COMP1521 21T3 — MIPS Basics

https://www.cse.unsw.edu.au/~cs1521/21T3/

Why Study Assembler?

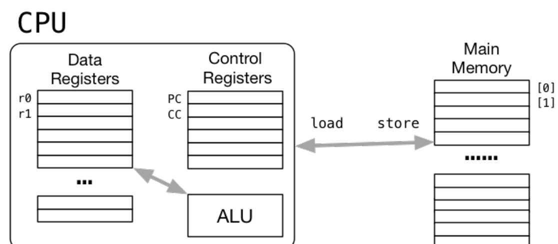
Useful to know assembly language because ...

- sometimes you are *required* to use it:
  - e.g., low-level system operations, device drivers
- improves your understanding of how compiled programs execute
  - very helpful when debugging
  - understand performance issues better
- performance tweaking ... squeezing out last pico-second
  - re-write that performance critical code in assembler!

CPU Components

A typical modern CPU has

- a set of *data* registers
- a set of *control* registers (including 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 Families Used in Game Consoles

Year	Console	Architecture	Chip	MHz
1995	PS1	MIPS	R3000A	34
1996	N64	MIPS	R4300	93
2000	PS2	MIPS	Emotion Engine	300
2001	xbox	x86	Celeron	733
2001	GameCube	Power	PPC750	486
2006	xbox360	Power	Xenon (3 cores)	3200
2006	PS3	Power	Cell BE (9 cores)	3200
2006	Wii	Power	PPC Broadway	730
2013	PS4	x86	AMD Jaguar (8 cores)	1800
2013	xbone	x86	AMD Jaguar (8 cores)	2000
2017	Switch	ARM	NVidia TX1	1000
2020	PS5	x86	AMD Zen 2 (8 cores)	3500
2020	xboxs	x86	AMD Zen 2 (8 cores)	3700

Fetch-Execute Cycle

- typical CPU program execution pseudo-code:

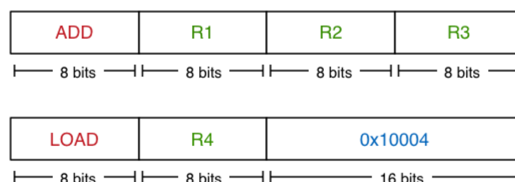
```
uint32_t program_counter = START_ADDRESS;
while (1) {
    uint32_t instruction = memory[program_counter];

    // move to next instruction
    program_counter++;

    // branches and jumps instruction may change program_counter
    execute(instruction, &program_counter);
}
```


Fetch-Execute Cycle

Executing an instruction involves:

- determine what the *operator* is
- determine if/which *register(s)* are involved
- determine if/which *memory location* 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):



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/routers, TVs
- but being out-competed by ARM and, more recently, RISC-V

COMP1521 uses the MIPS32 version of the MIPS family.

COMP1521 uses simulators, not real MIPS hardware:

- `spim` ... command-line-based simulator, good for testing
- `qtspim` ... GUI-based simulator, better for debugging
- `xspim` ... also GUI-based simulator - 1521 students prefer `qtspim`
- `mipsy` ... new `spim`/`qtspim` replacement written by Zac

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:

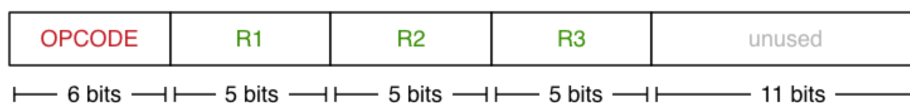
- between memory and register — **direct**, **indirect**
- constant to register — **immediate**
- register + register + destination register

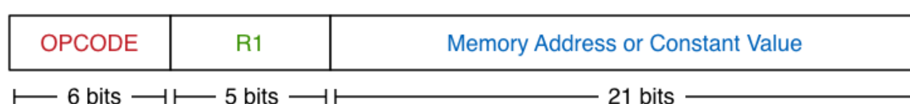
MIPS Instructions

Instructions are simply bit patterns.
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





Assembly Language

Instructions are simply bit patterns — on MIPS, 32 bits long.

