計算機結構 Final Project Report

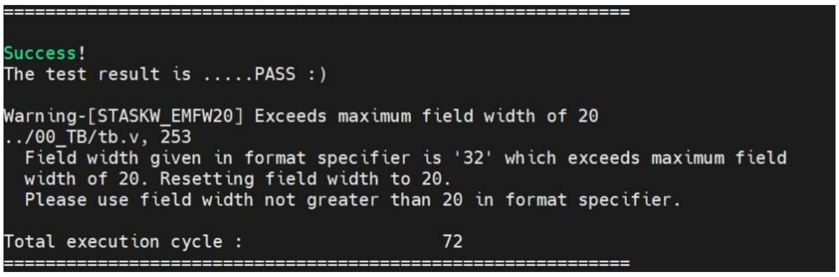
Group 1
B09502137 胡芝瑜 B10502076 金家逸

1. Outcome

a. Performance

Table 1. Single cycle CPU with cache performance

Instruction set	Execution cycle
I0	72
I1	385
I2	416
I3	650



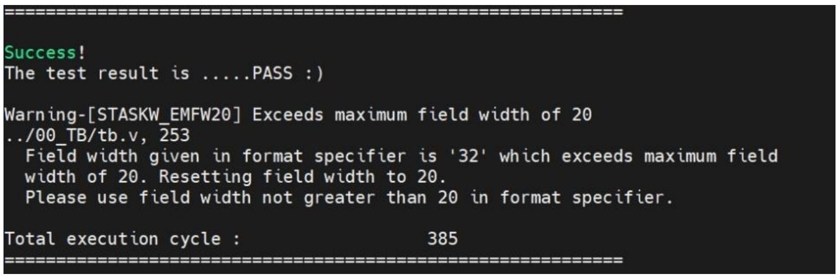
```
============================================================

Success!
The test result is .....PASS :)

Warning-[STASKW_EMFW20] Exceeds maximum field width of 20
../00_TB/tb.v, 253
  Field width given in format specifier is '32' which exceeds maximum field
  width of 20. Resetting field width to 20.
  Please use field width not greater than 20 in format specifier.

Total execution cycle :                   72
============================================================
```

Fig 1. I0 with cache



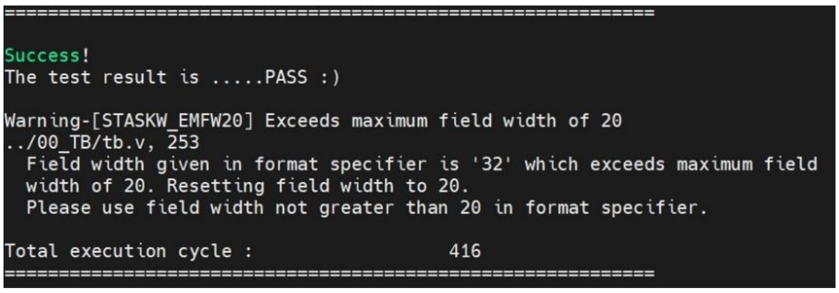
```
============================================================

Success!
The test result is .....PASS :)

Warning-[STASKW_EMFW20] Exceeds maximum field width of 20
../00_TB/tb.v, 253
  Field width given in format specifier is '32' which exceeds maximum field
  width of 20. Resetting field width to 20.
  Please use field width not greater than 20 in format specifier.

Total execution cycle :                  385
============================================================
```

Fig 2. I1 with cache



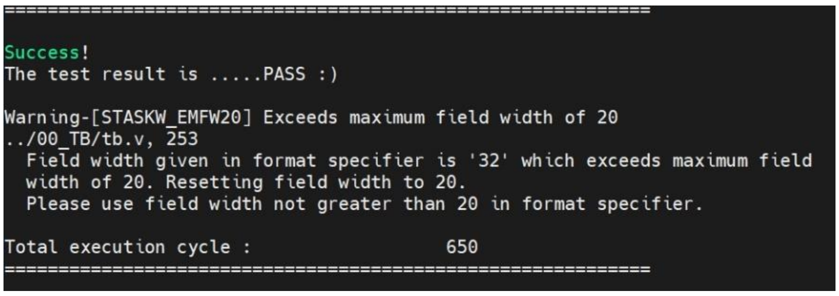
```
============================================================

Success!
The test result is .....PASS :)

Warning-[STASKW_EMFW20] Exceeds maximum field width of 20
../00_TB/tb.v, 253
  Field width given in format specifier is '32' which exceeds maximum field
  width of 20. Resetting field width to 20.
  Please use field width not greater than 20 in format specifier.

Total execution cycle :                  416
============================================================
```

