COMP1521 23T1 — MIPS Basics

https://www.cse.unsw.edu.au/~cs1521/23T1/

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!
- create games in pure assembler
  - e.g., RollerCoaster Tycoon

CPU Components

A typical modern CPU has:

- a set of *data* registers
- a set of *control* registers (including PC)
- a *control unit* (CU)
- an *arithmetic-logic unit* (ALU)
- a *floating-point unit* (FPU)
- caches
  - caches normally range from L1 to L3
    - L1 is the fastest and smallest
  - sometimes separate data and instruction caches
    - eg. L1d and L1i caches
- access to *memory* (RAM)
  - Address generation unit (AGU)
  - Memory management unit (MMU)
- a set of simple (or not so simple) instructions
  - transfer data between memory and registers
  - compute values using ALU/FPU
  - make tests and transfer control of execution

Different types of processors have different configurations of the above

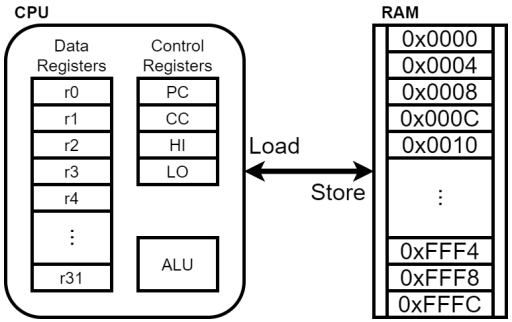
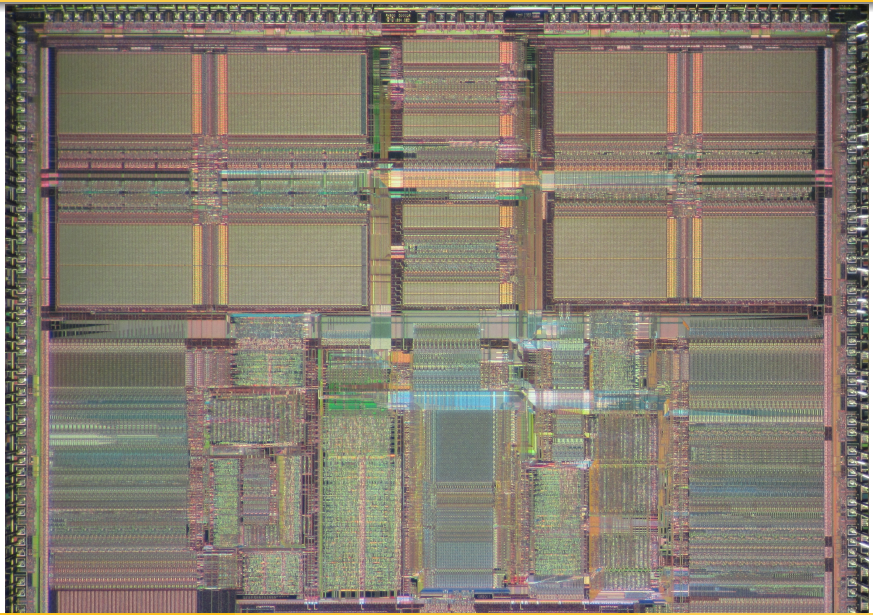


Figure 1: A Simple CPU

What A CPU Looks Like



CPU Architecture Families Used in Game Consoles

Year	Console	Architecture	Chip	MHz
1995	PS1	MIPS	R3000A	34
1996	N64	MIPS	R4200	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
2022	steam deck	x86	AMD Zen 2 (4 cores)	3500

MIPS Family

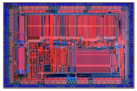
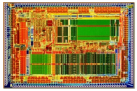
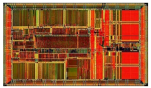
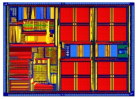
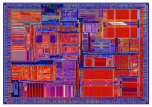
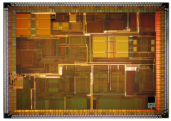
	MIPS R2000	MIPS R3000	MIPS R4000	MIPS R5000	MIPS R10000	MIPS R12000
						
