Why Study Assembler?

Useful to know assembly language because ...

- sometimes you are *required* to use it (e.g. device handlers)
- improves your understanding of how compiled programs execute
  - very helpful when debugging
  - understand performance issues better
- performance tweaking (squeezing out last pico-s)
  - re-write that performance critical code in assembler

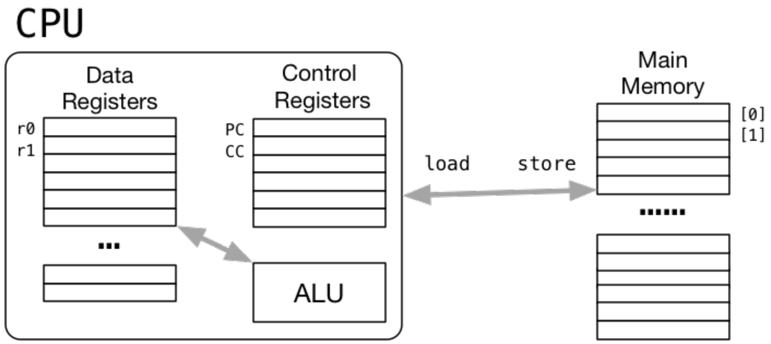
CPU Architecture

A typical modern CPU has

- a set of data registers
- a set of control registers (incl PC)
- an arithmetic-logic unit (ALU)
- access to 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

CPU Architecture



Fetch-Execute Cycle

All CPUs have program execution logic like:

```
uint32_t pc = STARTING_ADDRESS;
while (1) {
   uint32_t instruction = memory[pc];
   pc++;  // move to next instr
   if (instruction == HALT) {
      break;
   } else {
      execute(instruction);
   }
}
```

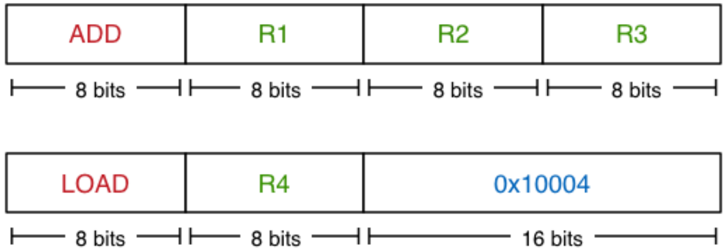
pc = Program Counter, a CPU register which tracks execution
Note that some instructions may modify pc (e.g. JUMP)

Fetch-Execute Cycle

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):



ADD | R1 | R2 | R3

8 bits | 8 bits | 8 bits | 8 bits

LOAD | R4 | 0x10004

8 bits | 8 bits | 16 bits

Assembly Language

Instructions are simply bit patterns within a 32-bit bit-string
Could specify machine code as a sequence of hex digits, e.g.

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

Assembly language is a symbolic way of specifying machine code

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

MIPS Architecture

MIPS is a well-known and simple architecture

- historically used everywhere from supercomputers to PlayStations, . . .
- still popular in some embedded fields e.g. modems, TVs
- but being out-competed by ARM (in phones, . . .)

We consider the MIPS32 version of the MIPS family

- `qtspim` ... provides a GUI front-end, useful for debugging
- `spim` ... command-line based version, useful for testing
- `xspim` ... GUI front-end, useful for debugging, only in CSE labs

Executables and source: *http://spimsimulator.sourceforge.net/*
Source code for browsing under */home/cs1521/spim*

MIPS Instructions

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

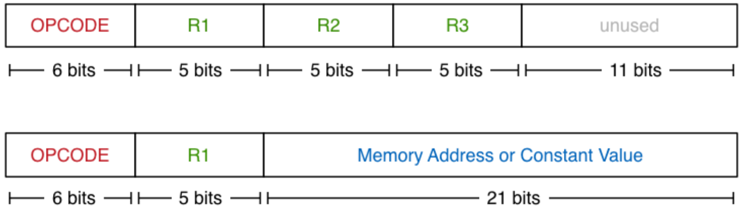
- between memory and register (direct, indirect)
- constant to register (immediate)
- register + register + destination register

