ECS404U: Computer Systems & Networks

Week 8: More into MIPS ISA: Memory Layout, Data Declaration, Branching instructions, I/O syscalls

Prof Edmund Robinson, Dr Arman Khouzani

November 09, 2020

EECS, QMUL

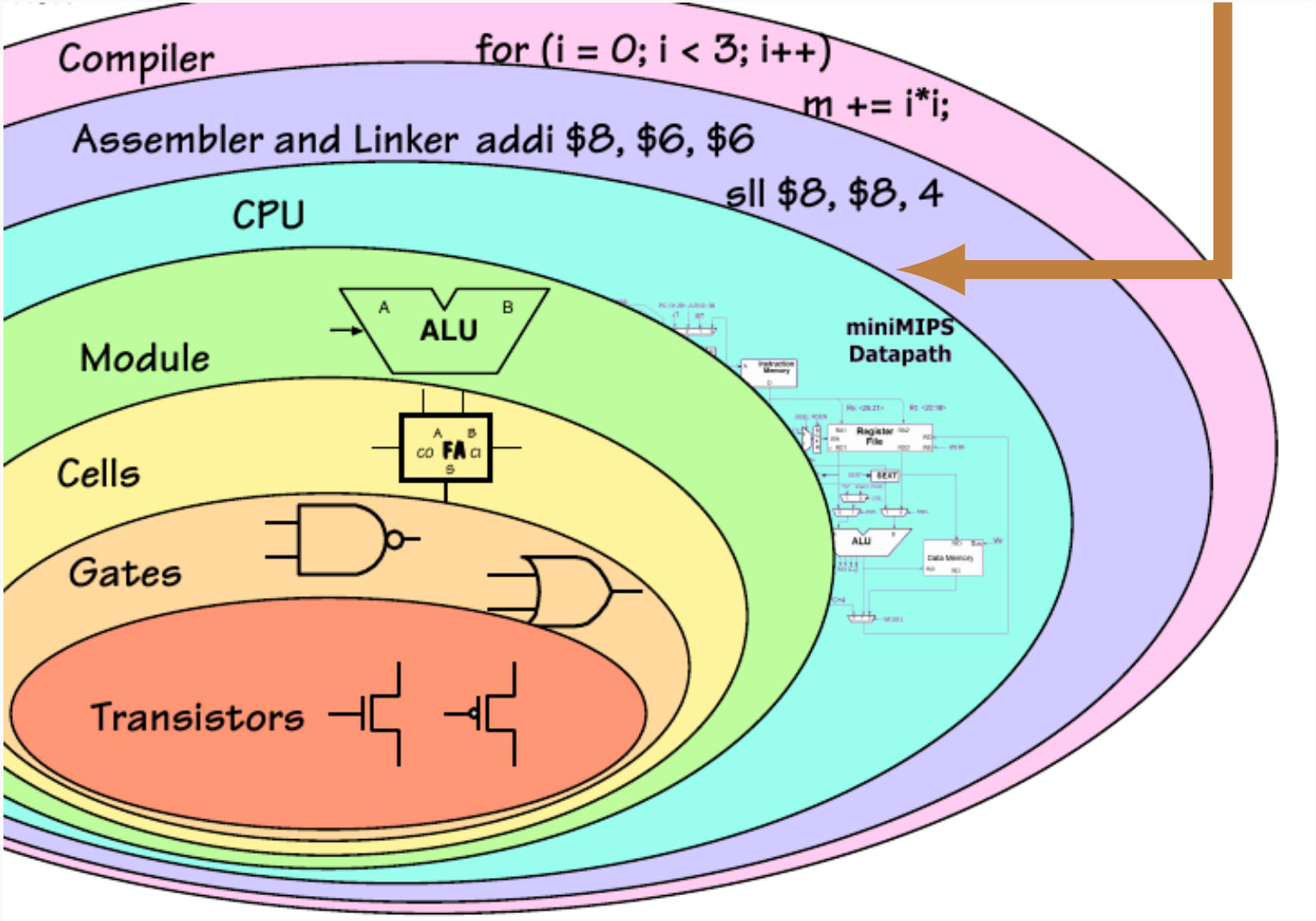
Agenda/Objectives

- Controlling the program flow: Sequencing (Jump and branching instructions)
- Understanding a simplified “memory layout” of a computer
  - Similarities and differences between instructions and data in the memory;
  - Declaring data in an assembly programme;
  - Memory Access (read and write);
  - Strings and Arrays;
- Simple I/O and syscalls
- Some practice MIPS assembly examples