- Could write machine code program just by specifying bit-patterns
  e.g as a sequence of hex digits:

  `0x3c041001  0x34020004  0x0000000c  0x03e00008`

  - unreadable! difficult to maintain!
- adding/removing instructions changes bit pattern for other instructions
- changing variable layout in memory changes bit pattern for instructions

Solution: **assembly language**, a symbolic way of specifying machine code

- write instructions using names rather than bit-strings
- refer to registers using either numbers or names
- allow names (labels) associated with memory addresses

Example 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 across 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: Registers

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, discards all written values
- registers `$1`, `$26`, `$27` reserved for use by system

More details on following slides ...

MIPS Architecture: Integer Registers

Number	Names	Conventional 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 (not 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)

MIPS Architecture: Integer Registers ... Usage Convention

- Except for registers 0 and 31, these uses are *only* programmers conventions
  - no difference between registers 1..30 in the silicon
- *Conventions* allow compiled code from different sources to be combined (linked).
- Some of these conventions are irrelevant when writing tiny assembly programs ... follow them anyway
- for general use, keep to registers $t0..$t9, $s0..$t7
- use other registers only for conventional purpose
  - e.g. only use $a0..$a3 for arguments
- *never* use registers 1, 26, 27 ($at, $k0, $k1)

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

Floating-point registers come in pairs:

- either use all 32 as 32-bit registers,
- or use only even-numbered registers for 16 64-bit registers

COMP1521 will not explore floating point on the MIPS

Data and Addresses

All operations refer to data, either

- in a register
- in memory
- a constant which is embedded in the instruction itself

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`, ...

The syntax for constant value is C-like:

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


Describing MIPS Assembly Operations

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

For example:

$$\texttt{add}\ \ \$R_d, \$R_s, \$R_t \quad \Longrightarrow \quad R_d := R_s + R_t$$

Integer Arithmetic Instructions

assembly	meaning	bit pattern
add r_d, r_s, r_t	r_d = r_s + r_t	000000ssssstttttddddd00000100000
sub r_d, r_s, r_t	r_d = r_s - r_t	000000ssssstttttddddd00000100010
mul r_d, r_s, r_t	r_d = r_s * r_t	011100ssssstttttddddd00000000010
rem r_d, r_s, r_t	r_d = r_s % r_t	pseudo-instruction
div r_d, r_s, r_t	r_d = r_s / r_t	pseudo-instruction
addi r_t, r_s, I	r_t = r_s + I	001000ssssstttttIIIIIIIIIIIIIIII

- integer arithmetic is 2's-complement.
- see also: **addu**, **subu**, **mulu**, **addiu**:
  instructions which do not stop execution on overflow.
- SPIM allows second operand (r_t) to be replaced by a constant,
  and will generate appropriate real MIPS instructions(s).

Extra Arithmetic Instructions

assembly	meaning	bit pattern
div r_s,r_t	hi = r_s % r_t; lo = r_s / r_t	000000ssssstttttt0000000000011010
mult r_s,r_t	hi = (r_s * r_t) » 32 lo = (r_s * r_t) & 0xffffffff	000000ssssstttttt0000000000011000
mflo r_d	r_d = lo	0000000000000000ddddd00000001010
mfhi r_d	r_d = hi	0000000000000000ddddd00000001001

- **mult** provides multiply with 64-bit result
- little use of these instructions in COMP1521 except challenge exercises
- pseudo-instruction **rem** r_d, r_s, r_t translated to **div** r_s,r_t plus **mfhi** r_d
- pseudo-instruction **div** r_d, r_s, r_t translated to **div** r_s,r_t plus **mflo** r_d
- **divu** and **multu** are unsigned equivalents of **div** and **mult**

