Week 04

Instruction Set Architectures

CPU Architecture

2/31

A typical modern CPU has

- a set of data registers
- a set of control registers (incl PC)
- an arithmetic-logic unit (ALU)
- access to random access memory (RAM)
- a set of simple instructions
  - transfer data between memory and registers
  - push values through the ALU to compute results
  - make tests and transfer control of execution

Different types of processors have different configurations of the above

- e.g. different # registers, different sized registers, different instructions

Instruction Sets

3/31

Two broad families of instruction set architectures ...

RISC (*reduced instruction set computer*)

- small(ish) set of simple, general instructions
- separate computation & data transfer instructions
- leading to simpler processor hardware
- e.g. MIPS, RISC, Alpha, SPARC, PowerPC, ARM, ...

CICS (*complex instruction set computer*)

- large(r) set of powerful instructions
- each instruction has multiple actions (comp+store)
- more circuitry to decode/process instructions
- e.g. PDP, VAX, Z80, Motorola 68xxx, Intel x86, ...

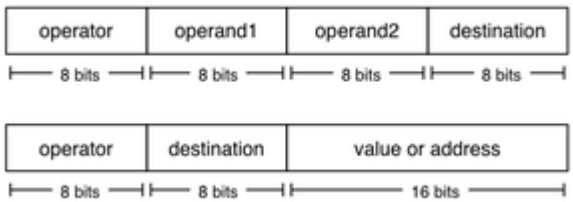
... Instruction Sets

4/31

Machine-level instructions ...

- typically have 1-2 32-bit words per instruction
- partition bits in each word into operator & operands
- #bits for each depends on #instructions, #registers, ...

Example instruction word formats:



Operands and *destination* are typically registers

... Instruction Sets

5/31

Common kinds of instructions (not from any real machine)

- **`load`** *Register*`,` *MemoryAddress*
  - copy value stored in memory at address into named register
- **`loadc`** *Register*`,` *ConstantValue*
  - copy value into named register
- **`store`** *Register*`,` *MemoryAddress*
  - copy value stored in named register into memory at address
- **`jump`** *MemoryAddress*
  - transfer execution of program to instruction at address
- **`jumpif`** *Register*`,` *MemoryAddress*
  - transfer execution of program if e.g. register holds zero value

---

... Instruction Sets

6/31

Other common kinds of instructions (not from any real machine)

- **`add`** $Register_1$`,` $Register_2$`,` $Register_3$ (similarly for **`sub`**, **`mul`**, **`div`**)
  - $Register_3 = Register_1 + Register_2$
- **`and`** $Register_1$`,` $Register_2$`,` $Register_3$ (similarly for **`or`**, **`xor`**)
  - $Register_3 = Register_1$ `&` $Register_2$
- **`neg`** $Register_1$`,` $Register_2$
  - $Register_2 = \sim Register_1$
- **`shiftl`** $Register_1$`,` *Value*`,` $Register_2$ (similarly for **`shiftr`**)
  - $Register_2 = Register_1 <<$ *Value*
- **`syscall`** *Value*
  - invoke a system service; which service determined by *Value*

Fetch-Execute Cycle

7/31

All CPUs have program execution logic like:

```
while (1)
{
   instruction = memory[PC]
   PC++  // move to next instr
   if (instruction == HALT)
      break
   else
      execute(instruction)
}
```

PC = Program Counter, a CPU register which keeps track of execution

Note that some instructions may modify PC further (e.g. JUMP)

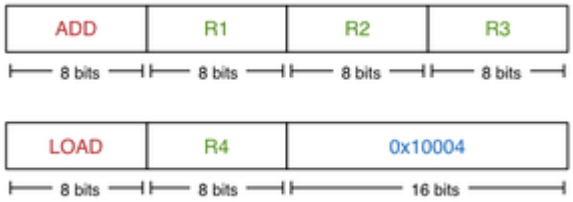
... Fetch-Execute Cycle

8/31

Executing an `instruction` involves

- determine what the operator is
- determine which registers, if any, are involved
- determine which memory location, if any, is involved
- carry out the operation with the relevant operands
- store result, if any, in appropriate register