A Quick Recap (where we are)

Machine Codes, Instruction Set Architecture

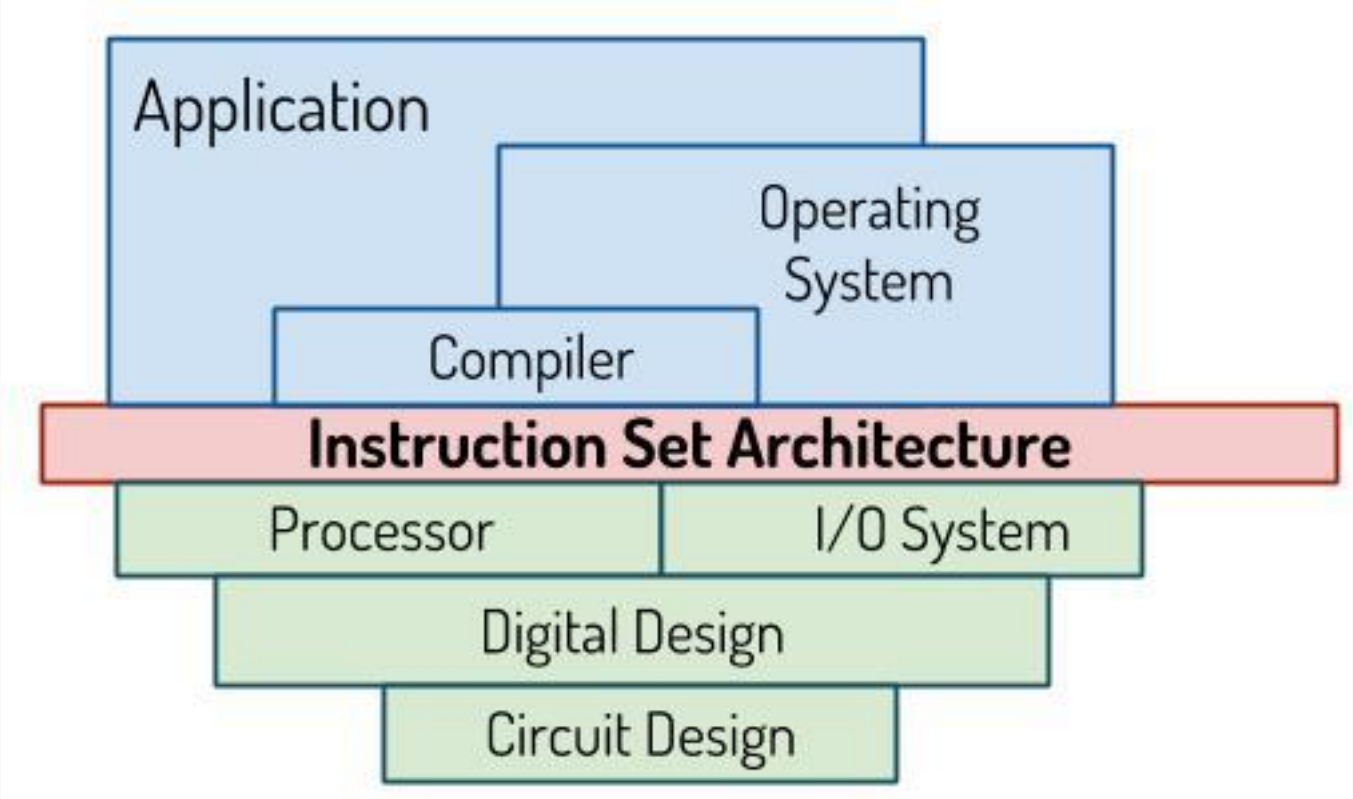
ISA: *the interface between hardware and software*



Machine Codes, Instruction Set Architecture

- A given microprocessor has implemented a set of operations (e.g. addition, bitwise logics, etc.)
- These operations along with the available registers & modes of accessing memory (but without the detail of how they are implemented in hardware) constitute the **Instruction Set Architecture** of a processor.
- ISA provides an an abstract computer for writing programmes that can directly run on the hardware.
- A **RISC** is an ISA that has a simpler and more efficient set of instructions (e.g. `MIPS`, `ARM`), unlike a **CISC** ISA (e.g. `x86`) which also involves complex instructions that take many clock cycles to perform.

Machine Codes, Instruction Set Architecture



Instruction Set Architecture provides a simplified but sufficient description of the hardware interface, so that the programmes that are translated to it can run directly on the hardware.

