



Week 24 Coursework – Assembling MIPS

Chalarampos Rotsos, Ibrahim Aref, Angie Chandler

The Task

- Implement a **MIPS Assembler Emulator**
 1. Translating assembler programs to bytecode (Week 20 – not marked).
 2. Execute the code in an emulator (Week 21/22/24 – C Coursework).
 - Similar to the MARS emulator without the UI.
 - Implement a limited set of instructions.
- Use supplied template to implement your code.
 - A half-written emulator with basic functionality.
 - Read input file, store it in memory.
 - Convert MIPS code into bytecode.

Why this Task?

- It covers all aspects of 150
 - ✓ Development is on Linux
 - ✓ We need to understand machine architectures
 - ✓ We use assembler
 - ✓ We use c
- Should show how all elements of SCC150 connect
- ... and, it is fun (hopefully)!

Generating Bytecode

Assembler

add \$s0 \$s0 \$s2



Week 20 Practical –
DONE!

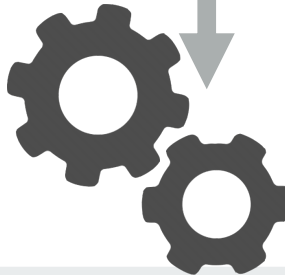
Bytecode

0x02128020

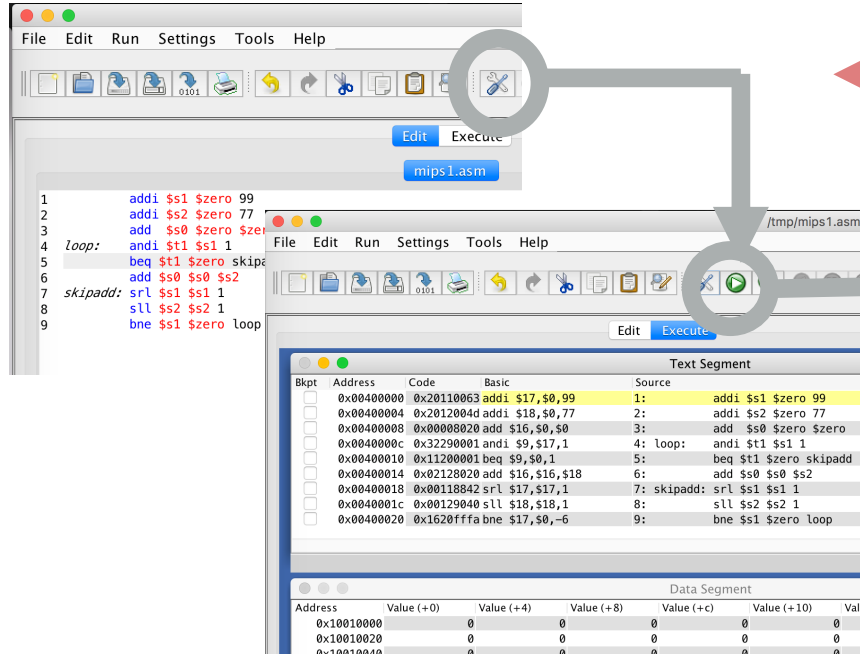


Week 24 Coursework

Execution



in MARS



Week 20
Practical

Week 24
CW

Test Code:

DATA:

0: fields: .word 88 77 0

PROGRAM:

0: lui \$t0 0x1001

1: lw \$a0 0(\$t0)

2: lw \$a1 4(\$t0)

3: add \$v0 \$zero \$zero

4: addi \$t2 \$zero 1

5: loop: andi \$t1 \$a0 1

6: bne \$t1 \$t2 skipadd

7: add \$v0 \$v0 \$a1

8: skipadd: srl \$a0 \$a0 1

9: sll \$a1 \$a1 1

10: bne \$a0 \$zero loop

11: sw \$v0 8(\$t0)

Week 20
Practical

Bytecode:

0x00400000 0x3c081001

0x00400004 0x8d040000

0x00400008 0x8d050004

0x0040000c 0x00001020

0x00400010 0x200a0001

0x00400014 0x30890001

0x00400018 0x152a0001

0x0040001c 0x00451020

0x00400020 0x00042042

0x00400024 0x00052840

0x00400028 0x1480fffa

0x0040002c 0xad020008

Execution :

executing 0x00400000 0x3c081001

executing 0x00400004 0x8d040000

executing 0x00400008 0x8d050004

executing 0x0040000c 0x00001020

Week 24
CW

...

