THE UNIVERSITY OF SYDNEY

# COMP2017 COMP9017 Assignment 2

Due: 11:59PM Tuesday 28 March 2023 local Sydney time

*This assignment is worth 5% + 30% of your final assessment*

## Task Description

Ah, yes, because what the world really needs right now is yet another virtual machine. But not just any virtual machine, no! We need one with the highly coveted and incredibly useful feature of heap banks. Because who needs to worry about memory allocation when you can just throw it in a heap, am I right? So gather 'round, folks, and get ready to develop the vm_RISKXVII, because nothing says "cutting edge" like a virtual machine named after a board game that this assignment has absolutely nothing to do with. Now, let's dive into the specs and get this party started!

In this assignment you will be implementing a simple virtual machine. Your program will take a single command line argument being the path to the file containing your RISK-XVII assembly code.

**Before** attempting this assignment it would be a good idea to familiarise yourself with registers, memory space, program counters, assembly and machine code. A strong understanding of these concepts is essential to completing this assignment. Section 3.6 and 3.7 of the course textbook provide specific detail to x86_64 architecture, however you can review these as a reference.

In order to complete this assignment at a technical level you should revise your understanding of bitwise operations, file IO, pointers and arrays.

Some implementation details are purposefully left ambiguous; you have the freedom to decide on the specifics yourself. Additionally this description does not define all possible behaviour that can be exhibited by the system; some error cases are not documented. You are expected to gracefully report and handle these errors yourself.

You are encouraged to ask questions on Ed[1]. Make sure your question post is of "**Question**" post type and is under "**Assignment**" category → "**A2**" subcategory. As with any assignment, make sure that your work is your own[2], and that you do not share your code or solutions with other students.

## The Architecture

In this assignment you will implement a virtual machine for an 32-bit instruction-set. The memory mapped virtual components of your machine are outlined below:

[1] https://edstem.org/au/courses/10466/discussion/
[2] Not GPT-3/4's, ChatGPT's or copilot's, etc.

- **0x0000 - 0x3ff**: Instruction Memory - Contains $2^{10}$ of bytes for text segment.
- **0x0400 - 0x7ff**: Data Memory - Contains $2^{10}$ of bytes for global variables, and function stack.
- **0x0800 - 0x8ff**: Virtual Routines - Accesses to these address will cause special operations to be called.
- **0xb700 +**: Heap Banks - Hardware managed 128 x 64 bytes banks of dynamically allocate-able memory.

Your machine also has a total of 32 registers, as well as a `PC` (program counter) that points to the address of the current instruction in memory. Each of the general-purpose registers can store 4 bytes (32 bits) of data that can be used directly as operands for instructions. **All registers are general-purpose except for the first one**, which has an address of 0. This register is called the **zero** register, as any read from it will return a value of 0. Writes to the **zero** register are ignored.

**During execution you should not store any information about the state of the machine outside of the virtual memory devices and the register bank.**

Note: A register stores a single value using a fixed bit width. The size of a register corresponding to the processor's *word size*, in this case 32 bits. Think of them as a primitive variable. Physical processor hardware is constrained, and the number of registers is always fixed. There are registers which serve specific purposes, and those which are general. Please identify these in the description and consider them for your solution. You need not consider special purpose registers, such as floating point, in this assignment.

# RISK-XVII Instruction-Set Architecture

An Instructions-Set Architecture(ISA) specifies a set of instructions that can be accepted and executed by the target machine. A program for the target machine is an ordered sequence of instructions.

Our virtual machine will operate on a home-brewed 'RISK-XVII' instruction set architecture. During marking, you will be provided with binaries in this ISA to run on your virtual machine. RISK-XVII is a reduced version of the well-known RV32I instruction set architecture, and your virtual machine should be able to execute binary programs compiled for RV32I, as long as they do not include instructions that were not specified by 'RISK-XVII'.

There are in total 33 instructions defined in RISK-XVII, they can be classified into three groups by their functionality:

1. Arithmetic and Logic Operations - e.g. `add, sub, and, or, slt, sll`
2. Memory Access Operations - e.g. `sw, lw, lui`
3. Program Flow Operations - e.g. `jal, beq, blt`

These instructions provide access to memory and perform operations on data stored in registers, as well as branching to different locations within the program to support conditional execution. Some instructions also contain data, i.e., an immediate number, within their encoding. This type of instruction is typically used to introduce hard-coded values such as constants or memory address offsets.