Fig 3. I2 with cache



```
============================================================

Success!
The test result is .....PASS :)

Warning-[STASKW_EMFW20] Exceeds maximum field width of 20
../00_TB/tb.v, 253
  Field width given in format specifier is '32' which exceeds maximum field
  width of 20. Resetting field width to 20.
  Please use field width not greater than 20 in format specifier.

Total execution cycle :                 650
============================================================
```

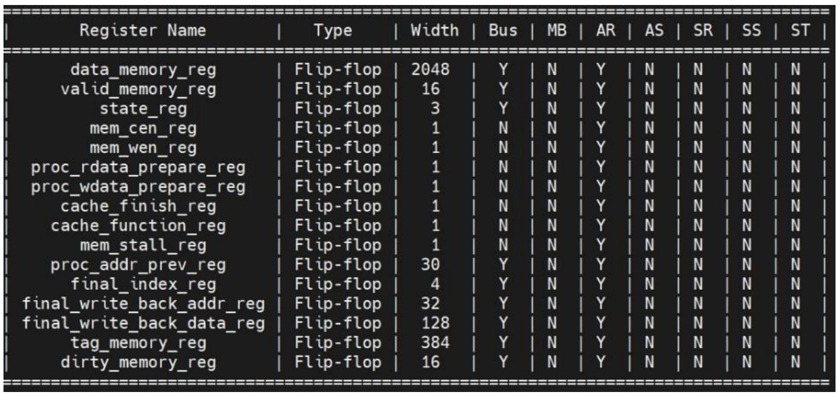
Fig 4. I3 with cache

b. Cache V.S. without cache

Table 2. Cache V.S. without cache performance

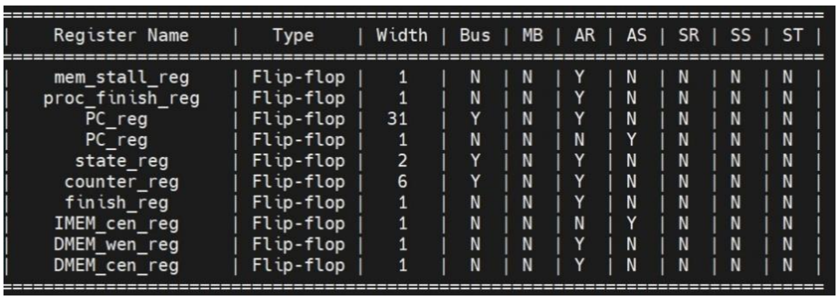
Instruction set	Without cache	With cache	Speedup
I0	84	72	1.16
I1	488	385	1.27
I2	434	416	1.04
I3	1465	650	2.25

c. Register table



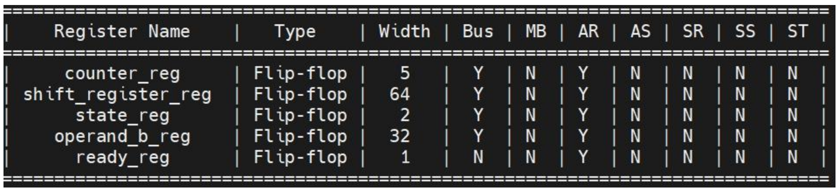
```
===============================================================================
|      Register Name        |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|     data_memory_reg       | Flip-flop | 2048  |  Y  | N  | Y  | N  | N  | N  | N  |
|     valid_memory_reg      | Flip-flop |  16   |  Y  | N  | Y  | N  | N  | N  | N  |
|        state_reg          | Flip-flop |   3   |  Y  | N  | Y  | N  | N  | N  | N  |
|       mem_cen_reg         | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
|       mem_wen_reg         | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
|  proc_rdata_prepare_reg   | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
|  proc_wdata_prepare_reg   | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
|     cache_finish_reg      | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
|    cache_function_reg     | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
|      mem_stall_reg        | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
|    proc_addr_prev_reg     | Flip-flop |  30   |  Y  | N  | Y  | N  | N  | N  | N  |
|     final_index_reg       | Flip-flop |   4   |  Y  | N  | Y  | N  | N  | N  | N  |
| final_write_back_addr_reg | Flip-flop |  32   |  Y  | N  | Y  | N  | N  | N  | N  |
| final_write_back_data_reg | Flip-flop |  128  |  Y  | N  | Y  | N  | N  | N  | N  |
|     tag_memory_reg        | Flip-flop |  384  |  Y  | N  | Y  | N  | N  | N  | N  |
|    dirty_memory_reg       | Flip-flop |  16   |  Y  | N  | Y  | N  | N  | N  | N  |
===============================================================================
```