Example instruction encodings (not from a real machine):



Assembly Language

9/31

Instructions are simply bit patterns within a 32-bit bit-string

Could describe machine programs as a sequence of hex digits, e.g.

```
Address    Content
0x100000   0x3c041001
0x100004   0x34020004
0x100008   0x0000000c
0x10000C   0x03e00008
```

Assembly languages provide a symbolic way of giving machine code

Often call "assembly language" as "assembler"

Slight notational abuse, because "assembler" also refers to a program that translates assembly language to machine code

---

... Assembly Language

10/31

Assembler = symbolic language for writing machine code

- write instructions using mnemonics rather than hex codes
- reference registers using either numbers or names
- can associate names to memory addresses

Style of expression is significantly different to e.g. C

- need to use fine-grained control of memory usage
- required to manipulate data in registers
- control structures programmed via explicit jumps

MIPS Architecture

11/31

MIPS is a well-known and relatively simple architecture

- very popular in a range of computing devices in the 1990's
- e.g. Silicon Graphics, NEC, Nintendo64, Playstation, supercomputers

We consider the MIPS32 version of the MIPS family

- using two variants of the open-source SPIM emulator
- **`qtspim`** ... provides a GUI front-end, useful for debugging
- **`spim`** ... command-line based version, useful for testing

Executables and source: http://spimsimulator.sourceforge.net/

Source code for browsing under /home/cs1521/spim/spim

... MIPS Architecture

12/31

MIPS CPU has

- 32 × 32-bit general purpose registers

- 16 × 64-bit double-precision registers
- PC ... 32-bit register (always aligned on 4-byte boundary)
- HI,LO ... for storing results of multiplication and division

Registers can be referred to as `$0..$31` or by symbolic names

Some registers have special uses e.g.

- register `$0` always has value 0, cannot be written
- registers `$1`, `$26`, `$27` reserved for use by system

More details on following slides ...

... MIPS Architecture

13/31

Registers and their usage

Reg	Name	Notes
`$0`	`zero`	the value 0, not changeable
`$1`	`$at`	**a**ssembler **t**emporary; reserved for assembler use
`$2`	`$v0`	**v**alue from expression evaluation or function return
`$3`	`$v1`	**v**alue from expression evaluation or function return
`$4`	`$a0`	first **a**rgument to a function/subroutine, if needed
`$5`	`$a1`	second **a**rgument to a function/subroutine, if needed
`$6`	`$a2`	third **a**rgument to a function/subroutine, if needed
`$7`	`$a3`	fourth **a**rgument to a function/subroutine, if needed
`$8..$15`	`$t0..$t7`	**t**emporary; must be saved by caller to subroutine; subroutine can overwrite

---

... MIPS Architecture

14/31

More register usage ...

Reg	Name	Notes
`$16..$23`	`$s0..$s7`	**s**afe function variable; must not be overwritten by called subroutine
`$24..$25`	`$t8..$t9`	**t**emporary; must be saved by caller to subroutine; subroutine can overwrite
`$26..$27`	`$k0..$k1`	for **k**ernel use; may change unexpectedly
`$28`	`$gp`	**g**lobal **p**ointer
`$29`	`$sp`	**s**tack **p**ointer
`$30`	`$fp`	**f**rame **p**ointer
`$31`	`$ra`	**r**eturn **a**ddress of most recent caller

... MIPS Architecture

15/31

Floating point register usage ...