The RISK-XVII instruction set is Turing complete and, therefore, can run any arbitrary program, just like your PC!

Instructions in the RISK-XVII instruction set architecture are encoded into 4 bytes of data. Since each instruction can access different parts of the system, six types of encoding formats were designed to best utilize the 32 bits of data to represent the operations specified by each instruction: R, I, S, SB, U, UJ. The exact binary format of each encoding type can be found in the table below.:

| Type | Format | | | | | |
|---|---|---|---|---|---|---|
| | 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
| R | func7 | rs2 | rs1 | func3 | rd | opcode |
| I | imm[11:0] | | rs1 | func3 | rd | opcode |
| S | imm[11:5] | rs2 | rs1 | func3 | imm[4:0] | opcode |
| SB | imm[12 \| 10:5] | rs2 | rs1 | func3 | imm[4:1 \| 11] | opcode |
| U | imm[31:12] | | | | rd | opcode |
| UJ | imm[20 \| 10:1 \| 11 \| 19:12] | | | | rd | opcode |

Let's take a look at some common fields in all types of encoding:

- `opcode` - used in all encoding to differentiate the operation, and even the encoding type itself.
- `rd, rs1, rs2` - register specifiers. `rs1` and `rs2` specify registers to be used as the source operand, while `rd` specifies the target register. Note that since there are 32 registers in total, all register specifiers are 5 bits in length.
- `func3, func7` - these are additional opcodes that specify the operation in more detail. For example, all arithmetic instructions may use the same `opcode`, but the actual operation, e.g. add, logic shift, are defined by the value of `func3`.
- `imm` - immediate numbers. These value can be scrambled within the instruction encoding. For example, in SB, the 11st bit of the actual value was encoded at the 7th bit of the instruction, while the 12rd bit was encoded at the 31th bit.

An RISK-XVII program can be illustrated as below:

```
[Instruction 1 (32 bits)]
[Instruction 2 (32 bits)]
[Instruction 3 (32 bits)]

[...]

[Instruction n (32 bits)]
```

# RISK-XVII Instructions

We will now cover in detail all instructions defined in RISK-XVII. Pay close attention as your virtual machine need to be able to decode and execute all of these to be eligible for a full mark! **You are encouraged to summarise them into a reference table before implementing your code.**

Let's use `M` to denote the memory space. `M[i]` denotes access to the memory space using address `i`. For example, to write an immediate value `2017` to the first word of data memory: `M[0x400] = 2017`. Similarly, we use `R` to denote the register bank, e.g. `R[rd] = R[rs1] + R[rs2]` denotes an operation that adds the value in `rs1` and `rs2`, then store the result into `rd`.

## Arithmetic and Logic Operations

### 1 `add`

- Add - This instruction simply adds two numbers together.
- Operation - `R[rd] = R[rs1] + R[rs2]`
- Encoding:
  - Type: R
  - `opcode = 0110011`
  - `func3 = 000`
  - `func7 = 0000000`

### 2 `addi`

- Add Immediate - Add a number from register with an immediate number.
- Operation - `R[rd] = R[rs1] + imm`
- Encoding:
  - Type: I
  - `opcode = 0010011`
  - `func3 = 000`

### 3 `sub`

- Subtract
- Operation - `R[rd] = R[rs1] - R[rs2]`
- Encoding:
  - Type: R
  - `opcode = 0110011`
  - `func3 = 000`
  - `func7 = 0100000`

### 4 `lui`

- Load Upper Immediate - Load the upper part of an immediate number into a register and set the lower part to zeros.
- Operation - `R[rd] = {31:12 = imm | 11:0 = 0}`
- Encoding:
  - Type: U
  - `opcode = 0110111`

### 5 `xor`

- XOR
- Operation - `R[rd] = R[rs1] ^ R[rs2]`
- Encoding:
  - Type: R
  - `opcode = 0110011`
  - `func3 = 100`
  - `func7 = 0000000`

### 6 `xori`

- XOR Immediate
- Operation - `R[rd] = R[rs1] ^ imm`
- Encoding:
  - Type: I
  - `opcode = 0010011`
  - `func3 = 100`