MIPS Instructions

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



OPCODE	R1	R2	R3	unused
6 bits	5 bits	5 bits	5 bits	11 bits

OPCODE	R1	Memory Address or Constant Value
6 bits	5 bits	21 bits

Examples MIPS Assembler

```
lw   $t1,address   # reg[t1] = memory[address]
sw   $t3,address   # memory[address] = reg[t3]
                   # address must be 4-byte aligned
la   $t1,address   # reg[t1] = address
lui  $t2,const     # reg[t2] = const << 16
and  $t0,$t1,$t2   # reg[t0] = reg[t1] & reg[t2]
add  $t0,$t1,$t2   # reg[t0] = reg[t1] + reg[t2]
                   # add signed 2's complement ints
addi $t2,$t3, 5    # reg[t2] = reg[t3] + 5
                   # add immediate, no sub immediate
mult $t3,$t4       # (Hi,Lo) = reg[t3] * reg[t4]
                   # store 64-bit result in
                   # registers Hi,Lo
seq  $t7,$t1,$t2   # reg[t7] = (reg[t1] == reg[t2])
j    label         # PC = label
beq  $t1,$t2,label # PC = label if reg[t1]==reg[t2]
nop                # do nothing
```

MIPS Architecture

MIPS CPU has

- 32 general purpose registers (32-bit)
- 16/32 floating-point registers (for float/double)
- 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 - Integer Registers

Number	Names	Usage
0	$zero	Constant 0
1	$at	Reserved for assembler
2,3	$v0,$v1	Expression evaluation and results of a function
4..7	$a0..$a3	Arguments 1-4
8..16	$t0..$t7	Temporary (not preserved across function calls)
16..23	$s0..$s7	Saved temporary (preserved across function calls)
24,25	$t8,$t9	Temporary (preserved across function calls)
26,27	$k0,$k1	Reserved for OS kernel
28	$gp	Pointer to global area
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Return address (used by function call instruction)

- Except for registers 0 and 31, these uses are only conventions.
- Conventions allow compiled code from different sources to be combined (linked).
- Most of these conventions are irrelevant when you are writing small MIPS assembly code programs.
- But use registers 8..23 for holding values.
- Definitely do not use 0,1 and 31

MIPS Architecture - floating point registers

Reg	Notes
`$f0..$f2`	hold return value of functions which return floating-point 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:

- floating point registers can be used as 32 32-bit register or 16 64-bit registers
- for 64-bit use even numbered registers

Data and Addresses

All operations refer to data, either

- in a register
- in memory
- literally (i.e. constant)

Computation operations refer to registers or constants.
Only load/store instructions refer to memory.
To access registers, you can also use $*name*
e.g. `$zero` == `$0`, `$t0` == `$8`, `$fp` == `$30`, ...
To refer to literals, use C-like constants:

```
1  3  -1  -2  12345  0x1  0xFFFFFFFF
"a string"  'a'  'b'  '1'  '\n'  '\0'
```

Memory Addressing Modes

Ways of specifying memory addresses:

Format	Address referred to
(*reg*)	contents of register e.g. `($8) ($fp) ($5)`
imm	immediate (= constant) e.g. `0 0x80000000 123456789`
imm(*reg*)	immediate + contents of register e.g. `0($fp) 0x80000000($9)`
sym	address of symbol (= name) e.g. `main endloop exit`
sym(*reg*)	address of symbol + reg contents e.g. `main($fp) array($9)`
sym +/- imm	address of symbol +/- immediate e.g. `main+8 endloop-4`
sym +/- imm(reg)	sym address +/- (imm + reg contents) e.g. `main+8($8) array+0($18)`

Describing MIPS Assembler Operations

- An *address* refers to a memory cell.
- `Mem[`*addr*`]` = contents of cell at address *addr*.
- &*name* = location of memory cell for *name*.
- Registers are denoted:

R_d	destination register (where result goes)
R_s	source register #1 (where data comes from)
R_t	source register #2 (where data comes from)

- `Reg[`*R*`]` = contents of register *R*.
- Data transfer is denoted by <-.

```
add $6, $7, $8  #  Reg[6] <- Reg[7] + Reg[8]
lw  $7, buffer  #  Reg[7] <- Mem[buffer]
```

Setting Register

`li` R_d *imm*	*l*oad *i*mmediate Reg[R_d] <- imm
`la` R_d *addr*	*l*oad *a*ddress Reg[R_d] <- addr
`move` $R_d R_s$	*move* data reg-to-reg Reg[R_d] <- `Reg`[R_s]

Setting Register

- The *li* (load immediate) instruction is used to set a register to a constant value, e.g

```
        li $8,  42     # $8 = 42
        li $24, 0x2a   # $24 = 42
        li $15, '*'    # $15 = 42
```

- The *move* instruction is used to set a register to the same value as another register, e.g

```
        move $8, $9   # assign to $8 value in $9
```

- Note destination is first register.