Template main()

```
int main() {  
    if (load_program(filename)<0)  
        return(-1);  
    if (make_bytecode()<0)  
        return(-1);  
    if (exec_bytecode()<0)  
        return(-1);  
    return(0);  
}
```

Done!

Week 20
Practical

Week 24
CW

Emulator Template

- A template is provided
 - [emulator_w24_template.zip](#)
 - You do not have to start from scratch
 - Basic functionality is provided
 - Implement a MIPS CPU emulator
- Input assumptions
 - Your code must use `.data` and `.text` directives to annotate code segments.
 - The program supports only `.word` data declarations.
 - Program file contains an instruction on each line.
 - A line may include in addition a label (within the line).

MIPS ASSEMBLER – Instructions

- The emulator should support the following instructions
 - **NOP** - no operation
 - **ADD** - addition
 - **ADDI** - addition immediate
 - **ANDI** - bitwise immediate and
 - **LUI** - load upper immediate
 - **SRL** - shift right logical
 - **SLL** - shift left logical
 - **BNE** - branch on not equal
 - **LW** - Load word
 - **SW** - Store word
 - **LHU** - Load half-word unsigned
 - **SH** - Store half-word
- Any SCC150 Assembler operation is based on these instructions, the 32 register names, labels and immediate values.

Instruction Set (I)

- Basic concept
 - lowering of the compiler to the hardware level
 - not raising of hardware to the software level (as with CISC)
- MIPS is a **32-bit architecture**. This defines
 - the range of values in basic arithmetic
 - the number of addressable bytes
 - the width of a standard register
- Simple set of instructions
 - All instructions have the same 32-bit format
 - Instructions operate on 32 32-bit registers
 - Designed to run in a single clock cycle

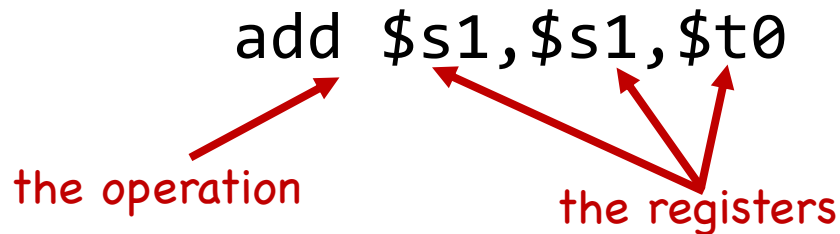
Instruction Set (II)

- 3 basic types of MIPS instructions
 - Register type (R-type)
 - Immediate type (I-type)
 - Jump type (J-type)
- What do we need to specify in an instruction?