### 7 `or`

- OR
- Operation - `R[rd] = R[rs1] | R[rs2]`
- Encoding:
  - Type: R
  - `opcode = 0110011`
  - `func3 = 110`
  - `func7 = 0000000`

### 8 `ori`

- OR Immediate
- Operation - `R[rd] = R[rs1] | imm`
- Encoding:
  - Type: I
  - `opcode = 0010011`
  - `func3 = 110`

### 9 `and`

- AND
- Operation - `R[rd] = R[rs1] & R[rs2]`
- Encoding:
  - Type: R
  - `opcode = 0110011`
  - `func3 = 111`
  - `func7 = 0000000`

### 10 `andi`

- AND Immediate
- Operation - `R[rd] = R[rs1] & imm`
- Encoding:
  - Type: I
  - `opcode = 0010011`
  - `func3 = 111`

### 11 `sll`

- Shift Left
- Operation - `R[rd] = R[rs1] « R[rs2]`
- Encoding:
  - Type: R
  - `opcode = 0110011`
  - `func3 = 001`
  - `func7 = 0000000`

### 12 `srl`

- Shift Right
- Operation - `R[rd] = R[rs1] » R[rs2]`
- Encoding:
  - Type: R
  - `opcode = 0110011`
  - `func3 = 101`
  - `func7 = 0000000`

### 13 `sra`

- Rotate Right - the right most bit is moved to the left most after shifting.
- Operation - `R[rd] = R[rs1] » R[rs2]`
- Encoding:
  - Type: R
  - `opcode = 0110011`
  - `func3 = 101`
  - `func7 = 0100000`

## Memory Access Operations

### 14 `lb`

- Load Byte - Load a 8-bit value from memory into a register, and sign extend the value.
- Operation - `R[rd] = sext(M[R[rs1] + imm])`
- Encoding:
  - Type: I
  - `opcode = 0000011`
  - `func3 = 000`

### 15 `lh`

- Load Half Word - Load a 16-bit value from memory into a register, and sign extend the value.
- Operation - `R[rd] = sext(M[R[rs1] + imm])`
- Encoding:
  - Type: I
  - `opcode = 0000011`
  - `func3 = 001`

### 16 `lw`

- Load Word - Load a 32-bit value from memory into a register
- Operation - `R[rd] = M[R[rs1] + imm]`
- Encoding:
  - Type: I
  - `opcode = 0000011`
  - `func3 = 010`

### 17 `lbu`

- Load Byte Unsigned - Load a 8-bit value from memory into a register
- Operation - `R[rd] = M[R[rs1] + imm]`
- Encoding:
  - Type: I
  - `opcode = 0000011`
  - `func3 = 100`

### 18 `lhu`

- Load Half Word Unsigned - Load a 16-bit value from memory into a register
- Operation - `R[rd] = M[R[rs1] + imm]`
- Encoding:
  - Type: I
  - `opcode = 0000011`
  - `func3 = 101`

### 19 `sb`

- Store Byte - Store a 8-bit value to memory from a register.
- Operation - `M[R[rs1] + imm] = R[rs2]`
- Encoding:
  - Type: S
  - `opcode = 0100011`
  - `func3 = 000`

### 20 `sh`

- Store Half Word - Store a 16-bit value to memory from a register.
- Operation - `M[R[rs1] + imm] = R[rs2]`
- Encoding:
  - Type: S
  - `opcode = 0100011`
  - `func3 = 001`

### 21 `sw`

- Store Word - Store a 32-bit value to memory from a register.
- Operation - `M[R[rs1] + imm] = R[rs2]`
- Encoding:
  - Type: S
  - `opcode = 0100011`
  - `func3 = 010`

## Program Flow Operations

### 22 `slt`

- Set if Smaller - Set `rd` to 1 if the value in `rs1` is smaller than `rs2`, 0 otherwise.
- Operation - `R[rd] = (R[rs1] < R[rs2]) ?  1 :  0`
- Encoding:
  - Type: R
  - `opcode = 0110011`
  - `func3 = 010`
  - `func7 = 0000000`

### 23 `slti`

- Set if Smaller Immediate - Set `rd` to 1 if the value in `rs1` is smaller than `imm`, 0 otherwise.
- Operation - `R[rd] = (R[rs1] < imm) ? 1 : 0`
- Encoding:
  - Type: I
  - `opcode = 0010011`
  - `func3 = 010`
  - ~~`func7 = 0000000`~~

