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-second)
  - re-write that performance critical code in assembler

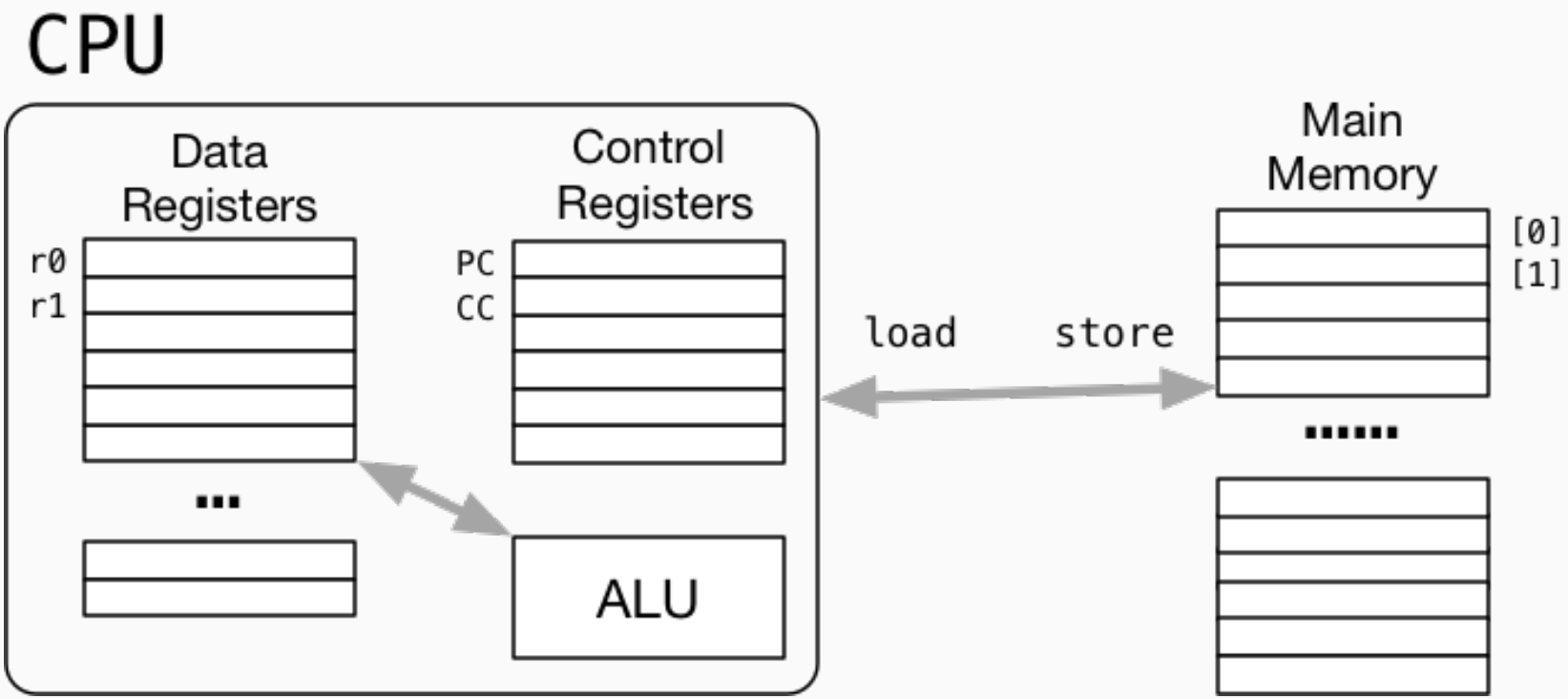
CPU Components

A typical modern CPU has

- a set of data registers
- a set of control registers (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 Layout


CPU
Data Registers
Control Registers
r0
r1
PC
CC
ALU
load
store
Main Memory
[0]
[1]