MIPS instruction set architecture

- The 32-bit MIPS machine instructions come in 3 formats: `R`, `I`, and `J`;
  - `R`: has references to three registers: e.g. some operation to be done on the content of two registers and the result be stored in the third register.
  - `I`: has reference to two registers and has an "immediate" value: e.g. some operation to be done on the content of one register and the immediate value, then the result be stored in the other register;
  - `J`: has no reference to any register, it only has an opcode and a constant value;
- In all three formats, the first 6 bits encodes the **opcode**, which specifies which operation to be done.

MIPS instruction set architecture: Registers

- MIPS has **32 general-purpose registers**, each of which are **32 bits wide**;
- The first register (register `0` or `$zero`) always has the constant all zeros (useful for moving data between registers, comparison instructions, etc);
- These 32 registers are indexed from `0` to `31`: in the 32-bit machine instruction, we designate which register through its number (with 5 bits), but in the MIPS Assembly language, we also have mnemonics for them, e.g. register `0` is represented by `$zero`, register `8` by `$t0`, and so on.

MIPS instruction set architecture: Registers

- Besides these general purpose registers, there are a few registers that are crucial for the working of the processor itself. Most notably:
  - `PC`: *Program Counter*, a (32-bit) register: contains the (memory) address of the next instruction to be fetched from memory;
  - `IR`: *Instruction Register*, a (32-bit) register: contains the most recently fetched instruction.
- The programmer does not have direct access to these registers. But then, how can we manipulate the flow of execution?

Branching and Jumps

Sequencing Instructions

The microprocessor fetches (from the main memory, i.e., RAM) the instruction whose address is determined by the content of the `PC` register.

- Normally, after execution of each instruction, the content of the `PC` gets incremented by **4**, to indicate the address of the next instruction in the memory;
  - *Question: why does PC get incremented by 4?*

Sequencing Instructions

The microprocessor fetches (from the main memory, i.e., RAM) the instruction whose address is determined by the content of the `PC` register.

- Normally, after execution of each instruction, the content of the `PC` gets incremented by **4**, to indicate the address of the next instruction in the memory;
  - ▶ *Question: why does PC get incremented by 4?*
  - ▷ *Answer: in a 32-bit architecture, each instruction is 32 bits long, which is 4 bytes. Each byte of the memory has its own address, so the address of the (starting byte of the) next instruction is 4 plus the address of the (starting byte of the) current instruction.*

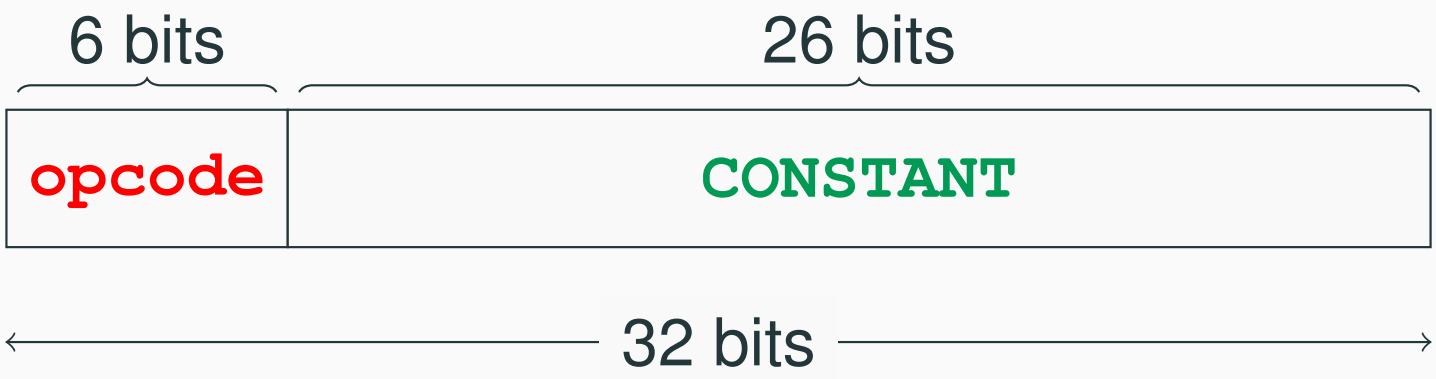
Sequencing Instructions

- ▷ However, no interesting program follows this simple sequence (think e.g. “sorting”, or any “loop”)
  - We need to be able to control the flow of the program (e.g. based on the data, etc.);
- ▶ This is why we need “sequencing instructions” like jumps and branching.