### 24 `sltu`

- Set if Smaller - Set `rd` to 1 if the value in `rs1` is smaller than `rs2`, 0 otherwise. Treat numbers as unsigned.
- Operation - `R[rd] = (R[rs1] < R[rs2]) ? 1 : 0`
- Encoding:
  - Type: R
  - `opcode = 0110011`
  - `func3 = 011`
  - `func7 = 0000000`

### 25 `sltiu`

- Set if Smaller Immediate - Set `rd` to 1 if the value in `rs1` is smaller than `imm`, 0 otherwise. Treat numbers as unsigned.
- Operation - `R[rd] = (R[rs1] < imm) ? 1 : 0`
- Encoding:
  - Type: I
  - `opcode = 0010011`
  - `func3 = 011`
  - ~~`func7 = 0000000`~~

### 26 `beq`

- Branch if Equal.
- Operation - `if(R[rs1] == R[rs2]) then PC = PC + (imm « 1)`
- Encoding:
  - Type: SB
  - `opcode = 1100011`
  - `func3 = 000`

### 27 `bne`

- Branch if Not Equal.
- Operation - `if(R[rs1] != R[rs2]) then PC = PC + (imm « 1)`
- Encoding:
  - Type: SB
  - `opcode = 1100011`
  - `func3 = 001`

### 28 `blt`

- Branch if Less Than/
- Operation - `if(R[rs1] < R[rs2]) then PC = PC + (imm « 1)`
- Encoding:
  - Type: SB
  - `opcode = 1100011`
  - `func3 = 100`

### 29 `bltu`

- Branch if Less Than. Treat numbers as unsigned.
- Operation - `if(R[rs1] < R[rs2]) then PC = PC + (imm « 1)`
- Encoding:
  - Type: SB
  - `opcode = 1100011`
  - `func3 = 110`

### 30 `bge`

- Branch if Greater Than or Equal.
- Operation - `if(R[rs1] >= R[rs2]) then PC = PC + (imm « 1)`
- Encoding:
  - Type: SB
  - `opcode = 1100011`
  - `func3 = 101`

### 31 `bgeu`

- Branch if Greater Than or Equal. Treat numbers as unsigned.
- Operation - `if(R[rs1] >= R[rs2]) then PC = PC + (imm « 1)`
- Encoding:
  - Type: SB
  - `opcode = 1100011`
  - `func3 = 111`

### 32 `jal`

- Jump and Link - Jump to target code address, and save the `PC` value of next instruction into a register.
- Operation - `R[rd] = PC + 4; PC = PC + (imm « 1)`
- Encoding:
  - Type: UJ
  - `opcode = 1101111`

### 33 `jalr`

- Jump and Link Register - Jump to target code address from a register, and save the `PC` value of next instruction into a register.
- Operation - `R[rd] = PC + 4; PC = R[rs1] + imm`
- Encoding:
  - Type: I
  - `opcode = 1100111`
  - `func3 = 000`

# Virtual Routines

Virtual routines are operations mapped to specific memory addresses such that a memory access operation at that address will have different effects. This can be used to allow programs running in the virtual machine to communicate with the outside world through input/output (I/O) operations.

As part of your task to implement necessary I/O functions for your virtual machine, you are required to develop the following routines:

### 1 0x0800 - Console Write Character

A memory store command to this address will cause the virtual machine to print the value being stored as a single ASCII encoded character to **stdout**.

### 2 0x0804 - Console Write Signed Integer

A memory store command to this address will cause the virtual machine to print the value being stored as a single 32-bit signed integer in decimal format to **stdout**.

### 3 0x0808 - Console Write Unsigned Integer

A memory store command to this address will cause the virtual machine to print the value being stored as a single 32-bit unsigned integer in lower case hexadecimal format to **stdout**.

### 4 0x080C - Halt

A memory store command to this address will cause the virtual machine to halt the current running program, then output `CPU Halt Requested` to `stdout`, and exit, regardless the value to be stored.

### 5 0x0812 - Console Read Character

A memory load command to this address will cause the virtual machine to scan input from `stdin` and treat the input as an ASCII-encoded character for the memory load result.