Setting A Register to An Address

- Note the *la* (load address) instruction is used to set a register to a labelled memory address.

```
        la $8, start
```

- The memory address will be fixed before the program is run, so this differs only syntatctically from the *li* instruction.
- For example, if *vec* is the label for memory address `0x10000100` then these two instructions are equivalent:

```
        la  $7, vec
        li  $7, 0x10000100
```

- In both cases the constant is encoded as part of the instruction.
- Neither *la* or *li* access memory - there are very different to the *lw* instruction.

Pseudo Instructions

- Both *la* and *li* are pseudo instructions provide by the assembler for user convenience.
- The assembler translates these pseudo-instructions into instructions actually implemented by the processor.
- For example, `li $7, 15` might be translated to `addi $7, $0, 15`
- If the constant is large the assembler will need to need translate a li/la instruction to two actual instructions.

Accessing Memory

These instructions move data between memory and CPU.
1, 2 and 4-bytes (8, 16 and 32 bit) quantities can be moved.
There are two operands the register which will supply/receive the value and the memory address.
For the 1 and 2-byte operations the low (least significant) bits of the register are used.

`lw` R_d *addr*	*l*oad *w*ord (32-bits) Reg[R_d] <- `Mem[addr..addr+3]`
`sw` R_s *addr*	*s*tore *w*ord (32-bits) `Mem[addr..addr+3]` <- `Reg[`R_s`]`
`lh` R_d *addr*	*l*oad *h*alf-word (16 bits) Reg[R_d] <- `Mem[addr..addr+1]`
`sh` R_d *addr*	*s*tore *h*alf-word (16 bits) `Mem[addr..addr+1]` <- `Reg[`R_s`]`
`lb` R_d *addr*	*l*oad *b*yte (8-bits) Reg[R_d] <- `Mem[addr]`
`sb` R_d *addr*	*s*tore *b*yte (8-bits) `Mem[addr]` <- `Reg[`R_s`]`

Addressing Modes

Examples of load/store and addressing:

```
main:
    la  $t0, vec        # reg[t0] = &vec[0]
    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]
    jr   $ra            # return
    .data               # 16 bytes of storage
vec: .space   16        # int vec[4];
```

Operand Sizes

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

Arithmetic Instructions

`add` $R_d R_s R_t$	*add*ition Reg[R_d] <- `Reg[`R_s`]+Reg[`R_t`]`
`add` $R_d R_s$ *imm*	*add*ition Reg[R_d] <- `Reg[`R_s`]`+imm
`sub` $R_d R_s R_t$	*sub*traction Reg[R_d] <- `Reg[`R_s`]-Reg[`R_t`]`
`mul` $R_d R_s R_t$	*mul*tiplication Reg[R_d] <- `Reg[`R_s`] * Reg[`R_t`]`
`div` $R_d R_s R_t$	*div*ision Reg[R_d] <- `Reg[`R_s`] / Reg[`R_t`]`
`rem` $R_d R_s R_t$	*rem*ainder Reg[R_d] <- `Reg[`R_s`] % Reg[`R_t`]`
`neg` $R_d R_s$	*neg*ate Reg[R_d] <- - `Reg[`R_s`]`

All arithmetic is signed (2's-complement).
The second operand (R_t) can be replaced by a constant in all the above instructions.
Unsigned versions of instructions are available
e.g. `addu`, `subu`, `mulu`, `divu`, ...

Logic Instructions

and $R_d R_s R_t$	logical *and* Reg[R_d] <- Reg[R_s] & Reg[R_t]
and $R_d R_s imm$	logical *and* Reg[R_d] <- Reg[R_s] & imm
or $R_d R_s R_t$	logical *or* Reg[R_d] <- Reg[R_s] | Reg[R_t]
not $R_d R_s$	logical *not* Reg[R_d] <- Reg[R_s]
xor $R_d R_s R_t$	logical *xor* Reg[R_d] <- Reg[R_s] R̂eg[R_t]

All of these instructions can use *imm* instead of R_t.