MIPS Sequencing Instructions: Jump

The simplest instruction to alter the default sequence of instructions is the jump instruction:

- `j <label>`



The J-format.

- *Effect:* loads the `PC` register with the address of the instruction that we have designated by a label (in our assembly code, using a colon).

MIPS Sequencing Instructions: Jump

[details Not part of assessment!]

- *Question: Recall that each address is 32-bits (`PC` is a 32-bit register), but the `CONSTANT` in the `j` instruction is 26 bits long (it cannot be longer, why?). How do we get to a 32-bit address from a 26-bit constant?*
  - *Answer:* Two zeros are added to the right of it (why?) and for the remaining 4 bits, it uses the high order four bits (four leftmost bits) of whatever is in the PC!
- *Question: Explain why using `j` instruction, you cannot jump to any 32-bit address in the memory. What range can you go to?*

MIPS Sequencing Instructions: Branching

With only a jump instruction, you can at most have an infinite loop! But what about controlling the flow of the program based on some condition, i.e., **branching**? (*Q: incidentally, why is it called branching?*)

MIPS Sequencing Instructions: Branching

With only a jump instruction, you can at most have an infinite loop! But what about controlling the flow of the program based on some condition, i.e., **branching**? (*Q: incidentally, why is it called branching?*)

- Branch-on-equal (branch-if-equal):

  `beq rs, rt, label`

  - compares the content of registers **`rs`** and **`rt`**, and if they are equal, executes the sequence of instructions starting from the line designated by **`label`**.

MIPS Sequencing Instructions: Branching

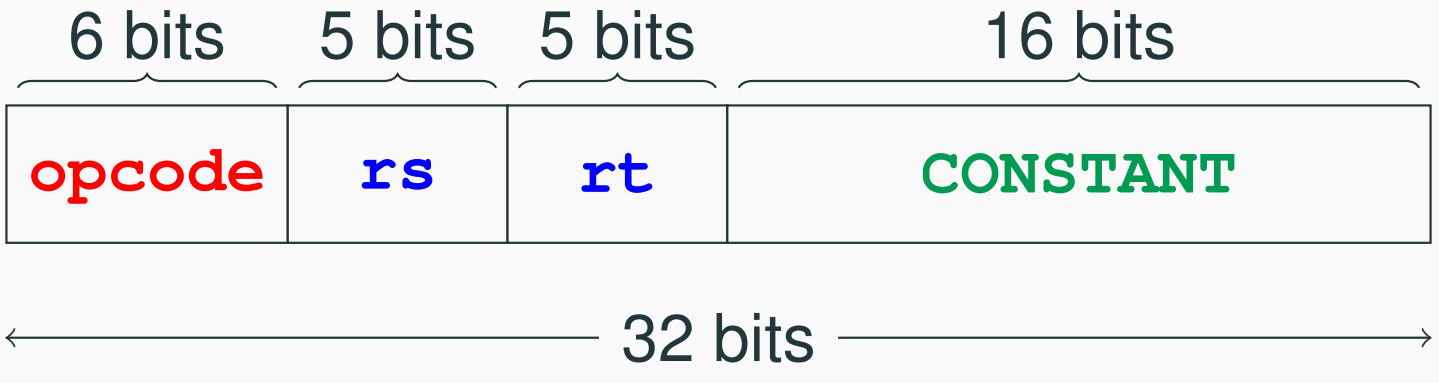
With only a jump instruction, you can at most have an infinite loop! But what about controlling the flow of the program based on some condition, i.e., **branching**? (*Q: incidentally, why is it called branching?*)

- Branch-on-equal (branch-if-equal):

  `beq rs, rt, label`

  - compares the content of registers **`rs`** and **`rt`**, and if they are equal, executes the sequence of instructions starting from the line designated by **`label`**.
- There is also **`bne`**: branch-on-not-equal.