### 6 0x0816 - Console Read Signed Integer

A memory load command to this address will cause the virtual machine to scan input from `stdin` and parse the input as a signed integer for the memory load result.

### 7 0x0820 - Dump PC

A memory store command to this address will cause the virtual machine to print the value of `PC` in lower case hexadecimal format to **stdout**.

#### 8 0x0824 - Dump Register Banks

A memory store command to this address will force the virtual machine to perform an Register Dump. See Error Handling.

#### 9 0x0828 - Dump Memory Word

A memory store command to this address will cause the virtual machine to print the value of `M[v]` in lower case hexadecimal format to **stdout**. `v` is the value being stored interpreted as an 32-bit unsigned integer.

#### 10 0x0830, 0x0834 - Heap Banks

These addresses are used by a hardware-enabled memory allocation system.

#### 11 0x0850 and above - Reserved

Our program will not call these addresses. You are free to use them for your own test cases and debugging purposes.

## Heap Banks

One of the biggest selling points of our RISK-XVII-based virtual machine is the hardware (virtual) enabled memory allocation support in addition to the two built-in static memory banks. Programs running inside the virtual machine can request more memory by interfacing with the allocation subsystem through virtual routines.

Your virtual machine program should manage memory allocation requests and ensure that the ownership of each block is always unique to `malloc` requests unless it is not used (free). You need to manage a total of 128 memory banks, each with 64 bytes. Each memory bank is a memory device that can be accessed as a linear array of bytes. To handle allocation requests larger than the size of a single bank, multiple consecutive banks need to be searched and allocated for the request. An error is returned if it is not possible to fulfill such a memory request. The mapped address of the initial byte of the first bank is 0xb700, and there are a total of 8192 bytes of memory that can be dynamically allocated.

**Specification** The below interfaces are defined for the virtual machine program:

#### 1 0x0830 - malloc

A memory store command to this address will request a chunk of memory with the size of the value being stored to be allocated. The pointer of the allocated memory (starting address) will be stored in `R[28]`. If the memory cannot be allocated, `R[28]` should be set to zero.

### 2 0x0834 - free

A memory store command to this address will free a chunk of memory starting at the value being stored. If the address provided was not allocated, an illegal operation error should be raised.

**Example** Let's consider a scenario where all 128 banks are not allocated yet (free), and we need to handle a `malloc` request with size 270. To fulfill the request, we need to find five free banks that are located consecutively, e.g., 64 + 64 + 64 + 64 + 14. We will set the first five blocks as used and return the address at the beginning of the first block: `R[28] = &Block[0]`.

Now suppose another request just came in with size 12. We need only a single block to fulfill the request, and after a search the first one-consecutive block is the sixth. We will mark the sixth block as used and return the address to complete the allocation: `R[28] = Block[5]`.

**Hint:** You are encouraged to use a linked list internally to store and maintain a record of the current allocation.

# Error Handling

## Register Dump

When an register dump was requested, the virtual machine should print the value of all registers, including `PC`, in lower case hexadecimal format to **stdout**. The output should be one register per line in the following format:

```
PC = 0x00000001;
R[0] = 0xffffffff;
R[1] = 0xffffffff;

R[...] = 0xffffffff;

R[31] = 0xffffffff
```

## Not Implemented

If an unknown instruction was detected, your virtual machine program should output `Instruction Not Implemented:` and the hexadecimal value of the encoded instruction to `stdout`, followed by a **Register Dump** and terminate. For example, when if `0xffffffff` were found in the instruction memory, your program should output:

```
Instruction Not Implemented: 0xffffffff
```

### Illegal Operation

When an illegal operation was raised, your virtual machine program should output `Illegal Operation:` and the hexadecimal value of the encoded instruction to `stdout`, followed by a **Register Dump** and then terminate.

Any memory accesses outside of defined boundaries will cause this error to be raised. Memory accesses to not yet allocated, or freed, heap banks will also cause this error to be raised.

