Jetic Gū

1. Handwritten submissions and proprietary formats (e.g. Pages or MS Word) **will not be graded**.
2. Mathematical expressions must be written <u>entirely</u> using LaTeX, otherwise **50%-100%** of marks will be deducted.
3. Circuits must be **tested**. Untested circuits will receive 0.

Submission File structure:

```
submission.zip
   - asm/
     - asm_arm16.py   // if you are using python
     - asm_arm16.cpp  // if you are using C++
     - makefile       // if you are using C++
   - main.cct         // this will be your main computer
   - csci250.clf      // your library
   - cct/
     - any other related circuit files*
   - README.md        // A list of all files in the sub-
mission, and what they are for
```

Todo.

Lab 5 (under construction)

ARM16 Specification

1. Introduction

This document details the specification of a ARM16 CPU, simplified from the 32bit ARMv7 Thumb specification.

2. Instruction list

ARM doesn't have fixed lengths for OPCODE. In the context of our ARM16, all instructions are 16bit. Briefly, the different kinds of operations could be determined from the first few digits of the instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OPCODE						Other operands									

Opcode	Instruction Encoding	
00xxxx	Register/Immediate: Arithmetic Operations	AU operations and CMP
010000	Register: Logical Operations	LU operations
010001	Special Register data instructions*	Don't care
01001x	Memory Load: from Literal Pool (PC with Offset)	MEM

Opcode	**Instruction Encoding**	
0101xx 011xxx 100xxx	Memory Load/Store (Single address)	MEM
1010XX	Relative Address calculation*	Don't care
1011xx	Misc*	Don't care
1100xx	Memory Load/Store (Blocks)*	Don't care
1101xx	Conditional branch: if-triggered subroutine/goto	B
11100x	Unconditional branch: jump	B
11111x	Do nothing	Idle

1. Register/Immediate: Arithmetic Operations:

Shifting (not required):

OP	**Instruction**	**Assembly**	**See**
00000	Logical Shift Left	–	Don't care
00001	Logical Shift Right	–	Don't care
00010	Arithmetic Shift Right	–	Don't care

Special Shifting (MOV implemented using Logic Shift Left):

OP	**Md**	**Instruction**	**Assembly**	**See**
00000	**00**	MOV	`MOV <Rd>, <Rm>`	`Rd_data <= Rm_data`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	Rm			Rd		

Addition/Subtraction format:

OP	**Md**	**Instruction**	**Assembly**	**See**
00011	**00**	Addition	`ADDS <Rd>, <Rn>, <Rm>`	`Rd_data <= Rn_data + Rm_data`
00011	**01**	Subtraction	`SUBS <Rd>, <Rn>, <Rm>`	`Rd_data <= Rn_data - Rm_data`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	Mode (Md)		Rm			Rn			Rd		

Addition/Subtraction with Imm3 format:

OP	Md	Instruction	Assembly	See
00011	**10**	Addition (Immediate)	`ADDS <Rd>, <Rn>, #<imm3>`	`Rd_data <= Rn_data + Imm(2 downto 0)`
00011	**11**	Subtraction (Immediate)	`SUBS <Rd>, <Rn>, #<imm3>`	`Rd_data <= Rn_data - Imm(2 downto 0)`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	Mode (Md)		Imm3			Rn			Rd		

MOV:

OP	Instruction	Assembly	See
00100	Move	`MOVS <Rd>, #<imm8>`	`Rd_data <= Imm`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			Imm8							

CMP (Compare), Immediate:

OP	Instruction	Assembly	See
00101	Compare	`CMP <Rn>, #<imm8>`	`CMP Rn_data, Imm`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Rn			Imm8							

Addition/Subtraction, with Imm8 format:

OP	Instruction	Assembly	See
00110	Add 8bits of immediate	`ADDS <Rdn>, #<imm8>`	`Rdn_data <= Rdn_data + Imm`
00111	Unsigned Subtract 8bits of immediate	`SUBS <Rdn>, #<imm8>`	`Rdn_data <= Rdn_data - Imm`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	X	Rdn			Imm8							

2. Register: Logical Operations

Logical operations all starts with 010000.