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

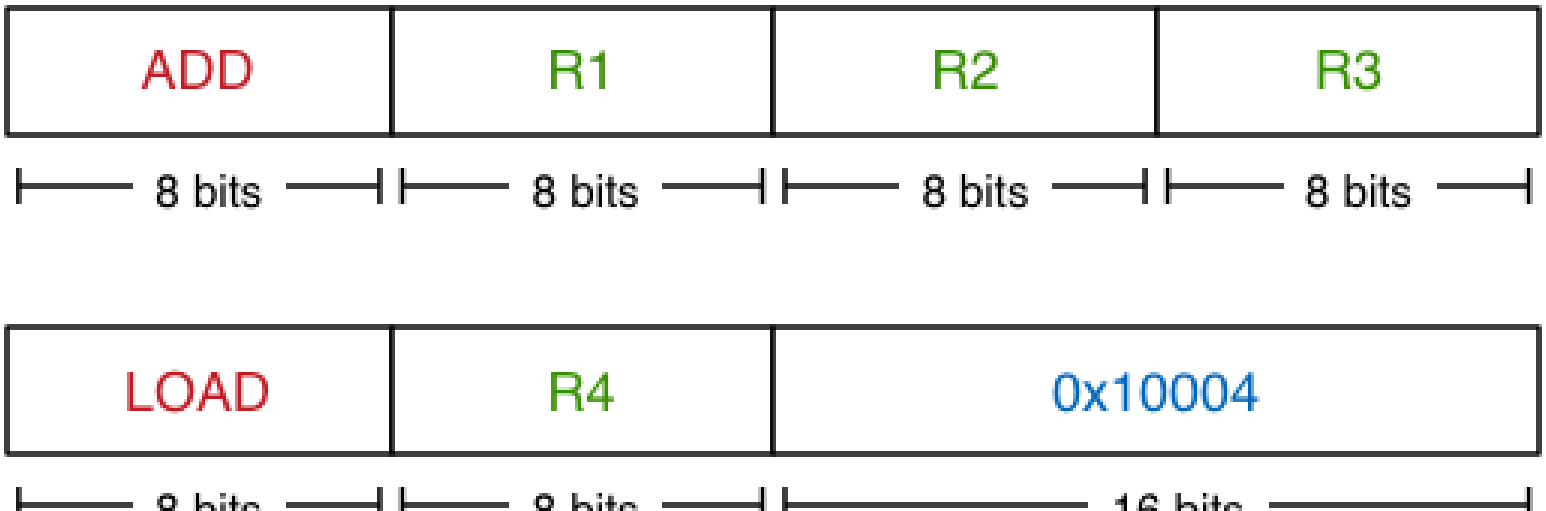
- `pc` = Program Counter, a CPU register which tracks execution
  - note some instructions modify `pc` (branches and jumps)

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):



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

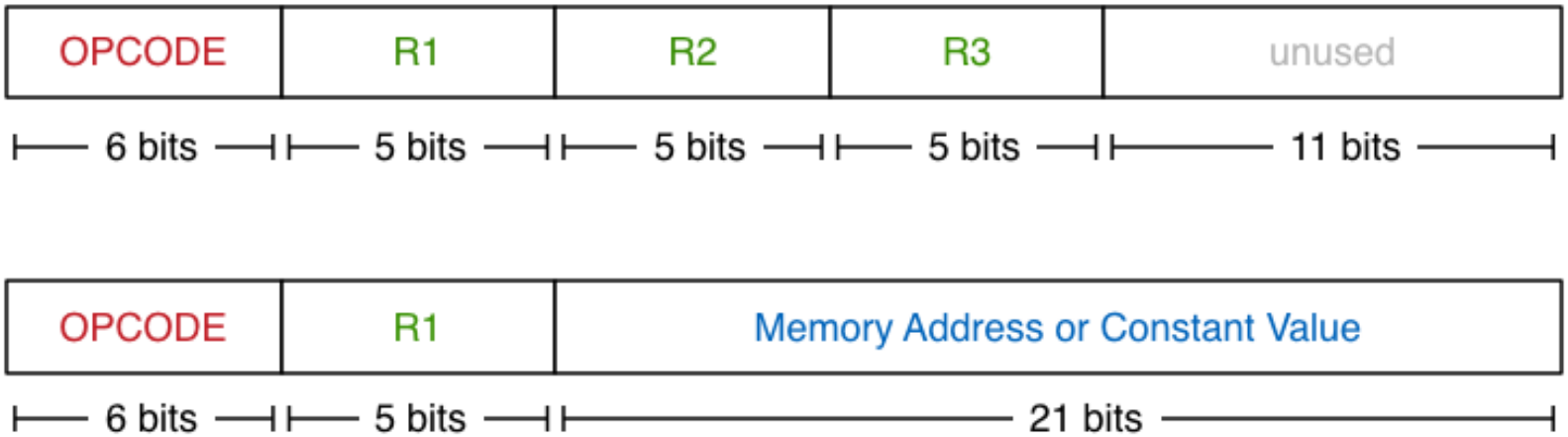
- between memory and register (direct, indirect)
- constant to register (immediate)
- register + register + destination register