## Starting the Virtual Machine

A binary file will be provided as an image of the instruction and data memory when running the program. This file can be found by opening the file path supplied as the first command line argument. The below C code outlines the format of the binary input file as a `struct`:

```c
#define INST_MEM_SIZE 1024
#define DATA_MEM_SIZE 1024
struct blob {
    char inst_mem[INST_MEM_SIZE];
    char data_mem[DATA_MEM_SIZE];
}
```

**All registers, including `PC`, will be initialised to zero at the initialisation of the virtual machine. During each cycle, the virtual machine should fetch and execute the instruction pointed by `PC`, and increase `PC` by 4.**

### Example 1 - Printing "H"

The following RISK-XVII assembly program will print the first letter of "Hello, World" to `stdout`:

```
00000000 <_start>:
   0:    7ff00113        addi   sp,x0,2047
   4:    00c000ef        jal    ra,10 <main>
   8:    000017b7        lui    a5,0x1
   c:    80078623        sb     zero,-2036(a5)

00000010 <main>:
  10:    000017b7        lui    a5,0x1
  14:    04800713        addi   a4,x0,72
  18:    80e78023        sb     a4,-2048(a5)
  1c:    00000513        addi   a0,x0,0
  20:    00008067        ret
```

In the program above, `<_start>` subroutine will initialize the virtual machine to prepare for C runtime. ret is a mnemonic for the instruction `jalr` where `rs1 = 1`. The `start` function loads the

address of C stack to `R[2]`, aka `sp`, the stack pointer, using the `addi` instruction. This is necessary to setup a runtime for the machine to execute C program. Using `addi` with the zero register as the source operand is a common method to load immediate value into a register.

Below is the source code for `<_start>` used in the example:

```
_start:

    # init stack pointer
    lui  sp, %hi(__stack_end)
    add  sp, sp, %lo(__stack_end)

    # jump to main
    call main

    # call halt routine.
    lui  a5,0x1
    sb   zero,-2036(a5)
```

The stack address was initialised to the bottom of data memory which indicates that the stack is empty, as stack grows upward. `sb` instruction writes to 0x0800 which will call the Console Write Character virtual routine to print of the character.

The second column indicates encoded instruction in hexadecimal format. Below illustrates the memory image input file for this program:

```
[1301F07F]
[EF00C000]
[B7170000]
[23860780]
[B7170000]
[13078004]
[2380E780]
[13050000]
[67800000]
[988 bytes of padding for instruction memory]
[1024 bytes of padding for data memory]
```

**The input memory image binary file will always be 2048 bytes large.**

Below is the equivalent C code to the RISK-XVII assembly program above:

```c
char volatile *const ConsoleWriteChar = (char *)0x0800;

int main() {
    *ConsoleWriteChar = 'H';
    return 0;
}
```

## Example 2 - Adding two numbers

The following C program running in the virtual machine will scan for two signed integer in `stdin`, then print the sum of these numbers to `stdout`:

```c
int volatile *const ConsoleWriteSInt = (int *)0x0804;

inline int scan_char() {
    int result;
    int addr = 0x0816;

    asm volatile("lw %[res], 0(%[adr])"
                 : [res]"=r"(result)
                 : [adr]"r"(addr));

    return result;
}

int main() {
    int a = scan_char();
    int b = scan_char();

    *ConsoleWriteSInt = a + b;

    return 0;
}
```

And the equivalent RISK-XVII assembly program:

```
00000000 <_start>:
   0:    7ff00113        li    sp,2047
   4:    00c000ef        jal   ra,10 <main>
   8:    000017b7        lui   a5,0x1
   c:    80078623        sb    zero,-2036(a5)

00000010 <main>:
  10:    00001737        lui   a4,0x1
  14:    81670793        addi  a5,a4,-2026
  18:    0007a683        lw    a3,0(a5)
  1c:    0007a783        lw    a5,0(a5)
  20:    00d787b3        add   a5,a5,a3
  24:    80f72223        sw    a5,-2044(a4)
  28:    00000513        li    a0,0
  2c:    00008067        ret
```

## Example 3 - 5 Sum

The following C program designed for our virtual machine will scan for up to five non-zero integers from `stdin`, and print the sum of these numbers to `stdout`. If a zero-valued number is found, the loop will break immediately and print out the sum. This program utilizes the `scan_char` function from **Example 2**.

```c
int volatile *const ConsoleWriteSInt = (int *)0x0804;
int main() {
    int count = 0;
    int sum = 0;
    int num;

    while(count++ < 5){
        num = scan_char();
        if(num == 0) break;

        sum += num;
    }

    *ConsoleWriteSInt = sum;

    return 0;
}
```

