Week 20 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 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

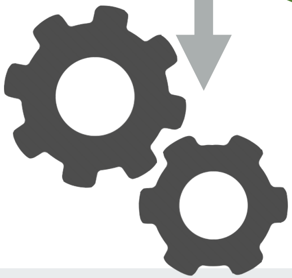
Assembler `add $s0 $s0 $s2`

Week 20 Practical

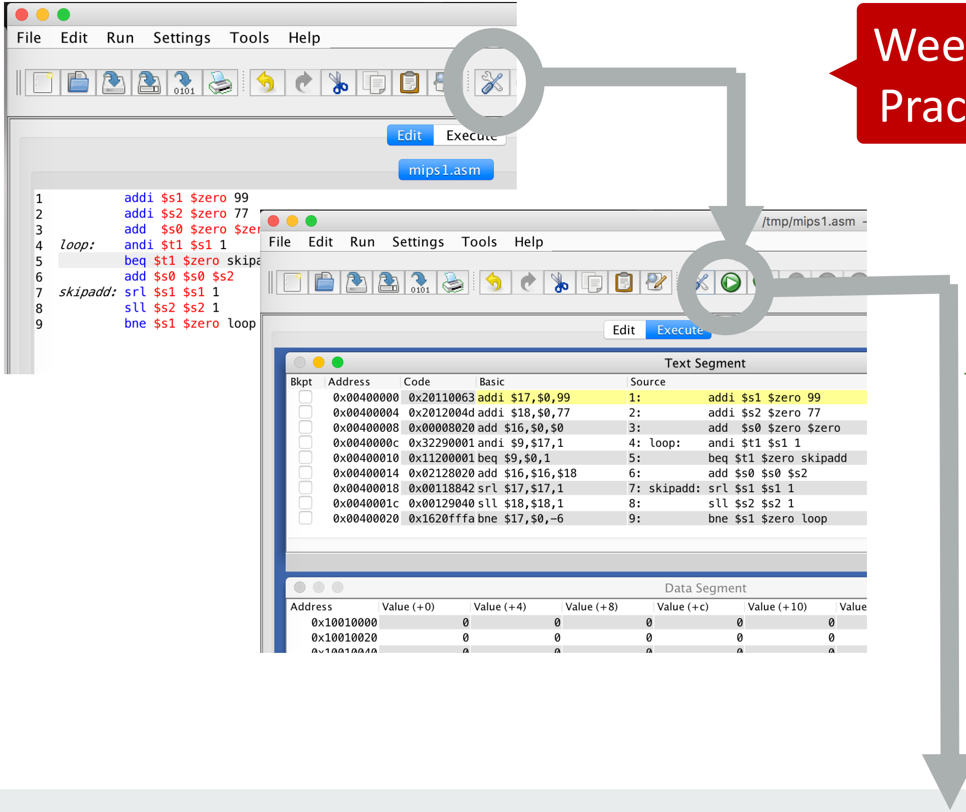
Bytecode `0x02128020`

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_w20_template.zip
  - You do not have to start from scratch
  - Basic functionality is provided
  - Fill in the gaps to obtain a working emulator

