DPST1092 24T3 — MIPS Basics

https://www.cse.unsw.edu.au/~dp1092/24T3/

Computer Architecture

In 1092 we run a compiler and produce a executable file.

We run a compiler (dcc or gcc)

- an executable file hello is stored on the hard drive
- We then run ./hello

What happens when you run this program? Where does it go?

How does it get executed? How does the CPU interact with hello?

Memory

Lets take this step by step

We have a hard drive

- Hard drives are refered to as secondary storage
- they usually have a large capacity
- they are non volatile memory (doesnt get erased when you turn off computer)

We have RAM(which for us is primary memory)

- primary memory
- very fast, traditionally closer to the CPU
- lower capacity

RAM

A program needs to be ‘in memory’ in order for it to run

- ‘memory’ typically refers to RAM
- Communicating between the CPU and drives is too slow

Our executiable file contains information on how to set up memory

- What instructions does the CPU need to follow?
- What strings do we need loaded into memory?

Our file is stored in RAM as 1’s and 0’s it is in machine code.

Machine code and Assembly

```
00100001000010010000000000001100
addi $t1, $t0, 12
```

Figure 1: Encoding

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!

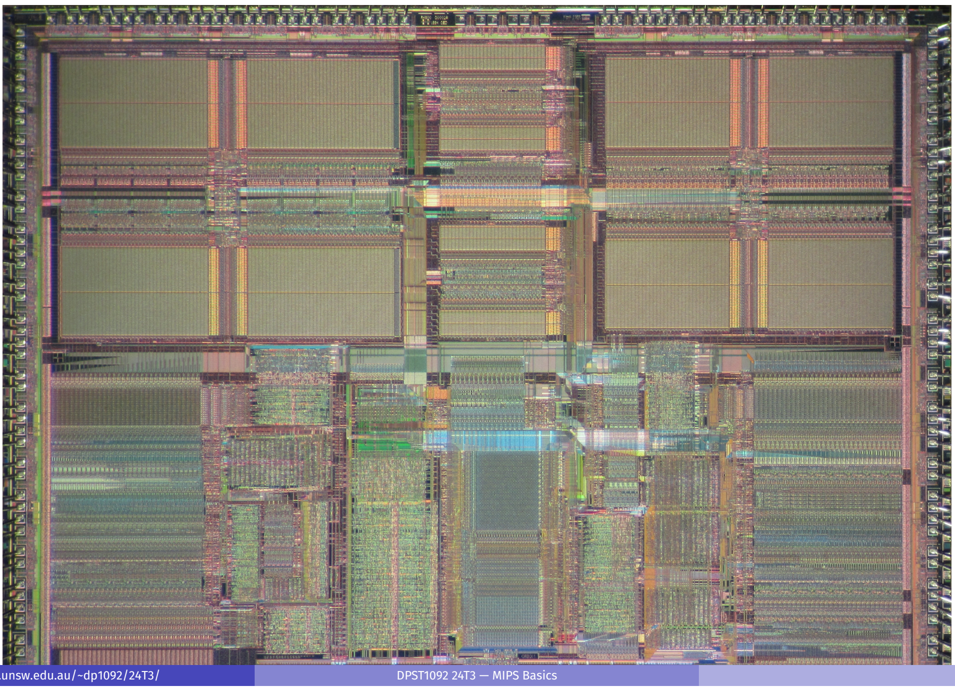
Trivia:

- there are games created in pure assembler
  - e.g., RollerCoaster Tycoon

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

What A CPU Looks Like



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)
- access to *memory* (RAM)
- 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

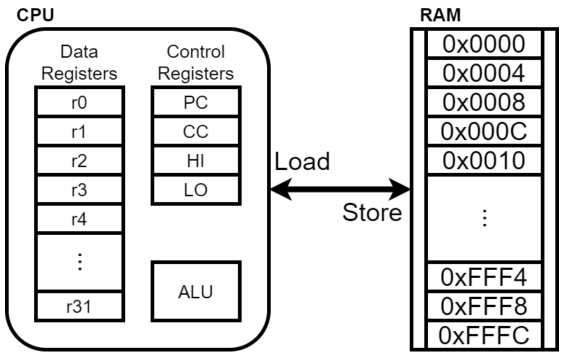


Figure 3: A Simple CPU

Different types of processors have different configurations of the above

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 / memory location

Example instruction encodings
(not from a real machine):

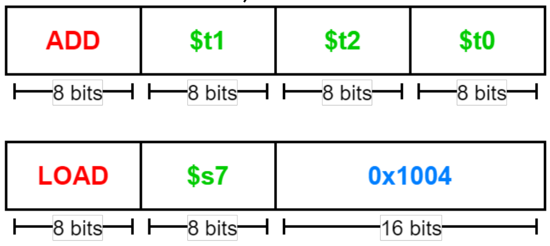


Figure 4: Fake Instructions

Assembly Language

Instructions are simply bit patterns

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

```
0x3c041001  0x34840000  0x20020004  0x0000000c  0x20020000  0x03e00008
```

  - unreadable!
  - difficult to maintain!