Reg	Notes
`$f0..$f2`	hold floating-point function results
`$f4..$f10`	temporary registers; not preserved across function calls

`$f12..$f14`	used for first two double-precision function arguments
`$f16..$f18`	temporary registers; used for expression evaluation
`$f20..$f30`	saved registers; value is preserved across function calls

Notes:

- registers come in pairs of 2 × 32-bits
- only even registers are addressed for double-precision

MIPS Assembly Language

16/31

MIPS assembly language programs contain

- comments ... introduced by #
- labels ... appended with `:`
- directives ... symbol beginning with `.`
- assembly language instructions

Programmers need to specify

- data objects that live in the data region
- functions (instruction sequences) that live in the code/text region

Each instruction or directive appears on its own line

... MIPS Assembly Language

17/31

Example MIPS assembler program:

```
# hello.s ... print "Hello, MIPS"

        .data           # the data segment
msg:    .asciiz "Hello, MIPS\n"

        .text           # the code segment
        .globl main
main:
         la $a0, msg    # load the argument string
         li $v0, 4      # load the system call (print)
         syscall        # print the string
         jr $ra         # return to caller (__start)
```

Color coding: label, directive, comment

... MIPS Assembly Language

18/31

Generic structure of MIPS programs

```
# Prog.s ... comment giving description of function
# Author ...

     .data     # variable declarations follow this line
               # ...

     .text     # instructions follow this line
     .globl main
main:          # indicates start of code
               # (i.e. first user instruction to execute)
               # ...

# End of program; leave a blank line to make SPIM happy
```

... MIPS Assembly Language

19/31

MIPS programs assume the following memory layout

Region	Address	Notes
text	`0x00400000`	contains only instructions; read-only; cannot expand
data	`0x10000000`	data objects; readable/writeable; can be expanded
stack	`0x7fffefff`	grows down from that address; readable/writeable
k_text	`0x80000000`	kernel code; read-only; only accessible kernel mode
k_data	`0x90000000`	kernel data; read/write; only accessible kernel mode

---

MIPS Instructions

20/31

MIPS has several classes of instructions:

- *load and store* .. transfer data between registers and memory
- *computational* ... perform arithmetic/logical operations
- *jump and branch* ... transfer control of program execution
- *coprocessor* ... standard interface to various co-processors
- *special* ... miscellaneous tasks (e.g. syscall)

And several addressing modes for each instruction

- between memory and register (direct, indirect)
- constant to register (immediate)
- register + register + destination register

... MIPS Instructions

21/31

MIPS instructions are 32-bits long, and specify ...

- an operation (e.g. load, store, add, branch, ...)
- one or more operands (e.g. registers, memory addresses, constants)

Some possible instruction formats

Exercise 1: Add two numbers

22/31

Write MIPS assembler that behaves like

```
int x = 3;
int y = 4;
void main(void)
{
   printf("%d\n", x+y);
}
```

Hints:

- syscall 1 prints a number, syscall 4 prints a string
- `.word` allocates 4 bytes in memory and initialises it

- addition needs two values loaded into registers

Exercise 2: Adding Interactively

23/31

Modify the program so that it behaves as follows:

```
int x;  int y;

void main(void) {
   printf("First number: ");
   scanf("%d", &x);
   printf("Second number: ");
   scanf("%d", &y);
   printf("%d\n", x+y);
}
```

Hints:

- `syscalls`: read_int:5, print_str:4, print_int:1
- read_int leaves the number read in `$v0`

Addressing Modes

24/31

Memory addresses can be given by

- symbolic name (label) (effectively, a constant)
- indirectly via a register (effectively, pointer dereferencing)

Examples:

```
prog:
a:    lw    $t0, var     # address via name
b:    lw    $t0, ($s0)   # indirect addressing
c:    lw    $t0, 4($s0)  # indexed addressing
```

If `$s0` contains `0x10000000` and `&var` = `0x100000008`

- computed address for `a:` is `0x100000008`
- computed address for `b:` is `0x100000000`
- computed address for `c:` is `0x100000004`

... Addressing Modes

25/31

Addressing modes in MIPS

Format	Address computation
(register)	address = *register = contents of register
k	address = k
k(register)	address = k + *register
symbol	address = &symbol = address of symbol
symbol ± k	address = &symbol ± k
symbol ± k(register)	address = &symbol ± (k + *register)

where *k* is a literal constant value (e.g. `4` or `0x10000000`)