- Input assumptions
  - You 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
```

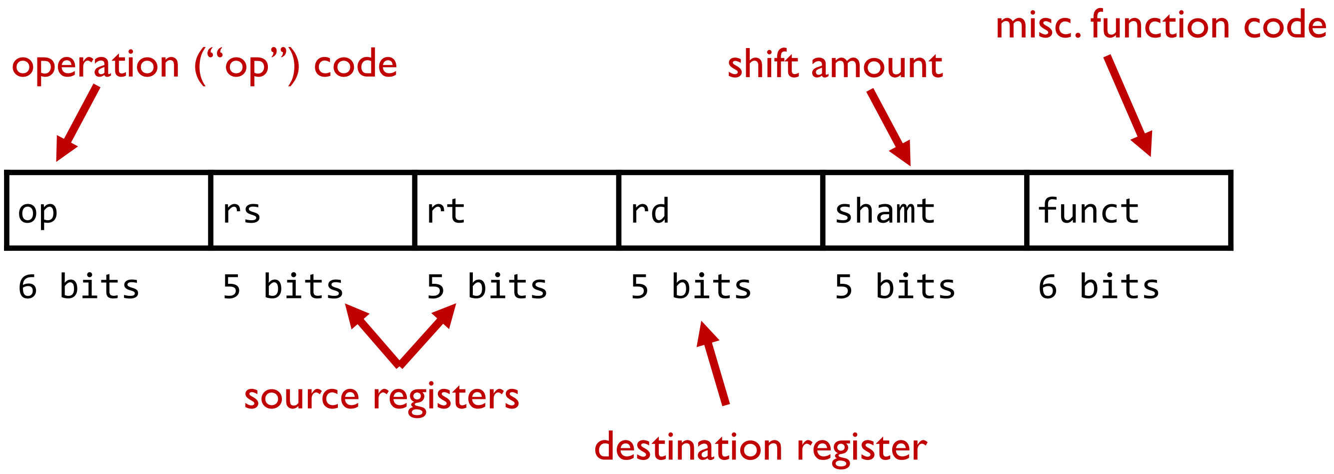
the operation

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

P&H fig. 2.14

Example:

```
add $s1,$s1,$t0
```

000000	10001	01000	10001	00000	100000
op	rs	rt	rd	shamt	funct

R-Type Example (2)

`add $s1,$s1,$t0`

000000	10001	01000	10001	00000	100000
op	rs	rt	rd	shamt	funct

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 compleme
nt

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

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.

Template walkthrough

Week 20 Practical

Lancaster University

- Constants

```
#define MAX_PROG_LEN 250
#define MAX_LINE_LEN 70
#define MAX_OPCODE   12
#define MAX_REGISTER 32
#define MAX_ARG_LEN  20


#define ADDR_TEXT    0x00400000
#define ADDR_DATA    0x10010000
```

- Registers

```
const char *register_str[] = {"$zero",
                              "$at",
                              "$v0", "$v1",
                              "$a0", "$a1", "$a2", "$a3",
```

Template walkthrough (1)

Week 20 Practical

Lancaster University

- The assembler program

```
char prog_str[MAX_PROG_LEN][MAX_LINE_LEN];
int prog_len=0; /*The length of the loaded program */
char data_str[MAX_PROG_LEN][MAX_LINE_LEN];
int data_len=0; /*The length of the loaded data section */
```

- Opcode function pointer definition

```
typedef int (*opcode_function)(unsigned int, unsigned
     int*, char*, char*, char*, char*);
```

- Arrays to simplify access to opcode functions

```
const char *opcode_str[] = {"nop", "add", "addi", "andi",
        "bne", "srl", "sll", "lui”, "lw", "sw", "lhu", "sh"};
opcode_function opcode_func[] = {&opcode_nop, NULL, NULL,
     NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL};
```

Template walkthrough (2)

Week 20 Practical

Lancaster University

- Main

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

- Load program (to be placed in prog_str[ ][ ] and data_str[][])

```
int load_program( char *filename){
   …
   while (fgets(&prog[prog_len][0], MAX_LINE_LEN, f) != NULL)
        prog_len++;
   …
}
```

Template walkthrough (3)

Week 20 Practical

Lancaster University

- Making bytecode