MIPS Sequencing Instructions: Branching

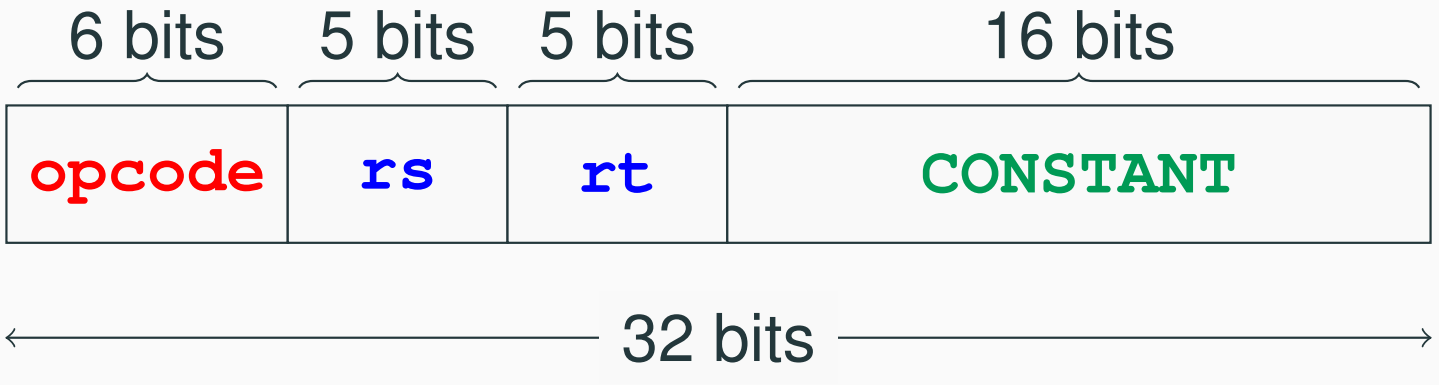


I-format

The (16-bit) constant is treated as the *relative* address from our current instruction address. In particular, the effect is: if the content of register **rs** is equal to **rt**, then:

PC = **PC**+4*__CONSTANT__

MIPS Sequencing Instructions: Branching



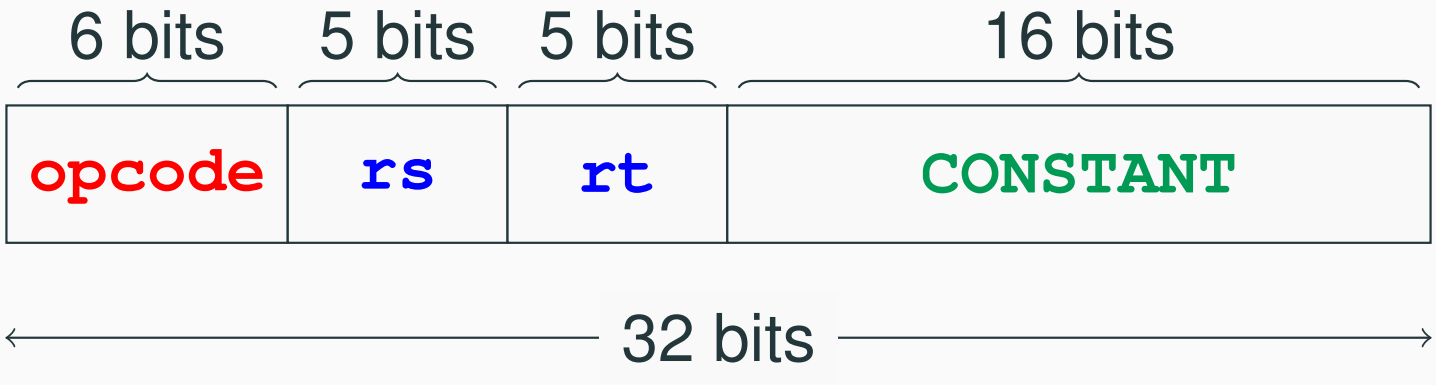
I-format

The (16-bit) constant is treated as the *relative* address from our current instruction address. In particular, the effect is: if the content of register **rs** is equal to **rt**, then:

PC = **PC**+4*__CONSTANT__

- *Q: why multiply by 4?*

MIPS Sequencing Instructions: Branching



I-format

The (16-bit) constant is treated as the *relative* address from our current instruction address. In particular, the effect is: if the content of register `rs` is equal to `rt`, then:

`PC` = `PC`+4*`CONSTANT`

- ▷ *Q: why multiply by 4?*
- ▷ *Q: how many instruction ahead/before can this go to?*

Memory Layout

Memory Layout

One of the features of the von-Neumann architecture (as opposed to *Harvard* architecture) is that both data and instructions reside in the main memory.

Memory Layout

One of the features of the von-Neumann architecture (as opposed to *Harvard* architecture) is that both data and instructions reside in the main memory.

- *Q: what is an advantage and a disadvantage of this feature?*

Memory Layout

One of the features of the von-Neumann architecture (as opposed to *Harvard* architecture) is that both data and instructions reside in the main memory.