Year	**1985**	**1988**	**1992**	**1996**	**1995**	**1998**
MIPS ISA	MIPS I (32-bit)	MIPS I (32-bit)	MIPS III (64-bit)	MIPS IV (64-bit)	MIPS IV (64-bit)	MIPS IV (64-bit)
Transistor count	110k	110k	2.3 – 4.6m	3.7m	6.8m	7.15m
Process node	2 µm	1.2 µm	0.35 µm	0.32 µm	0.35 µm	0.25 µm
Die size	80 mm²	40 mm²	84 – 100 mm²	84 mm²	350 mm²	229 mm²
Speed	12 – 33 MHz	20 – 40 MHz	50 - 250 MHz	150 – 266 MHz	180 – 360 MHz	270 – 400 MHz
Flagship devices	DECstation 2100 and 3100 workstations	***Sony PlayStation game console*** SGI IRIS and Indigo workstations ***NASA New Horizons space probe***	***Nintendo N64 game console*** Carrera Computers and DeskStation Technology PCs (Windows NT) SGI Onyx, Indigo, Indigo2, and Indy workstations	SGI O2 and Indy workstations Cobalt Qube servers HP LJ4000 laser printers	SGI Indigo2 and Octane workstations ***SGI Onyx and Onyx2 supercomputers*** NEC Cenju-4 supercomputers Siemens Nixdorf servers	SGI Octane 2, Onyx 2, and Origin workstations

Figure 3: MIPS Family

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 the appropriate register / memory location

Example instruction encodings
(not from a real machine):

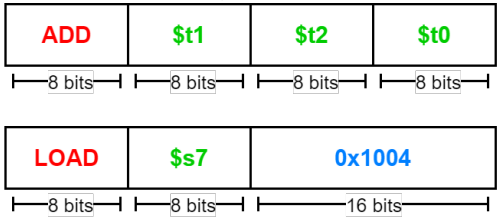


Figure 4: Fake Instructions

MIPS Architecture

MIPS is a well-known and simple architecture

- historically used everywhere from supercomputers to game consoles
- 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:

- `mipsy` ... command-line-based emulator written by Zac
  - source code: *https://github.com/insou22/mipsy*
- `mipsy-web` ... web (WASM) GUI-based version of `mipsy` written by Shrey
  - *https://cgi.cse.unsw.edu.au/~cs1521/mipsy/*

MIPS Instructions

MIPS has several classes of instructions:

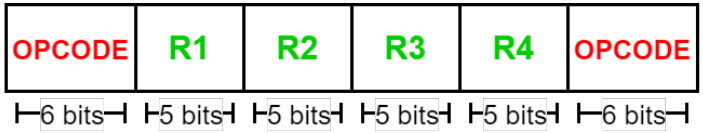
- *load and store* ... transfer data between registers and memory
- *computational* ... perform arithmetic/logical operations
- *jump and branch* ... transfer control of program execution
- *coprocessor* ... standard interface to various co-processors
  - coprocessors implement floating-point operations
  - won't be covered in COMP1521
- *special* ... miscellaneous tasks (e.g. syscall)

MIPS Instructions

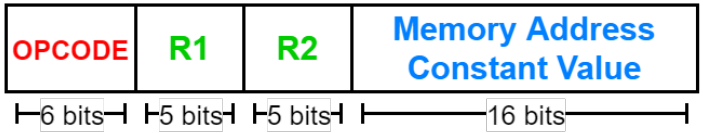
Instructions are simply bit patterns. MIPS instructions are 32-bits long, and specify ...

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

Some possible instruction formats



R-type



I-type

J-type

Assembly Language

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

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

  ```
  0x2002000b 0x20040048 0x0000000c 0x20040069 0x0000000c 0x2004000a 0x0000000c
  ```

  - unreadable!
  - difficult to maintain!
- adding/removing instructions changes bit pattern for other instructions
  - *branch* and *jump* instructions use relative offsets
- changing variable layout in memory changes bit pattern for instructions
  - *load* and *store* instructions require encoded addresses

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

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
slt     $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)
- 32/16 floating-point registers (for float/double)
  - pairs of floating-point registers used for double-precision (not used in COMP1521)
- *PC* ... 32-bit register (always aligned on 4-byte boundary)
  - modified by *branch* and *jump* instructions
- *Hi*, *Lo* ... store results of `mult` and `div`
  - accessed by `mthi` and `mflo` instructions only

Registers can be referred to as numbers (`$0`...`$31`), or by symbolic names (`$zero`...`$ra`)

Some registers have special uses:

- register `$0` (`$zero`) always has value 0, can not be changed
- register `$31` (`$ra`) is changed by `jal` and `jalr` instructions
- registers `$1` (`$at`) reserved for `mipsy` to use in pseudo-instructions
- registers `$26` (`$k0`), `$27` (`$k1`) reserved for operating-system to use in interrupts (exception handling and

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 Kernel use
28	gp	Global Pointer
29	sp	Stack Pointer
30	fp	Frame Pointer
31	ra	Return Address (used by function call instructions)

MIPS Architecture: Integer Registers ... Usage Convention

- Except for registers `zero` and `ra` (0 and 31),
  these uses are *only* programmer's conventions
  - no difference between registers 1..30 in the silicon
  - `mipsy` follows these conventions so `at`, `k0`, `k1` can change unexpectedly
- *Conventions* allow compiled code from different sources to be combined (linked).
  - *Conventions* are formalized in an *Application Binary Interface* (ABI)
- Some of these conventions are irrelevant when writing tiny assembly programs
  - follow them anyway
  - it's good practice
- for general use, keep to registers `t0`..`t9`, `s0`..`s7`
- use other registers only for conventional purposes
  - e.g. only, and always, use `a0`..`a3` for arguments
- *never* use registers `at`, `k0`,`k1`

Data and Addresses

All operations refer to data, either

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

Computation operations refer to registers or constants.