add \$s1,\$s1,\$t0

the operation

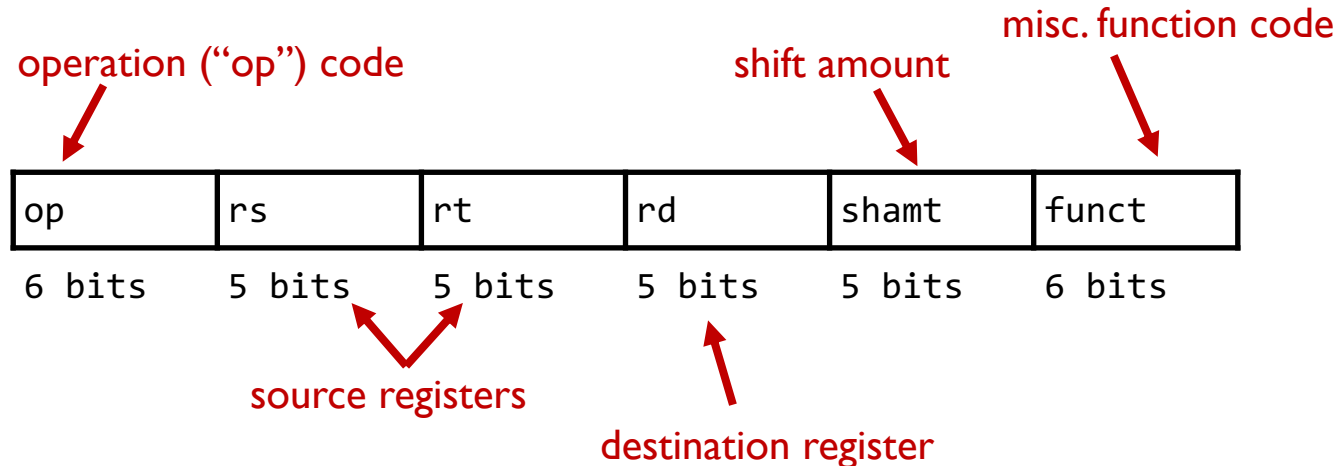
the registers



32 possible registers; $2^5 = 32$
i.e. five bits needed to specify each register

R-Type

- The elements of an R-Type instruction



R-Type Example (1)

Example instruction: addition with overflow

0x00	rs	rt	rd	0x00	0x20
------	----	----	----	------	------

Registers:

Name	Register number
\$zero	0
\$v0-\$v1	2-3
\$a0-\$a3	4-7
\$t0-\$t7	8-15
\$s0-\$s7	16-23
\$t8-\$t9	24-25
\$gp	28
\$sp	29
\$fp	30
\$ra	31

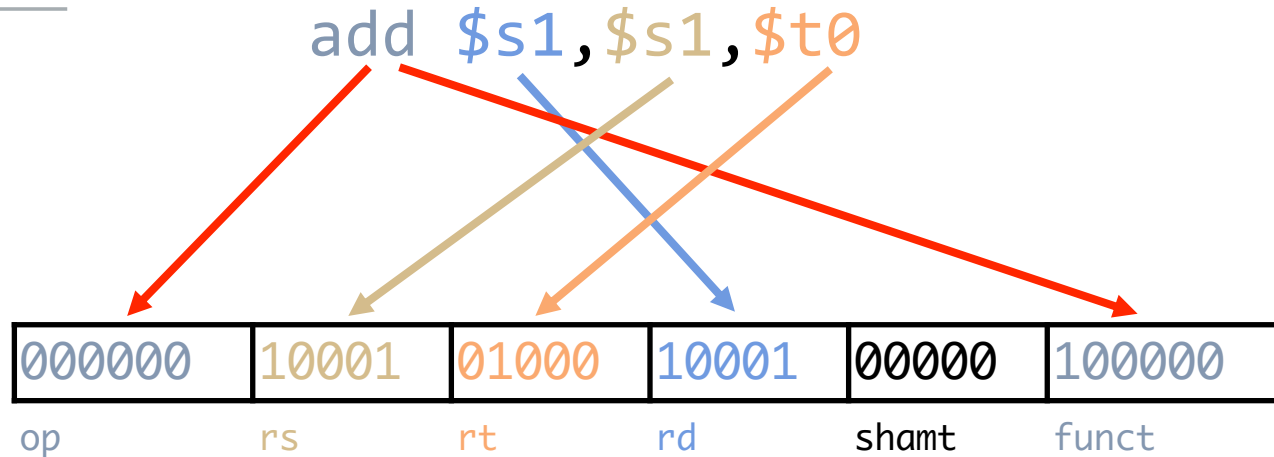
Example:

add \$s1, \$s1, \$t0



P&H fig. 2.14

R-Type Example (2)