OpA	OpB	Instruction	Assembly	See
010000	**0000**	Bitwise AND	`ANDS <Rdn>, <Rm>`	`Rdn_data <= Rdn_data and Rm_data`
010000	**0001**	Bitwise Exclusive OR	`EORS <Rdn>, <Rm>`	`Rdn_data <= Rdn_data xor Rm_data`
010000	**0010**	Logical Shift Left	–	Don't care
010000	**0011**	Logical Shift Right	–	Don't care
010000	**0100**	Arithmetic Shift Right	–	Don't care
010000	**0101**	Add with Carry	–	Don't care
010000	**0110**	Subtract with Carry	–	Don't care
010000	**0111**	Rotate Right	–	Don't care
010000	**1000**	Test	–	Don't care
010000	**1001**	Reverse Subtract from 0	–	Don't care
010000	**1010**	Compare Registers	`CMP <Rn>, <Rm>`	`CMP Rn_data, Rm_data`
010000	**1011**	Compare Negative	–	Don't care
010000	**1100**	Bitwise OR	`ORRS <Rdn>, <Rm>`	`Rdn_data <= Rdn_data or Rm_data`
010000	**1101**	Multiply Two Registers	–	Don't care
010000	**1110**	Bitwise Bit Clear	–	Don't care
010000	**1111**	Bitwise NOT	`MVNS <Rd>, <Rm>`	`Rd_data <= not Rm_data`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	Opcode B				Rm			Rdn (Rd, Rn)		

3. Memory Load: from Literal Pool (PC with Offset)

This category has 1 instruction.

Opcode	Instruction	Assembly	See
01001	Calculates an address from PC (R7), apply immediate offset and save it in target register	`LDR <Rt>, #<Imm8>`	`Rt_data <= MEM<PC_data + Imm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	Rt			Imm8							

4. Memory Load/Store (Single address)

This category has 4 instructions that needs to be included.

OpA	OpB	Instruction	Assembly	See
0101	**000**	Store Register	`STR <Rt>, [<Rn>, <Rm>]`	`MEM<Rn_data + Rm_data> <= Rt_data`
0101	**001**	Store Register Halfword		Don't care
0101	**010**	Store Register Byte	–	Don't care
0101	**011**	Load Register Signed Byte	–	Don't care
0101	**100**	Load Register	`LDR <Rt>, [<Rn>, <Rm>]`	`Rt_data <= MEM<Rn_data + Rm_data>`
0101	**101**	Load Register Halfword	–	Don't care
0101	**110**	Load Register Byte	–	Don't care
0101	**111**	Load Register Signed Halfword	–	Don't care

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	OpB			Rm			Rn			Rt		

OpA	OpB	Instruction	Assembly	See
0110	**0**	Store Register (Immediate)	`STR <Rt>, [<Rn>, #<imm5>]`	`MEM<Rn_data + Imm> <= Rt_data`
0110	**1**	Load Register (Immediate)	`LDR <Rt>, [<Rn>, #<imm5>]`	`Rt_data <= MEM<Rn_data + Imm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	OpB	Imm5					Rn			Rt		

OpA == 0111, 1000, 1001 are not required hence can be treated as don't care.

5. Conditional branch: if-triggered subroutine/goto

Op	Instruction	Assembly	See
1101	Conditional Branch	`B.cond <label> (<#imm8>)`	If condition is met, `PC <= Imm`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	cond				Imm8							

6. Unconditional branch: jump

Op	Instruction	Assembly	See
11100	Unconditional Branch	`B <label>` `(<#imm11>)`	`PC <= Imm`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	Imm11										

7. Idle (Do nothing)

The CPU does nothing when the instruction is idle.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	don’t care										

3. Separation of Main Memory and Instruction Cache

The ARM16 CPU mentioned here will have its main memory separated from the instruction cache. I've decided that it's more efficient for us to implement the CPU like this, without having to deal with the nuances of implementing a complex hardware cache.

The Main Memory should be connected to the datapath through a single D16 bus `db`, where the direction of information and whether information from `db` should be stored in the main memory is also controlled using `db_dir`. The memory module should be 16bit addressable (every word is 16bit, every address retrieves 16bit of data), with 16bits for address.

The instruction cache is a read-only memory unit. You can use the same specification for your main memory module, or an ROM unit. The choice is yours. The instruction cache should receive address from PC (R7 register) directly.

4. Register Array

Your CPU needs to contain 8 general-purpose registers. Among the 8 GPRs, R0-R6 do not have special roles, and R7 is your Programme Counter. Your register array should be implemented such that at every `CLK` when the whole array has `Rw==1`, if R7 is not receiving a new value from the `ALU` or `db`, R7's value should increase by 1 on its own.