Only load/store instructions refer to memory.

The syntax for constant value is C-like:

```
1  3  -1  -2  12345  0x1  0xFFFFFFFF 0b10101010 0o123
"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:

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 (covered later in COMP1521)
- also: **addu**, **subu**, **mulu**, **addiu** - equivalent instructions which do not stop execution on overflow.
- no *subi* instruction - use *addi* with negative constant
- `mipsy` will translate **add** and of **sub** a constant to **addi**
  - e.g. `mipsy` translates **`add $t7, $t4, 42`** to **`addi $t7, $t4, 42`**
  - for readability use **addi**, e.g. **`addi $t7, $t4, 42`**
- `mipsy` allows `$r_s$` to be omitted and will use `$r_d$`
  - e.g. `mipsy` translates **`add $t7, $t1`** to **`add $t7, $t7, $t1`**
  - for readability use the full instruction, e.g. **`add $t7, $t7, $t1`**

Integer Arithmetic Instructions - Example

```
addi $t0, $zero, 6    # $t0 = 6
addi $t5, $t0, 2      # $t5 = 8
mul  $t4, $t0, $t5    # $t4 = 48
add  $t4, $t4, $t5    # $t4 = 56
addi $t6, $t4, -14    # $t6 = 42
```

Extra Integer Arithmetic Instructions (little used in COMP1521)

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	0000000000000000ddddd000000001010
mfhi r_d	r_d = hi	0000000000000000ddddd000000001001

- **mult** mutliplies and provides a 64-bit result
  - **mul** instruction provides only 32-bit result (can overflow)
- **mipsy** translates **rem** r_d, r_s, r_t to **div** r_s,r_t plus **mfhi** r_d
- **mipsy** translates **div** r_d, r_s, r_t to **div** r_s,r_t plus **mflo** r_d
- **divu** and **multu** are unsigned equivalents of **div** and **mult**

Bit Manipulation Instructions (for future reference)

- instructions explained later when we cover bitwise operators

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

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

Shift Instructions (for future reference)

- instructions explained later when we cover bitwise operators

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
- **slav** and **sla** don't exist as arithmetic and logical left shifts are the same
- mipsy 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` * 65536	`00111100000tttttIIIIIIIIIIIIIIII`
syscall	system call	`00000000000000000000000000001100`

- MIPSY allows **li** and **la** to be used interchangably
  - for readability use **li** for constants, e.g 0, 0xFF, '#'
  - for readability use **la** for labels, e.g main
- probably not needed in COMP1521, but also similar instruction/psuedo-instructions to **slt**/**slti**:
  - **sle**/**slei**, **sge**/**sgei**, **sgt**/**sgti**, **seq**/**seqi**, **sne**/**snei**
  - and unsigned versions **sleu**/**sleui**, **sgeu**/**sgeui**, **sgtu**/**sgtui**, **sequ**/**sequi**, **sneu**/**sneu**
- **mipsy** may translate pseudo-instructions to **lui**

Example Use of Miscellaneous Instructions

```
        li      $t4, 42          # $t4 = 42
        li      $t0, 0x2a        # $t0 = 42 (hexadecimail @aA is 42 decimal)
        li      $t3, '*'         # $t3 = 42 (ASCII for * is 42)
        la      $t5, start       # $t5 = address corresponding to label start
        move    $t6, $t5         # $t6 = $t5
        slt     $t1, $t3, $t3    # $t1 = 0 ($t3 and $t3 contain 42)
        slti    $t7, $t3, 56     # $t7 = 1 ($t3 contains 42)
        lui     $t8, 1           # $t8 = 65536
        addi    $t8, $t8, 34464 # $t8 = 100000
```

Example Translation of Pseudo-instructions

Pseudo-Instructions

```
move $a1, $v0

li   $t5, 42

li   $s1, 0xdeadbeef


la   $t3, label
```

Real Instructions

```
addi $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 mipsy

MIPS is a machine architecture, including instruction set

mipsy is an *emulator* for the MIPS instruction set

- reads text files containing instruction + directives
- converts to machine code and loads into "memory"
- provides some 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 Mipsy

How to to execute MIPS code without a MIPS

- `1521 mipsy`
  - command line tool on CSE systems
  - load programs using command line arguments
  - interact using stdin/stdout via terminal
- `mipsy_web`
  - *https://cgi.cse.unsw.edu.au/~cs1521/mipsy/*
  - runs in web browser, load programs with a button
  - visual environment for debugging
- `spim`, `xspim`, `qtspim`
  - older widely used MIPS simulator
  - beware: missing some pseudo-instructions used in 1521 for function calls