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

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
  - wont be covered in DPST1092
- *special* ... miscellaneous tasks (e.g. syscall)

MIPS Architecture: Registers

MIPS CPU has

- 32 general purpose registers
  - Every register that we will be dealing with is 32-bits
- Registers are located on the actual CPU which makes them very fast
  - We will be working with these registers
- The PC register is a special 32-bit register
  - used to keep track of which instruction needs to be executed next
  - modified by *branch* and *jump* instructions
- *Hi* and *Lo* store results of `mult` and `div`
  - these are used when we have a result that doesnt fit a single register
  - accessed by `mthi` and `mflo` instructions only

MIPS Architecture: Registers

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
- FOR NOW: we are going to be using (`$t0`) - (`t9`), (`$v0`) and (`$a0`)

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
- 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 purpose
  - e.g. only, and always, use `a0`..`a3` for arguments
- *never* use registers `at`, `k0`,`k1`

System Calls

None of the instructions we have access to can interact with the outside world (eg. printing, scanning)

This is done for security reasons so users cannot interact with the Operating System
We ask the operating system to perform these tasks for us - this process is called a system call

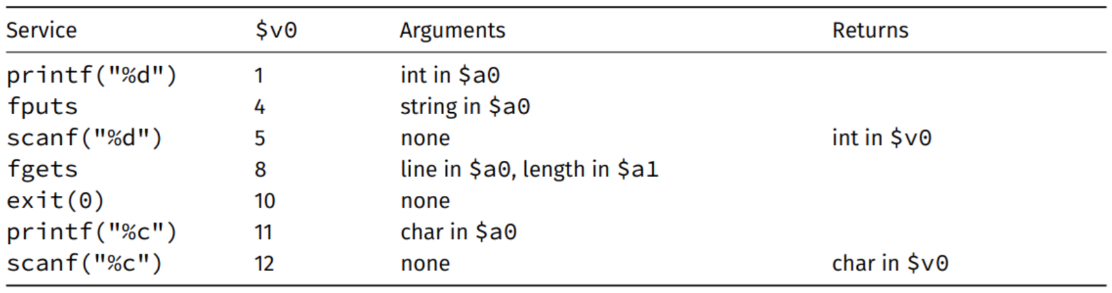
The operating system can access privileged instructions on the CPU (eg. communicating to other devices)

mipsy simulates a very basic operating system

There are many types of system calls. MIPS has several that you will use.

System calls

These are some of the most popular system calls for MIPS.