0x02288820

I-Type

The elements of an I-Type instruction

operation (“op”) code

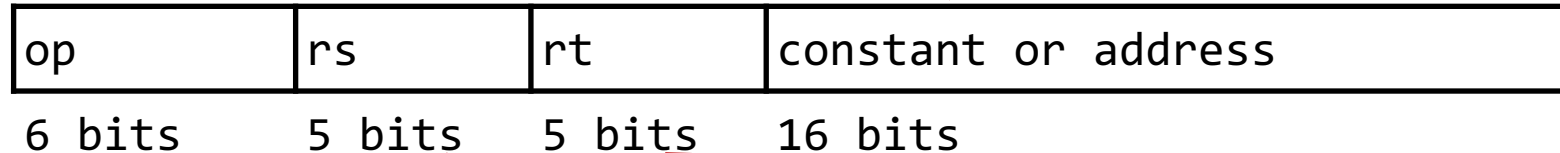
meaning depends on opcode, e.g.:

lw/sw: address offset

addi: signed number

andi: literal value

} two's
comple
nt



source register

destination register

Code Structure

- Makefile: The build file.
- emulator.h, main.c: source code file with main and preprocessor macros.
- **emulator.c**: This is the only source code file you need to modify.
- ip_csm.asm, simple_add.asm, eth_mult.asm, simple_loop.asm: sample MIPS assembly code. These files should work with the MARS emulator and should use them in order to test your code.

How to start

- This is a practical that aims to exercise your knowledge of C programming as well as your understanding of the MIPS ISA
 - Read carefully the code to understand what is going on.
 - Watch the video presentation of the code.
 - Revise slides from week 18 on MIPS instruction serialization.
 - Understand the example code and check the output of the MARS emulator.

What to do

Week 24
CW



- Write a loop to read one-by-one the instruction using the pc variable.
- Extract opcode and funct (if R-type) to figure out the instruction type.
- Implement code to extract the fields of an instruction.
- Perform appropriate changes on the registers for each instruction.
- Manipulate correctly the pc for branch operations.
- Ensure the right memory bytes are updated by sw/lw/lhu/sh.
- Use a method to terminate the program (e.g. nop).
- Comment code and use functions to improve the readability of your program.

Template Walkthrough (1)

Week 24
CW



/ function to execute bytecode */*

```
int exec_bytecode() {  
    printf("EXECUTING PROGRAM ...\n");  
    pc = ADDR_TEXT; // set program counter to the start of our program  
  
    // here goes the code to run the byte code  
  
    print_registers(); // print out the state of registers at the end of execution  
  
    print_memory(ADDR_DATA, 0x40);  
    printf("... DONE!\n");  
    return (0);  
}
```

Template walkthrough (2)

Week 24
CW



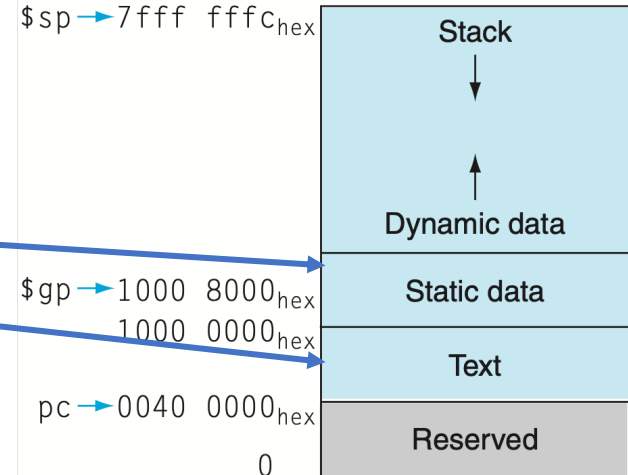
- Constants

```
/* These arrays should store the content of the text and data sections of the program *  
after the make_bytecode function. */
```

```
unsigned int data[MAX_PROG_LEN] = {0}; // the text memory with our instructions  
unsigned int text[MAX_PROG_LEN] = {0}; // the text memory with our instructions
```

```
/* Elements for running the emulator. */
```

```
unsigned int registers[MAX_REGISTER] = {0}; // the registers  
unsigned int pc = 0; // the program counter
```



Template walkthrough (3)

Week 24
CW



- Macros

```
#define ADDR_TEXT 0x00400000 //where the .text area starts in which the program lives
#define ADDR_DATA 0x10010000 //where the .text area starts in which the program lives
#define TEXT_POS(a) ((a==ADDR_TEXT)?(0):(a - ADDR_TEXT)/4) //can be used to access text[]
#define TEXT_ADDR_POS(j) (j*4 + ADDR_TEXT) //convert text index to address
#define DATA_POS(a) ((a==ADDR_DATA)?(0):(a - ADDR_DATA)/4) //can be used to access data_str[]
#define DATA_ADDR_POS(j) (j*4 + ADDR_DATA) //convert data index to address
```