Bit Manipulation Instructions

sll $R_d R_s R_t$	*s*hift *l*eft *l*ogical Reg[R_d] <- Reg[R_s]«Reg[R_t]
sll $R_d R_s imm$	*s*hift *l*eft *l*ogical Reg[R_d] <- Reg[R_s]« imm
srl $R_d R_s imm$	*s*hift *r*ight *l*ogical Reg[R_d] <- Reg[R_s]» imm
sra $R_d R_s imm$	*s*hift *r*ight *a*rithmetic Reg[R_d] <- Reg[R_s]» imm
rol $R_d R_s imm$	*ro*tate *l*eft Reg[R_d] <- rot(Reg[R_s] imm left)
ror $R_d R_s imm$	*ro*tate *r*ight Reg[R_d] <- rot(Reg[R_s] imm right)

All of these instructions can use R_t instead of *imm*.

Jump Instructions

Jumps control flow of program execution.

j *label*	*j*ump to location PC <- & label
jal *label*	*j*ump *a*nd *l*ink *ra* <- PC ; PC <- label
jr R_s	*j*ump via *r*egister PC <- Reg[R_s]
jalr R_s	*j*ump *a*nd *l*ink via *r*eg *Reg[31]* <- PC ; PC <- Reg[R_s]

Branch Instructions

Branches combine testing and jumping.

beq $R_s R_t label$	*b*ranch on *eq*ual if (Reg[R_s]==Reg[R_t]) PC <- label
bne $R_s R_t label$	*b*ranch on *n*ot *e*qual if (Reg[R_s]!=Reg[R_t]) PC <- label
blt $R_s R_t label$	*b*ranch on *l*ess *t*han if (Reg[R_s]<Reg[R_t]) PC <- label
ble $R_s R_t label$	*b*ranch on *l*ess or *e*qual if (Reg[R_s]<=Reg[R_t]) PC <- label
bgt $R_s R_t label$	*b*ranch on *g*reater *t*han if (Reg[R_s]>Reg[R_t]) PC <- label
bge $R_s R_t label$	*b*ranch on *g*reater or *e*qual if (Reg[R_s]>=Reg[R_t]) PC <- label

MIPS Instruction Set

Implementation of pseudo-instructions:

```
What you write               Machine code produced
---------------------        ---------------------

li   $t5, const              ori  $t5, $0, const

la   $t3, label              lui  $at, label[31..16]
                             ori  $t3, $at, label[15..0]

bge  $t1, $t2, label         slt  $at, $t1, $t2
                             beq  $at, $0, label

blt  $t1, $t2, label         slt  $at, $t1, $t2
                             bne  $at, $0, label
```

Note: use of `$at` register for intermediate results

MIPS vs SPIM

MIPS is a machine architecture, including instruction set
SPIM is an *emulator* for the MIPS instruction set

- reads text files containing instruction + directives
- converts to machine code and loads into "memory"
- provides debugging capabilities
  - single-step, breakpoints, view registers/memory, ...
- provides mechanism to interact with operating system (syscall)

Also provides extra instructions, mapped to MIPS core set

- provide convenient/mnemonic ways to do common operations
- e.g. `move $s0,$v0` rather than `addu $s0,$0,$v0`

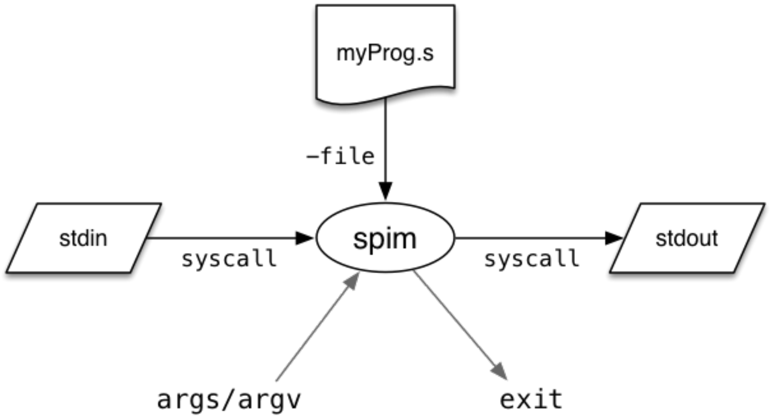
Using SPIM

Three ways to execute MIPS code with SPIM

- `spim` ... command line tool
  - load programs using `-file` option
  - interact using stdin/stdout via login terminal
- `qtspim` ... GUI environment
  - load programs via a load button
  - interact via a pop-up stdin/stdout terminal
- `xspim` ... GUI environment
  - similar to `qtspim`, but not as pretty
  - requires X-windows server