Bit Manipulation Instructions

assembly	meaning	bit pattern
and r_d, r_s, r_t	r_d = r_s & r_t	000000ssssstttttddddd00000100100
or r_d, r_s, r_t	r_d = r_s | r_t	000000ssssstttttddddd00000100101
xor r_d, r_s, r_t	r_d = r_s ^ r_t	000000ssssstttttddddd00000100110
nor r_d, r_s, r_t	r_d = ~ (r_s | r_t)	000000ssssstttttddddd00000100111
andi r_t, r_s, I	r_t = r_s & I	001100ssssstttttIIIIIIIIIIIIIIII
ori r_t, r_s, I	r_t = r_s | I	001101ssssstttttIIIIIIIIIIIIIIII
xori r_t, r_s, I	r_t = r_s ^ I	001110ssssstttttIIIIIIIIIIIIIIII
not r_d, r_s	r_d = ~ r_s	pseudo-instruction

- spim translates **not** r_d, r_s to **nor** r_d, r_s, $0

Shift Instructions

assembly	meaning	bit pattern
sllv r_d, r_t, r_s	r_d = r_t « r_s	000000ssssstttttddddd00000000100
srlv r_d, r_t, r_s	r_d = r_t » r_s	000000ssssstttttddddd00000000110
srav r_d, r_t, r_s	r_d = r_t » r_s	000000ssssstttttddddd00000000111
sll r_d, r_t, I	r_d = r_t « I	00000000000tttttdddddIIIII000000
srl r_d, r_t, I	r_d = r_t » I	00000000000tttttdddddIIIII000010
sra r_d, r_t, I	r_d = r_t » I	00000000000tttttdddddIIIII000011

- **srl** and **srlv** shift zeros into most-significant bit
  - this matches shift in C of **unsigned** value
- **sra** and **srav** propagate most-significant bit
  - this ensure shifting a negative number divides by 2
- spim provides **rol** and **ror** pseudo-instructions which rotate bits
  - real instructions on some MIPS versions
  - no simple C equivalent

Miscellaneous Instructions

assembly	meaning	bit pattern
li R_d, *value*	R_d = *value*	psuedo-instruction
la R_d, *label*	R_d = *label*	psuedo-instruction
move R_d, R_s	R_d = R_s	psuedo-instruction
slt R_d, R_s, R_t	R_d = R_s < R_t	000000ssssstttttddddd00000101010
slti R_t, R_s, I	R_t = R_s < I	001010ssssstttttIIIIIIIIIIIIIIII
lui R_t, I	R_t = I « 16	00111100000tttttIIIIIIIIIIIIIIII
syscall	system call	00000000000000000000000000001100

```
# examples of miscellaneous instructions...
start:
        li      $8,  42         # $8 = 42
        li      $24, 0x2a       # $24 = 42
        li      $15, '*'        # $15 = 42
        move    $8, $9          # $8 = $9
        la      $8, start       # $8 = address corresponding to start
```


Example Translation of Pseudo-instructions

Pseudo-Instructions

```
move $a1, $v0

li   $t5, 42

li   $s1, 0xdeadbeef


la   $t3, label
```

Real Instructions

```
addu $a1, $0, $v0

ori  $t5, $0, 42

lui  $at, 0xdead
ori  $s1, $at, 0xbeef

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


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 (primitive) 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, $v0, $0`

Using SPIM

Three ways to execute MIPS code with SPIM...

`spim` ... command line tool

- load programs using `-file` option
- interact using stdin/stdout via terminal

`qtspim` ... GUI environment

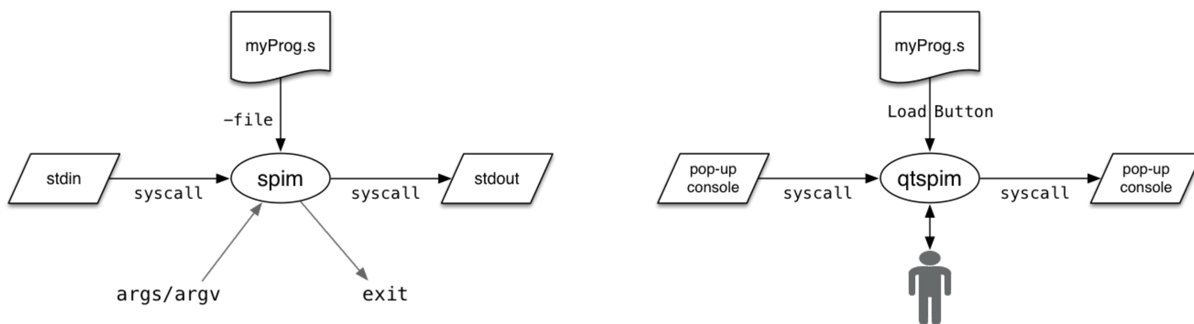
- load programs via a load button
- interact via a pop-up stdin/stdout terminal

`xspim` ... GUI environment

- similar to `qtspim`, but not as pretty

Plus we have `mipsy` zac's spim' replacement to make 1521 students life better,

Using SPIM



Using SPIM Interactively

