - moves value into register
    - Load word, Mem→Reg
  - `sw register,addr` - stores value from register
    - Store word, Reg→Mem
- Memory addressing format: Imm(Register)
  - Mem addr = Immediate + contents of register
  - Many variations supported in pseudo-instructions

Memory Operand Example

- C code:

```
A[12] = h + A[8];
```

  - h in $s2, base address of A in $s3, integer array A

- Compiled MIPS code: ???

lw register-operand, constant (register-with-base-address)

sw register-operand, constant (register-with-base-address)

Byte/Halfword Operations

- MIPS byte/halfword load/store
  - Used in string processing: character in C is a byte
  - lb/sb/lh/sh: same format as lw, sw
- What to do with other 24/16 bits in the 32 bit register?
  - Sign-extend: lb, lh

  - Zero-extend: lbu, lhu
  - Store ops just update with the low byte/halfword