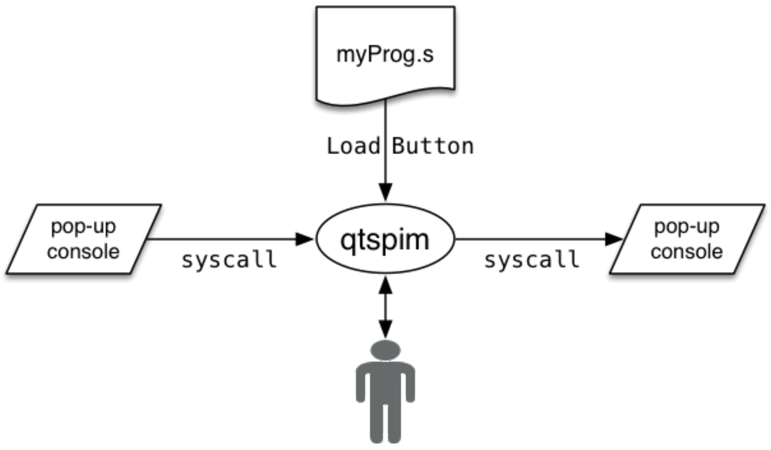
Using SPIM

Command-line tool:



Using SPIM

GUI tool:



System Calls

The SPIM interpreter provides I/O and memory allocation via the `syscall` instruction.

Service	n	Arguments	Result
`printf("%d")`	1	int in `$a0`	-
`printf("%f")`	2	float in `$f12`	-
`printf("%lf")`	3	double in `$f12`	-
`printf("%s")`	4	`$a0` = string	-
`scanf("%d")`	5	-	int in `$v0`
`scanf("%f")`	6	-	float in `$f0`
`scanf("%lf")`	7	-	double in `$f0`
`fgets`	8	buffer address in `$a0` length in `$a1`	-
`sbrk`	9	nbytes in `$a0`	address in `$v0`
`printf("%c")`	11	char in `$a0`	-
`scanf("%c")`	12	-	char in `$v0`
`exit(status)`	17	status in `$a0`	-

Also sytem calls for file I/O: open, read, write, close

MIPS (SPIM) memory layout

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

MIPS Assembly Language

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

Example MIPS assembler program

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

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)

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

Structure of Simple MIPS programs

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

main:           # indicates start of code
                # (i.e. first user instruction to execute)
                # ...

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

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

Assembler Directives

Directives (instructions to assembler, not MIPS instructions)

```
    .text       # following instructions placed in text
    .data       # following objects placed in data

    .globl      # make symbol available globally

a:  .space 18   # uchar a[18];  or  uint a[4];
    .align 2    # align next object on 2-byte addr

i:  .word 2     # unsigned int i = 2;
v:  .word 1,3,5 # unsigned int v[3] = {1,3,5};
h:  .half 2,4,6 # unsigned short h[3] = {2,4,6};
b:  .byte 1,2,3 # unsigned char b[3] = {1,2,3};
f:  .float 3.14 # float f = 3.14;

s:  .asciiz "abc" # char s[4] {'a','b','c','\0'};
t:  .ascii "abc"  # char s[3] {'a','b','c'};
```

Encoding MIPS Instructions as 32 bit Numbers

Assembler	**Encoding**
`$7 = $8 + $0`	
add `$d, $s, $t`	`000000ssssstttttddddd00000100000`
add `$7, $8, $0`	`00000000111010000000000000100000`
	0x01e80020 == 31981600
`$5 = $1 - $3`	
sub `$d, $s, $t`	`000000ssssstttttddddd00000100010`
sub `$5, $1, $3`	`00000000001000110010100000100010`
	0x00232822 == 2304034
`$2 = $2 + 1`	
addi `$d, $s, C`	`001000sssssdddddCCCCCCCCCCCCCCCC`
addi `$2, $2, 1`	`00100000010000100000000000000001`
	0x20420001 == 541196289

All instructions are variants of 3 patterns of bits.

Which simplifies silicon and human decoding (ass1!).

E.g. register numbers always in same place