```
$ 1521 spim
(spim) load "myprogram.s"
(spim) step 6
[0x00400000] 0x8fa40000  lw $4, 0($29)
[0x00400004] 0x27a50004  addiu $5, $29, 4
[0x00400008] 0x24a60004  addiu $6, $5, 4
[0x0040000c] 0x00041080  sll $2, $4, 2
[0x00400010] 0x00c23021  addu $6, $6, $2
[0x00400014] 0x0c100009  jal 0x00400024 [main]
(spim) print_all_regs hex
....
                General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000 ...
R1  (at) = 10010000  R9  (t1) = 00000000  R17 (s1) = 00000000 ...
```

System Calls

Our programs can't really do anything ... we usually rely on system services to do things for us.
syscall lets us make *system calls* for these services.

SPIM provides a set of system calls for I/O and memory allocation.
$v0 specifies which system call —

Service	$v0	Arguments	Returns
printf("%d")	1	int in $a0	
printf("%s")	4	string in $a0	
scanf("%d")	5	none	int in $v0
fgets	8	line in $a0, length in $a1	
exit(0)	10	status in $a0	
printf("%c")	11	char in $a0	
scanf("%c")	12	none	char in $v0

We won't use system calls 8, 12 much in COMP1521 - any input is mostly integers

System Calls ... Little Used in COMP1521

Service	$v0	Arguments	Returns
printf("%f")	2	float in $f12	
printf("%lf")	3	double in $f12	
scanf("%f")	6	none	float in $f0
scanf("%lf")	7	none	double in $f0
sbrk	9	nbytes in $a0	address in $v0
exit(status)	17	status in $a0	

Also system calls 13...16 support file I/O: open, read, write, close.

Not used in COMP1521 and rarely used by anyone.

Encoding MIPS Instructions as 32 bit Numbers

Assembler	Encoding
add $a3, $t0, $zero	
add $d, $s, $t	000000 sssss ttttt ddddd00000100000
add $7, $8, $0	000000 00111 01000 0011100000100000
	0x01e80020 (decimal 31981600)
sub $a1, $at, $v1	
sub $d, $s, $t	000000 sssss ttttt ddddd00000100010
sub $5, $1, $3	000000 00001 00011 0010100000100010
	0x00232822 (decimal 2304034)
addi $v0, $v0, 1	
addi $d, $s, C	001000 sssss ddddd CCCCCCCCCCCCCCCC
addi $2, $2, 1	001000 00010 00010 0000000000000001
	0x20420001 (decimal 541196289)

all instructions are variants of a small number of bit patterns
... register numbers always in same place

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
- instruction sequences that live in the code/text region

Each instruction or directive appears on its own line.

Our First MIPS program

C

```
int main(void) {
    printf("I love MIPS\n");
    return 0;
}
```

MIPS

```
main:
    # ... pass address of string as argum
    la   $a0, string
    # ... 4 is printf "%s" syscall number
    li   $v0, 4
    syscall
    li   $v0, 0       # return 0
    jr   $ra
    .data
string:
    .asciiz "I love MIPS\n"
```

source code for i_love_mips.s source code for i_love_mips.c