Template walkthrough (4)

Week 24
CW



- Helper function to print registers and pc

```
int print_registers(){
    int i;
    printf("registers:\n");
    for(i=0;i<MAX_REGISTER;i++){
        printf(" %d: %d\n", i, registers[i]);
    }
    printf(" Program Counter: 0x%08x\n", pc);
    return 0;
}
```

Template walkthrough (4)

Week 24
CW

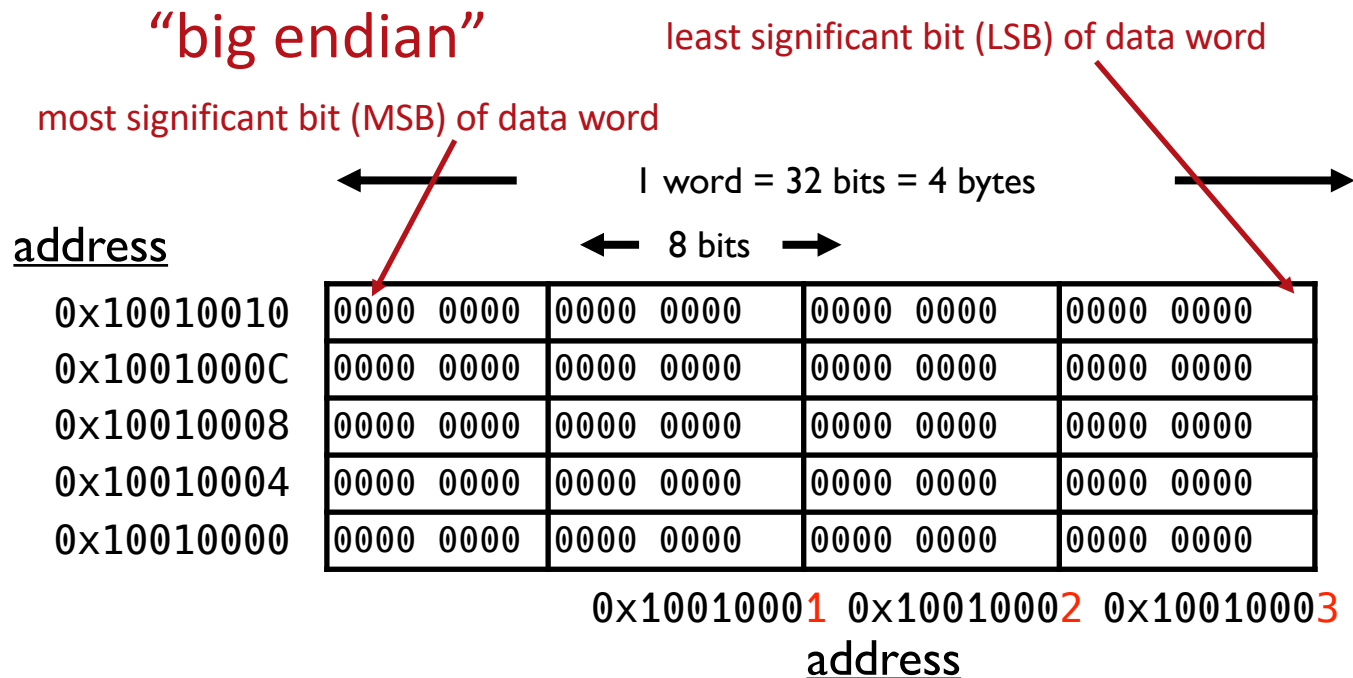


- Helper function to memory content

```
/* a function to print the state of the machine */
int print_memory(int start, int len) {
    int i, end = start + len;
    printf("Memory region (0x%08x):\n\t\t", start);
    for (i = 0; i < 8; i++)
        printf("\tValue(0x%02x)", (i*4));
    printf("\n");
    for (; start < end; start += 0x20) {
        printf("0x%08x\t", start);
        for (i = 0; i < 8; i++)
            printf("\t0x%08x", ntohl(data[DATA_POS(start + i*4)])); printf("\n");
    }
    return (0);
}
```