- ▷ *Q: what is an advantage and a disadvantage of this feature?*

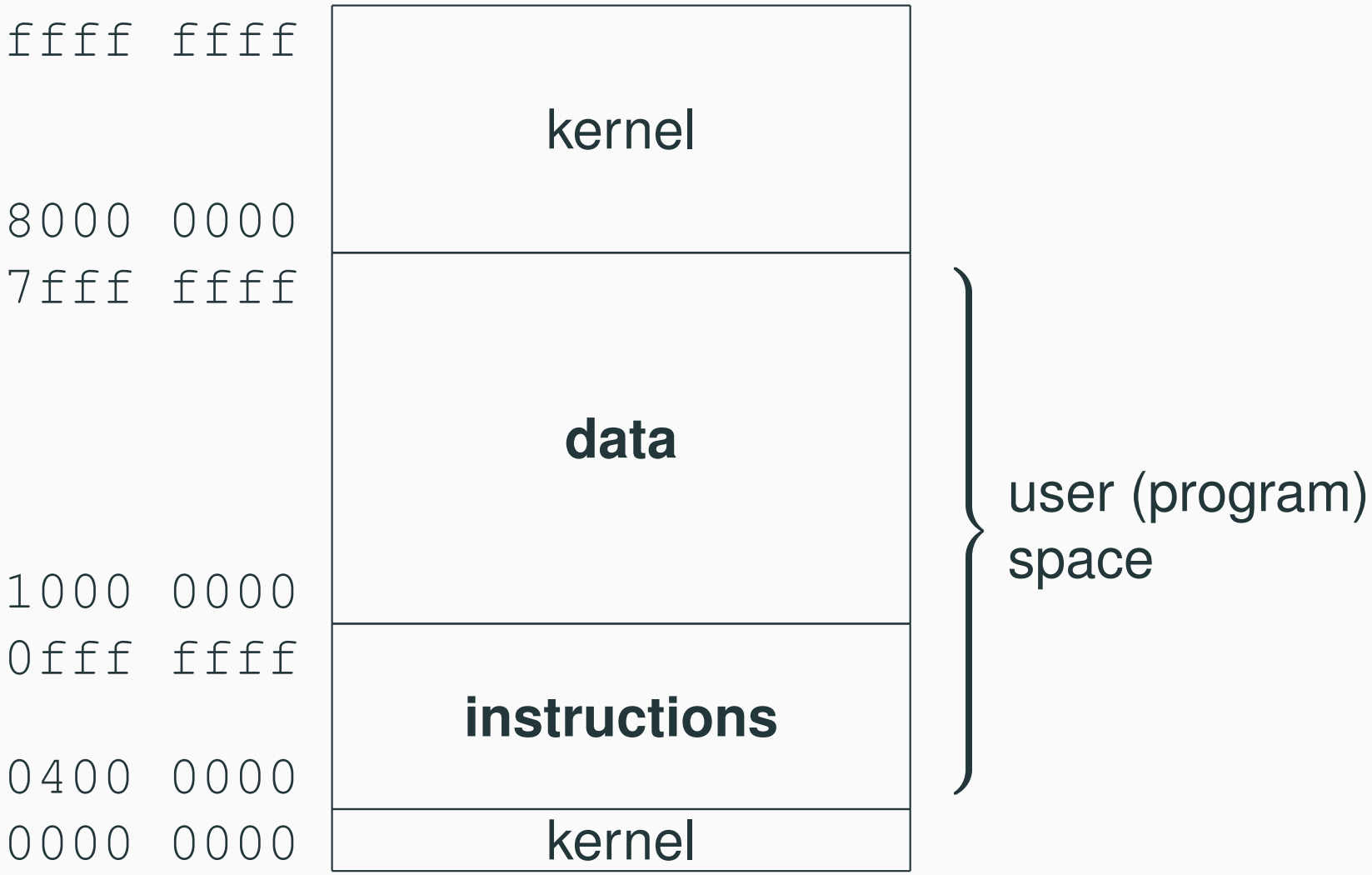
- ► So we need to have a **memory "layout"** (a.k.a. a **memory "map"** to keep the data and instructions in an orderly fashion, and separated as much as possible (why?!).