... Addressing Modes

26/31

Examples of load/store and addressing:

```
    .data
vec: .space   16      # int vec[4], 16 bytes of storage
    .text
__start:
    la  $t0, vec      # reg[t0] = &vec
    li  $t1, 5        # reg[t1] = 5
    sw  $t1, ($t0)    # vec[0] = reg[t1]
    li  $t1, 13       # reg[t1] = 13
    sw  $t1, 4($t0)   # vec[1] = reg[t1]
    li  $t1, -7       # reg[t1] = -7
    sw  $t1, 8($t0)   # vec[2] = reg[t1]
    li  $t2, 12       # reg[t2] = 12
    li  $t1, 42       # reg[t1] = 42
    sw  $t1, vec($t2) # vec[3] = reg[t1]
```

Exercise 3: MIPS Addressing

27/31

Consider the following memory contents and MIPS instructions

```
Label Address      Content         Instructions
x:    0x10010000   0x00010101    * la $t0, x
y:    0x10010004   0x10010000    * lw $t0, x
z:    0x10010008   0x0000002A    * lb $t0, x
eol:  0x1001000C   0x0000000A      la $s0, z
                                 * lw $t0, ($s0)
Notes:                             li $s0, 8
0x10101 = 65793                  * lw $t0, y($s0)
0x101   = 257                      lw $s0, y
                                 * lw $t0  ($s0)
                                   li $s0, 4
                                 * lw $t0, x+4($s0)
```

What will be (a) the computed address, (b) the value of the destination register (`$t0`) after each of the starred MIPS instructions is executed?

Consider the following memory contents and MIPS instructions

```
Label Address      Content         Instructions
x:    0x10010000   0x00010101    * la $t0, x
y:    0x10010004   0x10010000    * lw $t0, x
z:    0x10010008   0x0000002A    * lb $t0, x
eol:  0x1001000C   0x0000000A      la $s0, z
                                 * lw $t0, ($s0)
Notes:                             li $s0, 8
0x10101 = 65793                  * lw $t0, y($s0)
0x101   = 257                      lw $s0, y
                                 * lw $t0  ($s0)
                                   li $s0, 4
                                 * lw $t0, x+4($s0)
```

Write C code that (roughly) behaves like this assembly code.

Assume that there are global variables with the same names as the registers.

Exercise 4: Check the results

29/31

Write MIPS assembler code to display the values from the previous exercise.

Use `$a0` as the destination register for all examples

E.g.

```
lw $a0, y($s0)   # load value into arg[0] register
li $v0, 1        # set up for print_int syscall
syscall          # print the number
```

Print a "\n" after each number

Operand Sizes

30/31

MIPS instructions can manipulate different-sized operands

- single bytes, two bytes ("halfword"), four bytes ("word")

Many instructions also have variants for signed and unsigned

Leads to many opcodes for a (conceptually) single operation, e.g.

- `LB` ... load one byte from specified address
- `LBU` ... load unsigned byte from specified address
- `LH` ... load two bytes from specified address
- `LHU` ... load unsigned 2-bytes from specified address
- `LW` ... load four bytes (one word) from specified address
- `LA` ... load the specified address

All of the above specify a destination register

Exercise 5: Operand Sizes

31/31

Consider the following memory contents and MIPS instructions

```
Label Address      Content         Instructions
x:    0x10010000   0x00010101    * la  $t0, x
y:    0x10010004   0x00008000    * lw  $t0, x
z:    0x10010008   0x0000002A    * lh  $t0, x
                                 * lh  $t0, y
Notes:                           * lhu $t0, y
0x10101 = 65793                  * lbu $t0, y
0x101   = 257
0x8000  = 32768
```

What will be the value (in hexadecimal) of the destination register after each of the starred MIPS instructions is executed?

Produced: 17 Aug 2017