Using mipsy Interactively

```
$ 1521 mipsy
[mipsy] load my_program.s
success: file loaded

[mipsy] step 6

_start:
0x80000000 kernel [0x3c1a0040]    lui    $k0, 64
0x80000004 kernel [0x375a0000]    ori    $k0, $k0, 0
0x80000008 kernel [0x0340f809]    jalr   $ra, $k0

main:
0x00400000 2   [0x20020001]    addi   $v0, $zero, 1          #  li  $v0, 1
0x00400004 3   [0x2004002a]    addi   $a0, $zero, 42         #  li  $a0, 42
0x00400008 4   [0x0000000c]    syscall                       #  syscall

[SYSCALL 1] print_int: 42

[mipsy]
```

Important 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.

mipsy 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`	
`fputs`	4	string in `$a0`	
`scanf("%d")`	5	none	int in `$v0`
`fgets`	8	line in `$a0`, length in `$a1`	
`exit(0)`	10	none	
`printf("%c")`	11	char in `$a0`	
`scanf("%c")`	12	none	char in `$v0`

- We won't use system calls 8, 12 much in COMP1521 - most input will be integers

Other System Calls ... Little Used in COMP1521

- for completeness some other system calls provided by **`mipsy`**
- probably not needed for COMP1521, except could appear in challenge exercise or provided code

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 `$f0`
`sbrk(nbytes)`	9	nbytes in `$a0`	address in `$v0`
`open(filename, flags, mode)`	13	filename in `$a0`, flags in $a1, mode $a2	fd in `$v0`
`read(fd, buffer, length)`	14	fd in `$a0`, buffer in $a1, length in $a2	number of bytes read in `$v0`
`write(fd, buffer, length)`	15	fd in `$a0`, buffer in $a1, length in $a2	number of written in $v0
`close(fd)`	16	fd in `$a0`	
`exit(status)`	17	status in `$a0`	

Encoding MIPS Instructions as 32 bit Numbers

Assembler	Encoding
`add $a3, $t0, $zero`	
`add $d, $s, $t`	`000000 sssss ttttt ddddd 00000 100000`
`add $7, $8, $0`	`000000 01000 00000 00111 00000 100000`
	`0x01003820` (decimal 16791584)
`sub $a1, $at, $v1`	
`sub $d, $s, $t`	`000000 sssss ttttt ddddd 00000 100010`
`sub $5, $1, $3`	`000000 00001 00011 00101 00000 100010`
	`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

- assembly language instructions
- labels ... appended with `:`
- comments ... introduced by #
- directives ... symbol beginning with `.`
- constant definitions, equivalent of #define in C, e.g:

```
MAX_NUMBERS = 1000
```

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

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

MIPS

```
# print a string in MIPS assembly
# Written by: Andrew Taylor <andrewt@uns
# Written as a COMP1521 lecture example
main:
    la  $a0, string   # ... pass address
    li  $v0, 4        # ... 4 is printf
    syscall
    # return 0
    li  $v0, 0
    jr  $ra
    .data
string:
    .asciiz "I love MIPS\n"
```

source code for i_love_mips.s

MIPS Programming

Writing correct assembler directly is hard.

Recommended strategy:

- write,test & debug a solution in C
- map down to “simplified” C
- test “simplified” C and ensure correct
- translate simplified C statements to MIPS instructions

Simplified C

- does *not* have complex expressions
- *does* have one-operator expressions

Adding Two Numbers — C to Simplified C

C

```c
int main(void) {
    int x = 17;
    int y = 25;
    printf("%d\n", x + y);
    return 0;
}
```

source code for add.c

Simplified C

```c
int main(void) {
    int x, y, z;
    x = 17;
    y = 25;
    z = x + y;
    printf("%d", z);
    printf("%c", '\n');
    return 0;
}
```

source code for add.simple.c

Adding Two Numbers — Simple C to MIPS

Simplified C

```c
int x, y, z;
x = 17;
y = 25;
z = x + y;
printf("%d", z);
printf("%c", '\n');
```

MIPS

```
# add 17 and 25 then print the result
# Written by: Andrew Taylor <andrewt@unsw.edu.au>
# Written as a COMP1521 lecture example
main:
    # x in $t0
    # y in $t1
    # z in $t2
    li   $t0, 17        # x = 17;
    li   $t1, 25        # y = 25;
    add  $t2, $t1, $t0  # z = x + y
    move    $a0, $t2       # printf("%d", z);
    li   $v0, 1
    syscall
    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall
    li   $v0, 0         # return 0
    jr   $ra
```