```
int make_bytecode() {
    unsigned int bytecode;
            …
  while (j < prog_len) {
    sscanf(&prog[j][0], …
       …
      (*opcode_func[i])(j, &bytecode, opcode, arg1, arg2, arg3)
      …
      }
    …
  return 0;
}
```

Space to hold the 32 bit instruction

Runs through each line of the program sequentially

sscanf to break each line into tokens

call the appropriate function that matches the opcode string

bytecode is passed as pointer; the 32 bit instruction will be returned here

What to Implement ?

Week 20 Practical

Lancaster University

- Implement methods to convert string representation of assembly instruction into 32-bit representations.
  - Start simple and implement one-by-one the instructions.
  - Your code should be implemented in make_bytecode().
  - The default code use the opcode_nop to serialize all instruction; implement your custom serializer for all other instructions.

PART II: Hints

Lancaster University

- Instruction: `add $s1 $s1 $t0`

0x00	rs	rt	rd	0x00	0x20

- First 6 bits are 0x00; last 11 bit are 0x20:

```
*bytecode = 0x20;      //instruction: 0x00000020
```

- Destination is $s1, register number is:

```
if(!strcmp(reg,register_str[i])){ … } // i = 17
```

- Destination (rd) starts at bit 11:

```
*bytecode |= i << 11; //instruction: 0x00008820
```

- After adding rs (starts bit 21) and rt:

```
instruction: 0x02288820
```

Assembler example (1)

Week 20 Practical

Lancaster University

```
#include <stdio.h>
#include <string.h>
#define MAX_REGISTER 32
const char *register_str[] = {"$zero",
                              "$at",
                              "$v0", "$v1",
                              "$a0", "$a1", "$a2", "$a3",
                              "$t0", "$t1", "$t2", "$t3", "$t4", "$t5", "$t6", "$t7",
                              "$s0", "$s1", "$s2", "$s3", "$s4", "$s5", "$s6", "$s7",
                              "$t8", "$t9",
                              "$k0", "$k1",
                              "$gp ", " $sp ", " $fp ", " $ra "};

int main() {
    unsigned int bytecode; // this will at the end contain our result: 0x02288820
    int i = 0;             // counter for our loops

    /* the line is "add $s1 $s1 $t0", it is already broken down into tokens */
    char *label = "";
    char *opcode = "add";
    char *arg1 = "$s1";
    char *arg2 = "$s1";
    char *arg3 = "$t0";

    printf("executing line: |%s|%s|%s|%s|%s|\n", label, opcode, arg1, arg2, arg3);
```

Assembler example (2)

Week 20 Practical

```
if(!strcmp(opcode, "add")){ // do add, we can only do this here with this program
    // step1: First 6 bits are 0000000 and last 11 bit are 00000100000
    // -> 00000000 00000000 00000000 00100000 -> 0x00000020 -> 0x20
    bytecode = 0x20;
    // step2: translate destination register into a number (one of 32)
    for(i=0;i<MAX_REGISTER;i++){
        f(!strcmp(arg1,register_str[i])) break;
    }
    // $s1 -> reg No. 17 -> 10001
    // 17 needs to start at bit 11, not 0 -> use shift
    // 17 << 11 is 00000000 00000000 10001000 00000000
    // then use OR:
    // 00000000 00000000 00000000 00100000
    // 00000000 00000000 10001000 00000000
    // -----------------------------------
    // 00000000 00000000 10001000 00100000 -> 0x00008820
    bytecode |= i << 11;
    printf("bytecode: 0x%08x\n",bytecode);

    // step3: translate source 1 register into number (same as above)

    // step4: translate source 2 register into number (same as above)

    // result: 0x02288820
}
```

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).

Test Code: Ethiopian multiplication

- Multiply a and b (for example, a=17 b=34)
- Create 2 columns
- Halve a until reaching 1; double b
- Add 2nd column if a is odd

```
17      34
 8     (68) <-- even
 4    (136) <-- even
 2    (272) <-- even
 1     544
      -----
       578
```

- This can be implemented using assembler ...

Getting help on week 20

- 2xDrop-in (face-to-face and online) sessions
  - Wednesday 22/3 13:00-15:00, B076
  - Thursday 23/3 12:00-14:00, B076
- 1xDrop-in (online only) session
  - Wednesday 22/3 15:00-16:00
- Teams event

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.