Below is the equivalent RISK-XVII assembly program:

```
00000000 <_start>:
  0:    7ff00113        li    sp,2047
  4:    00c000ef        jal   ra,10 <main>
  8:    000017b7        lui   a5,0x1
  c:    80078623        sb    zero,-2036(a5)

00000010 <main>:
 10:    000017b7        lui   a5,0x1
 14:    81678793        addi  a5,a5,-2026
 18:    0007a783        lw    a5,0(a5)
 1c:    02078c63        beqz  a5,54 <main+0x44>
 20:    00500713        li    a4,5
 24:    00000693        li    a3,0
 28:    00001637        lui   a2,0x1
 2c:    81660613        addi  a2,a2,-2026
 30:    00f686b3        add   a3,a3,a5
 34:    fff70713        addi  a4,a4,-1
 38:    00070663        beqz  a4,44 <main+0x34>
 3c:    00062783        lw    a5,0(a2)
```

```
40:    fe0798e3        bnez  a5,30 <main+0x20>
44:    000017b7        lui   a5,0x1
48:    80d7a223        sw    a3,-2044(a5)
4c:    00000513        li    a0,0
50:    00008067        ret
54:    00078693        mv    a3,a5
58:    fedff06f        j    44 <main+0x34>
```

# Compilation and Execution

Your virtual machine program will be compiled by running the default rule of a make file. Upon compiling your program should produce a single `vm_riskxvii` binary. Your binary should accept a single argument in the form of the path to a 'RISK-XVII' memory image binary file to execute.

```
make
./vm_riskxvii <memory_image_binary>
```

Please make sure the above commands will compile and run your program. An example Makefile has been provided in the Scaffold, but you're encouraged to customize it to your needs. Additionally, consider implementing the project using multiple C source files and utilizing header files.

Tests will be compiled and run using two make rules; `make tests` and `make run_tests`.

```
make tests
make run_tests
```

These rules should build any tests you need, then execute each test and report back on your correctness.

Failing to adhere to these conventions will prevent your markers from running your code and tests. In this circumstance you will be awarded a mark of 0 for this assignment.

# Marking Criteria

The following is the marking breakdown, each point contributes a portion to the total 5% + 30% of the assignment. You will receive a result of zero if your program fails to compile.

For full marks the total size on disk of your program's binary should not exceed 20kB.

Marks are allocated on the basis of:

- Automated Test Cases - 5 - Passing automatic test cases, a number of tests will *not* be released or run until after your final submission.
- Viva - 30 - You will need to answer questions from a COMP2017 teaching staff member regarding your implementation. You will be required to attend a zoom session with COMP2017

teaching staff member after the code submission deadline. A reasonable attempt will need to be made, otherwise you will receive zero for the assessment.

In this session, you will be asked to explain:

- How your program counter is affected by the opcodes executed.
- How your code organises and manages the stack memory for function calls.
- What are the edge cases you considered for when your program returns 1.
- Answer further questions.
- Your code will also be assessed on C coding style conventions (see Ed resources). Clean code will attract the best grade.

Correctness is based on:

- `vm_riskxvii` - return value of the program.
- `vm_riskxvii` - printed standard output of the binary program execution (all virtual routines).

Additionally marks will be deducted on the basis of:

- Compilation - If your submission does not compile you will receive an automatic mark of zero for this assessment.
- Style - Poor code readability will result in the deduction of marks. Your code and test cases should be neatly divided between header and source files in appropriate directories, should be commented, contain meaningful variable names, useful indentation, white space and functions should be used appropriately. Please refer to this course's style guide for more details.
- Tests - A lack of tests, or a lack of thorough testing will result in the deduction of marks. Please provide your test cases along with appropriate scripts to build, run and report the results of your tests. As a number of tests will not be released until after your final submission you are strongly encouraged to test all aspects of your program.
- Graceful Error Handling - The description above contains a number of undefined behaviours. Your program should gracefully catch each of these error, report, and exit. Should your program crash marks will be deducted. Your error messages should be meaningful in addition to the specified error handling format.
- Memory Leaks - Code that leaks memory will receive a mark of 0.

**Warning:** Any attempts to deceive or disrupt the marking system will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not follow the assignment description or if your code is unnecessarily or deliberately obfuscated.

# Academic declaration

By submitting this assignment you declare the following:

*I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.*

*I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgment from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.*

*I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.*

*I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.*