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



Assembly Language

- Instructions are simply bit patterns (32-bits on the MIPS)
- Could write machine code program just by specifiying the bit-pattern e.g as a sequence of hex digits:

```
0x3c041001
0x34020004
0x0000000c
0x03e00008
```

- unreadable and 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 string
  - refer to registers using either numbers or names
  - allow names (labels) to be associated with memory addresses

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	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 (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 differenmce 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 MIPS assembly code programs - follow them anyway
- for general use keep to registers 8..25 ($t0..$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 $s0..$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

Notes:

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

add $\$R_d$, $\$R_s$, $\$R_t$ # $R_d = R_s + R_t$

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	$r_t = r_s + \mathtt{I}$	001000ssssstttttIIIIIIIIIIIIIIII

- integer arithmetic is 2's-complement.
- see also equivalent **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;	`000000ssssstttttt0000000000011010`
	`lo` $= r_s$ / r_t	
`mult` r_s,r_t	`hi` $= (r_s$ * $r_t)$ >> 32	`000000ssssstttttt0000000000011000`
	`lo` $= (r_s$ * $r_t)$ & 0xffffffff	
`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	$r_d = \sim (r_s \mid r_t)$	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	$r_d = \sim r_s$	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	$r_d = r_t << \mathtt{I}$	00000000000tttttdddddIIIII000000
`srl` r_d, r_t, I	$r_d = r_t >> \mathtt{I}$	00000000000tttttdddddIIIII000010
`sra` r_d, r_t, I	$r_d = r_t >> \mathtt{I}$	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

Jump Instructions

assem.	meaning	bit pattern
`j` *label*	pc=pc&0xF0000000|(X<<2)	000010XXXXXXXXXXXXXXXXXXXXXXXX
`jal` *label*	r_{31} = pc + 4; pc=pc&0xF0000000|(X<<2)	000011XXXXXXXXXXXXXXXXXXXXXXXX
`jr` r_s	pc = r_s	000000ssss0000000000000000001000
`jalr` r_s	r_{31} = pc + 4; pc = r_s	000000ssss0000000000000000001001

- jump instruction **unconditionally** transfer execution to a new location
- assembler will calculate correct value for X from location of *label* in code
- `jal` & `jalr` set r_{31} (`$ra`) to address of the next instruction
  - used for function calls
  - return can then be implemented with `jr $ra`

Branch Instructions

assembler	meaning	bit pattern
`beq` r_s,r_t,*label*	if (r_s == r_t) pc += I<<2	000100ssssstttttIIIIIIIIIIIIII
`bne` r_s,*label*	if (r_s != r_t) pc += I<<2	000101ssssstttttIIIIIIIIIIIIII
`ble` r_s,r_t,*label*	if (r_s <= r_t) pc += I<<2	pseudo-instruction
`bgt` r_s,r_t,*label*	if (r_s > r_t) pc += I<<2	pseudo-instruction
`blt` r_s,r_t*label*	if (r_s < r_t) pc += I<<2	pseudo-instruction
`bge` r_s,r_t*label*	if (r_s >= r_t) pc += I<<2	pseudo-instruction
`blez` r_s,*label*	if (r_s <= 0) pc += I<<2	000110sssss00000IIIIIIIIIIIIII
`bgtz` r_s,*label*	if (r_s > 0) pc += I<<2	000111sssss00000IIIIIIIIIIIIII
`bltz` r_s,*label*	if (r_s < 0) pc += I<<2	000001sssss00000IIIIIIIIIIIIII
`bgez` r_s,*label*	if (r_s >= 0) pc += I<<2	000001sssss00001IIIIIIIIIIIIII

- branch instruction **conditionally** transfer execution to a new location
- assembler will calculate correct value for *I* from location of *label* in code
- `spim` allows second operand (r_t) to be replaced by a constant and will generate appropriate real MIPS instructions(s).