Fig 5. Cache



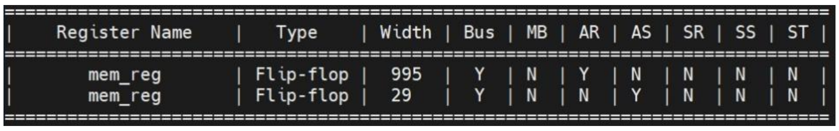
```
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|    mem_stall_reg    | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
|   proc_finish_reg   | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
|       PC_reg        | Flip-flop |  31   |  Y  | N  | Y  | N  | N  | N  | N  |
|       PC_reg        | Flip-flop |   1   |  N  | N  | N  | Y  | N  | N  | N  |
|      state_reg      | Flip-flop |   2   |  Y  | N  | Y  | N  | N  | N  | N  |
|     counter_reg     | Flip-flop |   6   |  Y  | N  | Y  | N  | N  | N  | N  |
|     finish_reg      | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
|    IMEM_cen_reg     | Flip-flop |   1   |  N  | N  | N  | Y  | N  | N  | N  |
|    DMEM_wen_reg     | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
|    DMEM_cen_reg     | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
===============================================================================
```

Fig 6. CHIP



```
==========================================================================================
|    Register Name    |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
==========================================================================================
|     counter_reg     | Flip-flop |   5   |  Y  | N  | Y  | N  | N  | N  | N  |
| shift_register_reg  | Flip-flop |  64   |  Y  | N  | Y  | N  | N  | N  | N  |
|      state_reg      | Flip-flop |   2   |  Y  | N  | Y  | N  | N  | N  | N  |
|    operand_b_reg    | Flip-flop |  32   |  Y  | N  | Y  | N  | N  | N  | N  |
|      ready_reg      | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
==========================================================================================
```

Fig 7. MULDIV

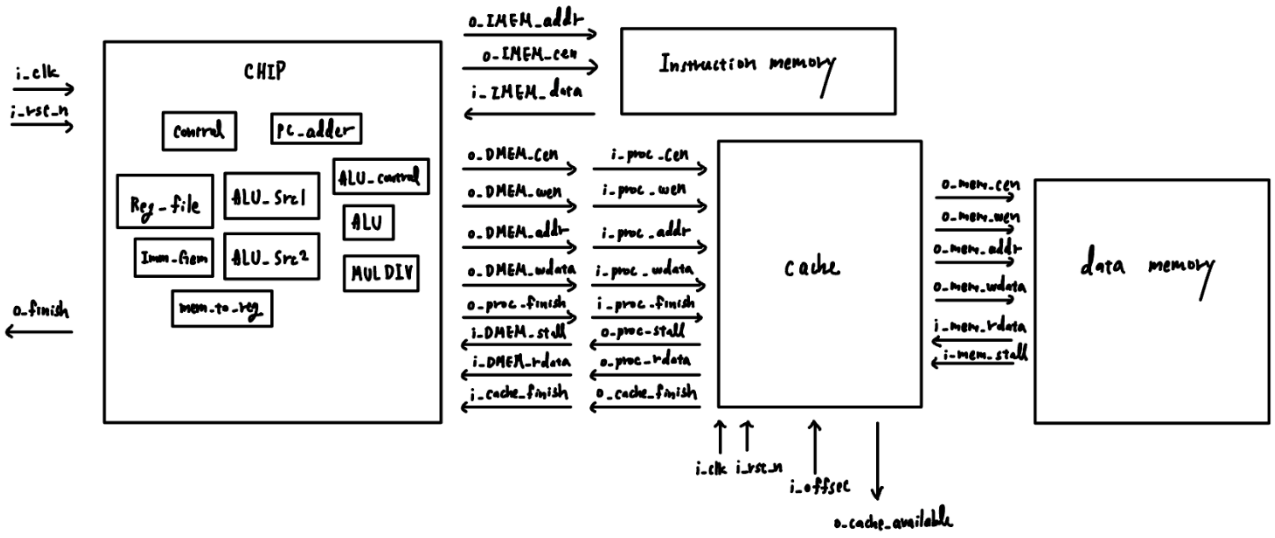


```
==========================================================================================
|    Register Name    |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
==========================================================================================
|       mem_reg       | Flip-flop |  995  |  Y  | N  | Y  | N  | N  | N  | N  |
|       mem_reg       | Flip-flop |  29   |  Y  | N  | N  | Y  | N  | N  | N  |
==========================================================================================
```

Fig 8. Reg_file

2. Single cycle CPU design

a. Block diagram



b. Instructions not in slide

For AUIPC, we add a MUX before ALU input port 1, to control the data input to ALU be PC or Read data 1, also add the instruction to immediate transform part and ALU control, so that when meet AUIPC, ALU will receive PC from input port 1 and immediate from input port 2 and add these two to complete AUIPC.