Declaring Data

```
1  .text
2      li $v0, 10 # syscall to exit
3      syscall # Exit
4  .data
5      MSG:    .asciiz "abacus"
6      MY_INT: .word 15
7      MY_INT_ARRAY:
8          .word   1
9          .word   2
10         .word   3
11         .word   4
12         .word   5
13         .word   6
14         .word   7
15     MY_INT_ARRAY_LEN:    .word   7
```

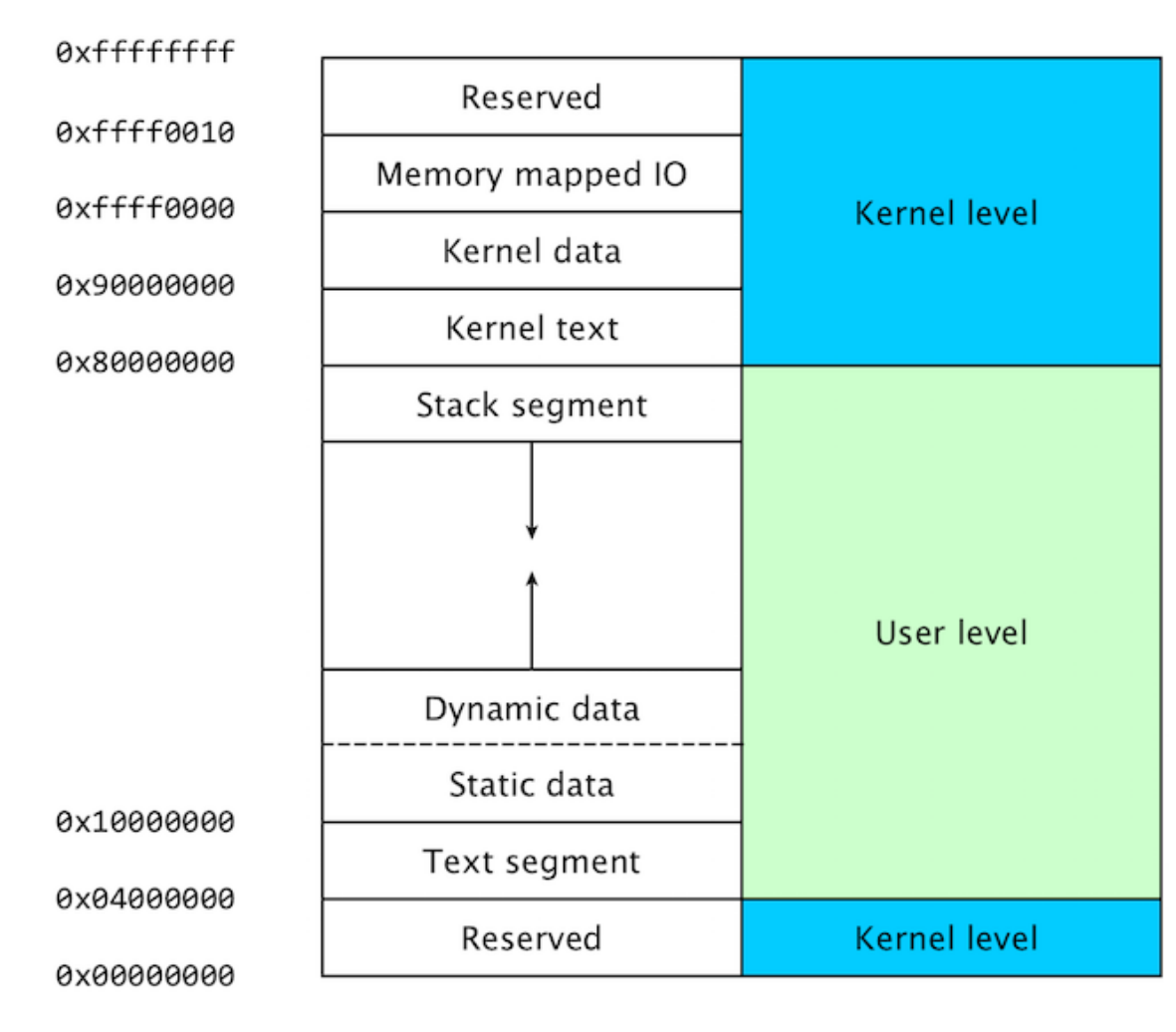
data_example.asm

Memory Layout



Memory layout in a (32-bit) MIPS.

Memory Layout



Detailed memory layout in a (32-bit) MIPS.

Recall that MIPS is an example of load/store architecture: any operation can only be done on the contents of registers. This means that in order to process data we have to first read them from memory into registers.