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	Real Instructions
`move $a1, $v0`	`addu $a1, $0, $v0`
`li $t5, 42`	`ori $t5, $0, 42`
`li $s1, 0xdeadbeef`	`lui $at, 0xdead` `ori $s1, $at, 0xbeef`
`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`

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`

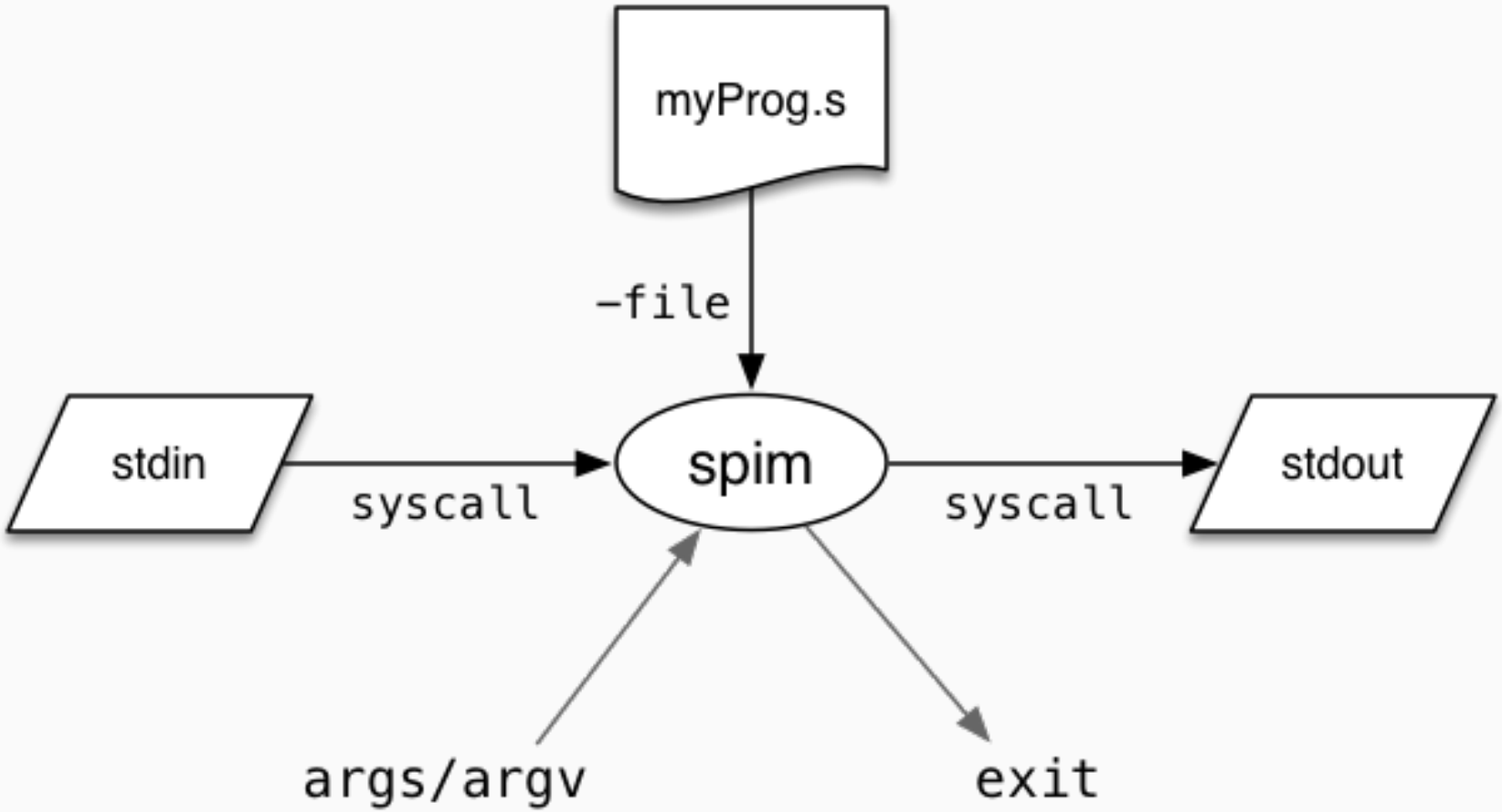
Using SPIM

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

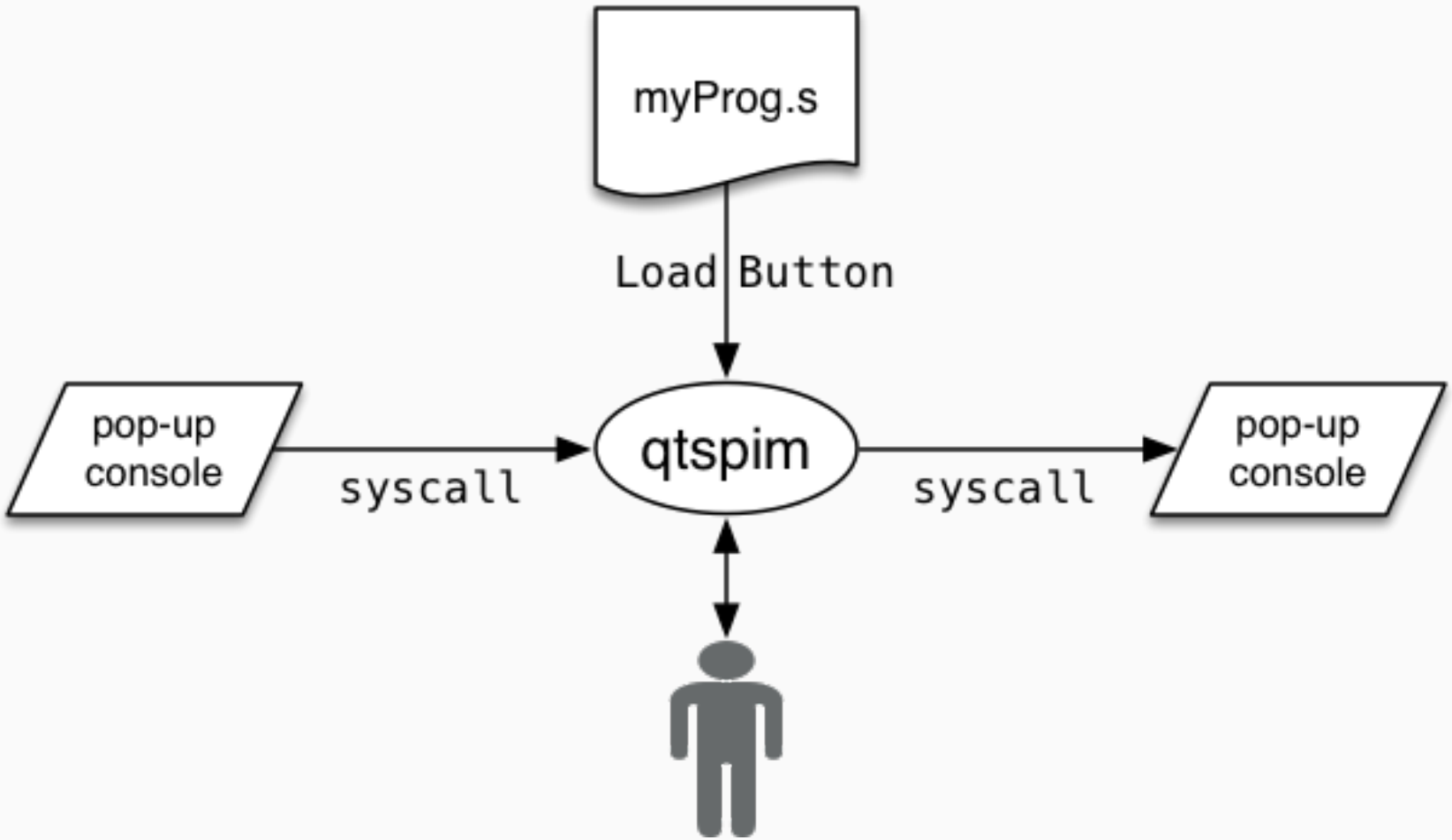
Using SPIM

Command-line tool:



Using SPIM

GUI tool:



Using Spim Interactively

```
$ 1521 spim
Loaded: /home/cs1521/share/spim/exceptions.s
(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 ...
```

Key System Calls

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

value **$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	buffer address in $a0	
		length in $a1	
`exit(0)`	10	status in $a0	
`printf("%c")`	11	char in ‘$a0	
`scanf("%c")`	12	none	char in $v0

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..17 for file I/O: open, read, write, close here

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 00000000000100000`
	0x01e80020 (decimal 31981600)
`sub $a1, $at, $v1`	
`sub $d, $s, $t`	`000000 sssss ttttt ddddd00000100010`
`sub $5, $1, $3`	`000000 00001 00011 00101000000100010`
	0x00232822 (decimal 2304034)
`addi $v0, $v0, 1`	
`addi $d, $s, C`	`001000 sssss ddddd CCCCCCCCCCCCCCCC`
`addi $2, $2, 1`	`001000 00010 00010 00000000000000001`
	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
- functions (instruction sequences) that live in the code/text region

Each instruction or directive appears on its own line

Our First MIPS program

C

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

MIPS

```
main:
    la   $a0, string  # pass address of string as argument
    li   $v0, 4       # 4 is printf "%s" syscall number
    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