Word and byte addressing

Week 24
CW

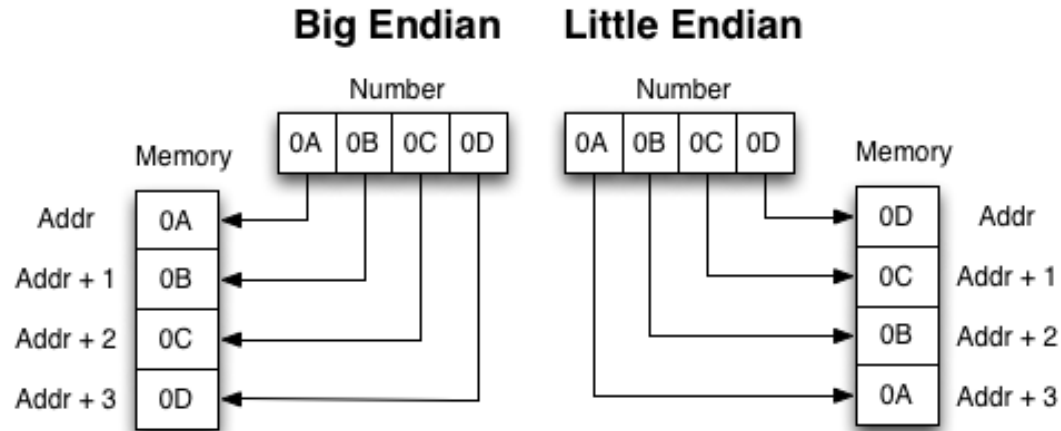


Processor Endianness

Week 24
CW



- MIPS is big-endian, Intel CPUs are little endianness.



Endianness conversion

Week 24
CW



```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);
uint32_t ntohl(uint32_t netlong);
```

- The **htonl()** function converts the unsigned integer *hostlong* from little endianness to big endianness.
 - Use it before storing to memory.
- The **ntohl()** function converts the unsigned integer *netlong* from big endianness to little endianness.
 - Use it once you read from memory.
- What happens when you read half-words (16-bits/2 bytes)?

Marking details

Week 24
CW



- Submission deadline: **Friday, 19 May 2023, 16:00.**
 - Please contact the teaching office (scc-teaching-office@lancaster.ac.uk) if you need an extension.
- You must submit:
 - emulator.c – the template file containing your code update.
 - Mark explanation – a document (docx, pdf) explaining your expected mark for functionality and code elegance/clarity/organization.

Getting support

Week 24
CW



- You have three labs until week 24 (week 21, 22, 24).
 - Ask TA/Academics any questions.
 - Be careful to not share code or solution on the chat.
- Additional sessions:
 - ***Thursday 15:00-16:00, weeks 21-25, teams-online.***
- Write you own MIPS programs to test your code..
 - One instruction per-line: `[Label:]instr $op1 $op2 $op3/imm`
- Use MARS to verify emulator correctness.
 - Does your registers output match MARS registers when running the same code?

Testing code

- Provided Makefile can help you to build code automatically.
 - In a Linux system type: make
 - Generate binary file emulator with debug information
- Run the program using: ./emulator -i filename.asm
 - Assembles file filename.asm
- Three sample asm files:
 - simple_add.asm: MIPS add program.
 - simple_loop.asm: MIPS loop program.
 - eth_mult.asm: MIPS Ethiopian multiplication (<https://www.bbc.co.uk/programmes/p00zjz5f>).
 - ip_csm.asm: A checksum generation for IP headers (https://en.wikipedia.org/wiki/Internet_checksum).

Hints

- Start from simple instruction.
 - Adding support for add, addi, andi, will give a significant number of marks.
 - Leave memory operations for last; they will be the most complex element.
- Be careful of signedness.
 - If an instruction accepts signed immediate, you will have to make sure that 16-bit signedness is not lost during translation.
- Be careful of endianness.
 - Memory in template code is big-endian, memory on your CPU is little endianness.
 - Use ntohl to read from memory with the correct signedness.
 - Use htonl before storing in memory.