For the store into register part of JAL and JALR, we add an input in register to memory MUX, so when we detect JAL or JALR, we can input PC+4 to write port of register. Besides, we use respective detector at control, to show if this instruction is JAL or JALR.

For the branch (BEQ, BNE, BLT, and BGE), we will separate these four types and not branch instruction into different representation to store in branch detector at control part, also, we will subtract the two inputs in ALU and separate outcome into same, larger, smaller three different representations of output.

For the PC part with JAL, JALR, BEQ, BNE, BLT, and BGE, firstly, we combine all its control part detector value from control, and the branch comparison outcome from ALU into one input of PC adder, also, PC offset get from immediate, Read data 1 from register, and PC itself input to PC adder, too. Then we can identify the instruction type from the input and calculate the correct output of PC.

For ECALL, when we detect the instruction, we will make sure all instruction done and data from cache transmit into memory, and then also set o_finish into 1.

c. Multicycle instruction

When encounter multicycle instruction, which is multiplication, we will change to multicycle operation state, which will stall the CPU, and count 32 cycle to wait for the calculation, then we will store the calculation outcome in one cycle, and change back to idle state to continue operating.

3. Cache design

We implemented a direct-mapped write back cache with a configuration of 2 bits for the Byte offset, 2 bits for Block offset, 4 bits for the index, and the remaining 24 bits for the tag. The cache consists of a total of 16 blocks, with each block containing 4 words.

Since the offset of each instruction set is different, when data is loaded, firstly we will subtract the rightmost 4 bit of instruction set's offset from the input data memory address, then the new address's right most 4 bits will become its Block offset and Byte offset. When we need to write data back to memory, the rightmost 4 bit of instruction set's offset will be added back to the block's address, which will successfully rebuild the needed address for memory to write back one block with 4 words.

For Read Hit, the value stored in the cache will be directly returned to the chip. For a Read Miss, if the dirty bit of the data in the cache is 1, the data in the cache will be written back to data memory before loading the data from data memory, and then the data will be returned to the chip. For a Write Hit, the value in the cache will be directly modified, and the dirty bit will be set to 1. In the case of a Write Miss, if the dirty bit of the data to be replaced is 1, it will be written back to data memory first, otherwise, the data will be loaded from data memory, overwriting the value in the cache, and the dirty bit will be set to 1. After the chip executes all instructions, the cache will write back all data with dirty bits set to 1 to data memory, completing the entire operation.

If run CPU without cache is needed, set cache available to 0 can change the CPU operation into without Cache condition.

4. Work distribution table

	金家逸	胡芝瑜
CPU & Cache	Base structure design & debugging	Design checking & refining & debugging
Report	Cache	CPU

5. Observation

在這次的 final project 中，藉由 CPU 與 cache 的實作，我們可以更清楚的了解他們內部與彼此之間如何傳遞訊號與進行判讀，從而完成指令。在過程中，最重要也最具有挑戰性的就是確保每一個 instruction 都有正確的 path 與得到預想的 outcome ，這些常常在理解上容易，但是在實作上卻發現有很多小細節需要注意，同時有許多部分要進行 control 與 mux 等其他配合，才可以得到預期的結果。此外，像是在進行 branch 指令的設計時，剛開始只有針對其相減結果是否為零進行判讀，則遇到其他不是 beq 指令就會出現問題，這種時候從 nWave, assembly language 之間進行比對與判別錯誤的能力就很重要。還有，由於硬體語言與平時軟體語言並不相同，在寫時需要注意更多會不會造成 latch 的小細節，有些軟體能行得通的方式也要注意迴避，而且在硬體語言上會把所有東西都分成不同區塊進行不同方面的判別，再合成整個 instruction 的 implementation ，因此在寫的時候也需要有點整個架構的畫面，才不會在寫的時候有所缺漏或是出錯，後續在 debug 上也更容易由邏輯上找到當初的錯誤從而更正。