Memory Access

We can read the data we have stored in memory using the following commands:

- Load address:
  `la rt, <label of the data>`
  puts the address of the variable denoted by its label (which we have declared in the `.data` section of our code) in register `rt`.
- Load word:
  `lw rt, offset(rs)`
  puts 4 bytes (a word) from the memory into register `rt`, whose starting address (address of the its first Byte) is `$rs + offset`.[1]

[1] Note: by putting a dollar sign $ behind the name of a register, we mean the "content" of that register.

Memory Access

To store values back in memory, we can use the following instruction:

- Store word:
  `sw rt, offset(rs)`
  the content of register **`rt`** is written in memory whose address (of its first Byte is) **`$rs + offset`**. Note that as the only exception to the general rule, here, the destination is written second!

Memory Access

- Write a MIPS Assembly code that increment the value of **`MY_INT`** that we have declared in the memory.

Memory Access

- Write a MIPS Assembly code that increment the value of **`MY_INT`** that we have declared in the memory.

```
1    la   $t0, MY_INT      # t0=address(MY_INT)
2    lw   $t1, 0($t0)      # t1=value(MY_INT)
3    addi $t1, $t1, 1      # t1++
4    sw   $t1, 0($t0)      # value(MY_INT)++
```

I/O: System calls

I/O: System Calls

Service	Code in `$v0`	Arguments	Result
print integer	1	`$a0` = integer to print	-
print string	4	`$a0` = address of null-terminated string	-
read integer	5	-	$v0 holds integer read

Some of the Syscall services available in **`QtSpim`** , along with the detail of how to invoke them.

[Example demo.]

Practice Example for MIPS Assembly

Suppose we have four variables initially loaded into registers **t0**, **t1**, **t2**, **t3**. Let's implement a *while loop* in MIPS Assembly:

```
while (t0> t1) {
    t2 = t2 + t0;
    t0 = t0 - t3;
}
```

Let's do this through steps. First, the easy part: the inside part of the loop:

```
add $t2, $t2, $t0    # t2 <- t2+t0
sub $t0, $t0, $t3    # t0 <- t0-t3
```

Practice Example for MIPS Assembly

Let's turn it to a loop, and for now, without any conditions:

```
LOOP:
            # branching instruction to come here!
    add $t2, $t2, $t0    # t2 <- t2+t0
    sub $t0, $t0, $t3    # t0 <- t0-t3
    j LOOP
EXIT:   # exit, or the rest of the code
```

So what should we put as the branching condition to exit from the loop?

Practice Example for MIPS Assembly

The first (incorrect) attempt in the class was the following:

```
LOOP:
    beq $t0, $t1, EXIT   # goto exit if t0==t1
    add $t2, $t2, $t0    # t2 <- t2+t0
    sub $t0, $t0, $t3    # t0 <- t0-t3
    j LOOP
EXIT:    # exit, or the rest of the code
```

- although **`$t0`** keeps decrementing by **`$t3`**, it may never become equal to **`$t1`**: it may skip over it!
- even before entering the loop, we can have **`$t0<$t1`**, then the **`while`** loop should have not executed at all!

Note that though this is wrong, this will get $\sim 6/10$ mark!

Practice Example for MIPS Assembly

The second (still incorrect) attempt was the following:

```
LOOP:
    slt $t4, $t0, $t1          # t4 <- (t0<t1)
    bne $t4, $zero, EXIT       # if t4!=0 goto EXIT
    add $t2, $t2, $t0          # t2 <- t2+t0
    sub $t0, $t0, $t3          # t0 <- t0-t3
    j LOOP
EXIT:   # exit, or the rest of the code
```

- This implements **`while (t0>= t1)`** instead!
  (because even if **`$t0==$t1`**, the loop is executed)

Note that **`slt`** itself is not a branching instruction.

Again even though this is wrong, it'd get $\sim 8/10$ mark.

Practice Example for MIPS Assembly

And finally, a correct answer:

```
LOOP:
     slt $t4, $t1, $t0          # t4 <- (t1<t0)
     beq $t4, $zero, EXIT       # if t4==0 goto EXIT
     add $t2, $t2, $t0          # t2 <- t2+t0
     sub $t0, $t0, $t3          # t0 <- t0-t3
     j LOOP
EXIT:    # exit, or the rest of the code
```

Note that there may be more than one correct implementation, any correct answer will get 10/10 mark!

- More practice examples will be in your lab sheets 8, 9, 10, coursework, textbook, and relevant questions from previous year exams.