5. Pipeline Requirements

Your ARM16 CPU can be designed with pipeline. The standard pipeline that can be followed here is the classic 3 stage model:

1. Fetch

   A new instruction is fetched into the **Instruction Register**
2. Decode

   Your instruction decoder takes the value from the **Instruction Register** as input, then generates the appropriate **Control Word** (of your design) to your ALU, your mem_ctrl, your db_ctrl, etc., and store the control word in a special **Control Word** register. Retrieved register data may also be stored in the **Control Word** register.
3. Execute

   Your ALU, mem_ctrl, and db_ctrl executes the instruction according to your control word, then updates the register array accordingly.

Important: if the instruction is Branch (`B`), or any instructions that might require **stalling**, to successfully implement the pipeline, you must implement a mechanism to stall individual stages as they become necessary.

Notably:

1. stalling for branching: all instructions after a `B` instruction must be stalled before continuing, allowing the potentially new PC to be correctly calculated and loaded.
2. stalling for register values: if the instruction currently being executed needs to change the content in the register array (excluding PC which requires a `B` instruction to change it's value from the compiler/assembler level), the Decoding of the next instruction should be stalled, to prevent incorrect values from the register array from being retrieved.
3. alternatively to point 2, if you can modify the pipeline to get rid of the necessity of doing so, you may also do so.

6. Assembler Requirement

6.1 Development Target

Our development target is your ARM16 CPU computer, so your assembler should output in a way such that it can be loaded into the instruction cache of your ARM16 CPU.

Your assembler may be programmed in Python or C++. If Python, name it as **`asm_arm16.py`**. If C++, name it as **`asm_arm16.cpp`**, and include a **`makefile`**.

The assembler will need to be a programme that can take 1 mandatory argument and 1 optional. e.g. in Python:

```
python3 asm_arm16.py SRC_CODE [-o OUT_FILE]
```

`SRC_CODE` here will be a source code file written in assembly. `OUT_FILE` is the output filename, and should be defaulted to `./a.out` unless the value is provided.

6.2 Directives and Symbols

You need only to support the `.global` directive. Your assembler should also check if `_main` is declared using `.global` and defined.

Users of your assembler should be able to write programmes with functions. Functions are declared the same way as `_main`, but doesn't need to involve `.global`.

Symbols of Functions need to fulfil basic requirements for variable names for other programming languages such as python. Exception being the register names. In ARM16, we use `x0` - `x6` as register names. Optionally, users may decide to use uppercase for register names or Linux style register names (`r0` - `r6`), whether you should support it is up to you.

Users may also decide to use literal numbers, you will need to support decimal, hexadecimal, and optionally binary literals. The detailed formats can be found in LS17.

6.3 Function Calls

Your users may decide to declare functions and use `B/BL` to call these functions. You will need to support this. Your assembler should make sure `B/BL` is the only way for the programmer to change `PC`'s value, so you should prevent `PC` from being changed by other instructions, including by the programmer using `MOV`.

Your assembler should also implement `BL`. For our ARM16, this is done by:

1. Saving current `PC` to `R6` (`LR`: Link Register) using MOV
2. Use `B` to branch to target function, after its finished execution,
3. Use `MOV` to go back to original `PC` stored in `R6` (e.g. main function)

You are not required to support recursive function calls, you can assume only in `_main` will other functions be called. Notice also that users may decide to use Symbols/Labels for their functions, so your assembler will need to figure out the appropriate memory addresses for these functions' instructions in the main memory.

7. Grading

Students who have successfully implemented a pipeline, in addition to correct submission of an assembler will receive A+ in this course, without the need of attending the final exam.

Otherwise, the grading of this Lab will be divided into the following:

1. Successful implementation of an Instruction Decoder (2pt)
2. Successful implementation of the ALU, by assembling address calculation, the Arithmetic Unit, the Logical Unit, and CMP module, connecting it to the main memory module and show that it works (1pt)
3. Successful implementation of the Instruction Cache, and register array (1pt)
4. Successful integration of the above 3 (3pt), by way of testing the implementation with a programme assembled by the reference assembler.
5. Correctly implemented the assembler as required (3pt).
    1. 1.5pt given to an executable assembler that can perform basic arithmetics and register microoperations.
    2. 1.5pt given to correct implementation of `B/BL` instruction, function call, function declaration using labels, and necessary checks to prevent PC from being changed by the programmer using other instructions.