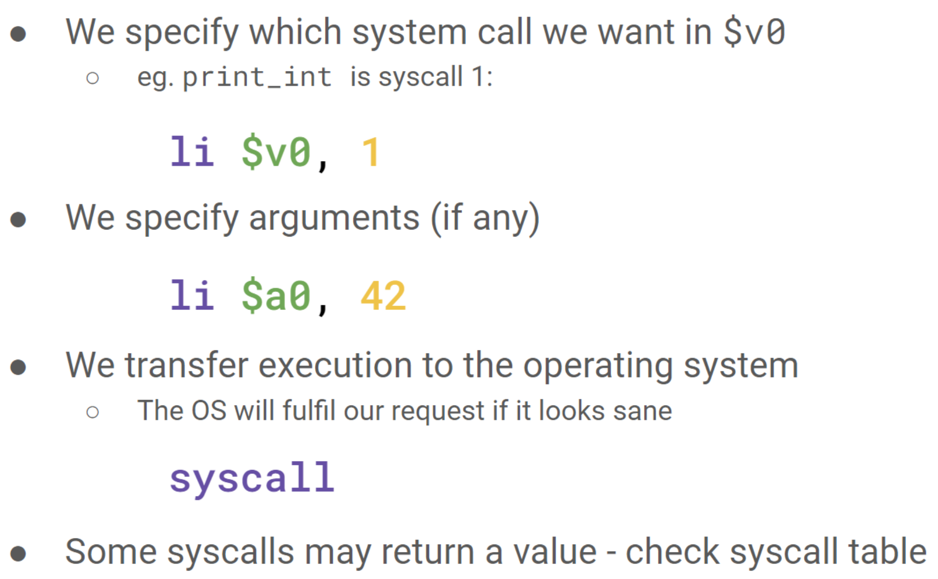


Service	$v0	Arguments	Returns
`printf("%d")`	1	int in `$a0`	
`fputs`	4	string in `$a0`	
`scanf("%d")`	5	none	int in `$v0`
`fgets`	8	line in `$a0`, length in `$a1`	
`exit(0)`	10	none	
`printf("%c")`	11	char in `$a0`	
`scanf("%c")`	12	none	char in `$v0`

Figure 5: Systems calls

System calls (How to use them)

Using system calls varies depending on what you want to do.



- We specify which system call we want in `$v0`
  - eg. `print_int` is syscall 1:

```
li $v0, 1
```

- We specify arguments (if any)

```
li $a0, 42
```

- We transfer execution to the operating system
  - The OS will fulfil our request if it looks sane

```
syscall
```

- Some syscalls may return a value - check syscall table

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

DPST1092 uses the MIPS32 version of the MIPS family.

DPST1092 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 vs mipsy

MIPS is a machine architecture, including instruction set

mipsy is an *simulator* 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

- `1092 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

Our First MIPS program

C

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

MIPS

```
# print a string in MIPS assembly
main:
    # ... pass address of string as argu
    la   $a0, string
    # ... 4 is printf "%s" syscall numbe
    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

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.

A simple MIPS Computation

```
main:
   lw     $t0, x          # $t0 = x
   addi   $t0, $t0, 4     # $t0 = x + 4
   li     $t1, 2          # $t1 = 2
   mul    $t0, $t0, $t1   # $t0 = (x+4) * 2
   sw     $t0, y          #    y = (x+4) * 2
   li     $v0, 0          # return 0
   jr     $ra

   .data
x: .word 3    # int x = 3;
y: .space 4   # int y;
```

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.

The syntax for constant value is C-like:

```
1  3  -1  -2  12345  0x1  0xFFFFFFFF 0b10101010 0123
"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:

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

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
- 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 **sub** of 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**

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, -12    # $t6 = 42
```

Extra Integer Arithmetic Instructions (little used in DPST1092)

assembly	meaning	bit pattern
div r_s,r_t	`hi` = r_s % r_t; `lo` = r_s / r_t	`000000sssssttttt0000000000011010`
mult r_s,r_t	`hi` = (r_s * r_t) » 32 `lo` = (r_s * r_t) & 0xffffffff	`000000sssssttttt0000000000011000`
mflo r_d	r_d = `lo`	`0000000000000000ddddd000000001010`
mfhi r_d	r_d = `hi`	`0000000000000000ddddd000000001001`

- **mult** provides multiply with 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

Bit Manipulation Instructions

assembly	meaning	bit pattern
and r_d, r_s, r_t	r_d = r_s & r_t	`000000sssssttttt ddddd00000100100`
or r_d, r_s, r_t	r_d = r_s | r_t	`000000sssssttttt ddddd00000100101`
xor r_d, r_s, r_t	r_d = r_s ^ r_t	`000000sssssttttt ddddd00000100110`
nor r_d, r_s, r_t	r_d = ~ (r_s | r_t)	`000000sssssttttt ddddd00000100111`
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

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

Shift Arithmatic vs Logic

Arithmatic shift is used when are dealing with a signed number.

- If 11010100 (-42 in 2s compliment) is shifted 2 bytes to the right
- We get 11110101 which is -11 in 2s compliment. (this is division by 2^n)
- When right shift it pads left most bits (MSB) with 1
- When left shift it pads right most bits (LSB) with 1

Logic shifts just simply pad both sides with 0s.

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`

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, $4    # $t1 = 0 ($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
```

Important System Calls

We often 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`

A simple system call Example

C

```
int main(void) {
    printf("%d", 42);
    return 0;
}
```

source code for print_42.s

MIPS

```
# A simple example that prints out an in
main:
    li      $v0, 1        # printf("%d",4
    li      $a0, 42
    syscall
    li      $v0, 0        # set return va
    jr      $ra           # return from m
```

Exercise: Add two numbers 1

Write MIPS assembler that behaves like

```
int main(void) {
   int x = 3;
   printf("%d\n", x+5);
   return 0;
}
```

Hints:

- li loads a constant into a register
- the number stored in $v0 determines what kind of system call it is
- syscall 1 prints the number located in register $a0
- syscall 11 prints the character located in register $a0

Exercise: Add two numbers 2

Write MIPS assembler that behaves like

```
int x = 3;
int main(void) {
   printf("%d\n", x+5);
   return 0;
}
```

Hints:

- word allocates 4 bytes in memory and initialises it
- you will need to load the value of X from RAM into a register to do the addition using lw

Exercise: Add two numbers interactively

Write MIPS assembler that behaves like

```
int main(void) {
   int x, y;
   printf("First number: ");
   scanf("%d", &x);
   printf("Second number: ");
   scanf("%d", &y);
   printf("%d\n", x+y);
   return 0;
}
```

Exercise: Find the average

Modify the code from the previous example so it implements the following:

```
int main(void) {
   int x, y;
   printf("First number: ");
   scanf("%d", &x);
   printf("Second number: ");
   scanf("%d", &y);
   printf("%d\n", (x+y)/2);
   return 0;
}
```

Exercise: Bit operations

Write the following code:

```
int main(void) {
   unsigned int x = 42;
   x = x >> 1;
   printf("%d\n",x);
   x = x << 2;
   printf("%d\n",x);
   return 0;
}
```

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 Three Numbers — C to Simplified C

C

```
int main(void) {
    int w = 3;
    int x = 17;
    int y = 25;
    printf("%d\n", w + x + y);
    return 0;
}
```

source code for add.c

Simplified C

```
int main(void) {
    int w, x, y, z;
    w = 3;
    x = 17;
    y = 25;
    z = w + x;
    z = z + 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 w, x, y, z;
w = 3;
x = 17;
y = 25;
z = w + x;
z = z + y;
printf("%d", z);
printf("%c", '\n');
```

MIPS

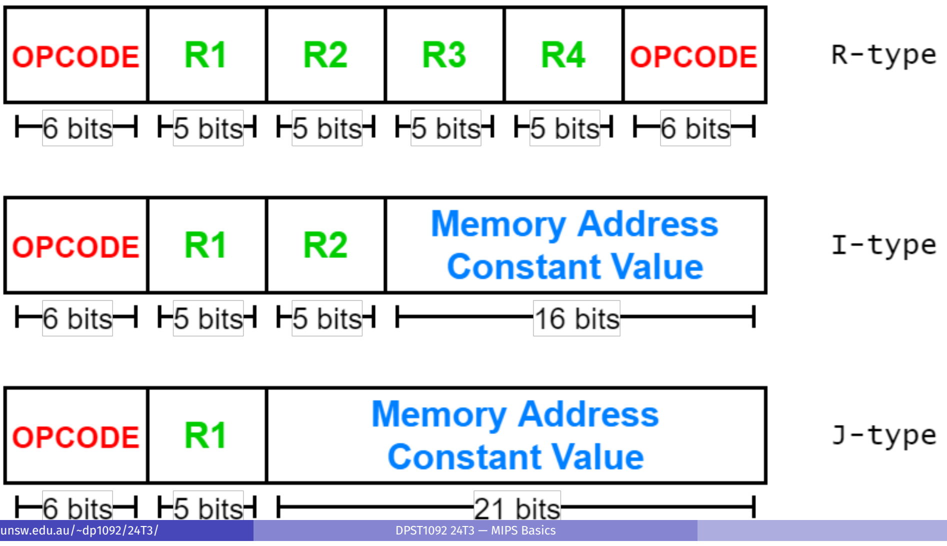
```
# add 3, 17 and 25 then print the result
main:
    # w in $t0, x in $t0
    # y in $t2, z in $t3
    li   $t0, 3         # w = 3;
    li   $t1, 17        # x = 17;
    li   $t2, 25        # y = 25;
    add  $t3, $t0, $t1 # z = w + x
    add  $t3, $t3, $t2 # z = z + y
    move $a0, $t3       # printf("%d", z);
    li   $v0, 1
    syscall
    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall
    li   $v0, 0         # return 0
    jr   $ra
```

source code for add.s

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



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 1003820)
`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

Pseudo-instructions

Pseudo-instructions are not real MIPS instructions, but are provided by mipsy for our convenience

Pseudo-Instructions

```
move $a1, $v0

li   $t5, 42

li   $s1, 0xdeadbeef


la   $t3, label
```

Real Instructions

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

addi $t5, $0, 42

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

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