Computer Architecture

b11901152:林育正、b11901179:詹仲睿

December 25, 2023

1 Block diagram of the design and introduction

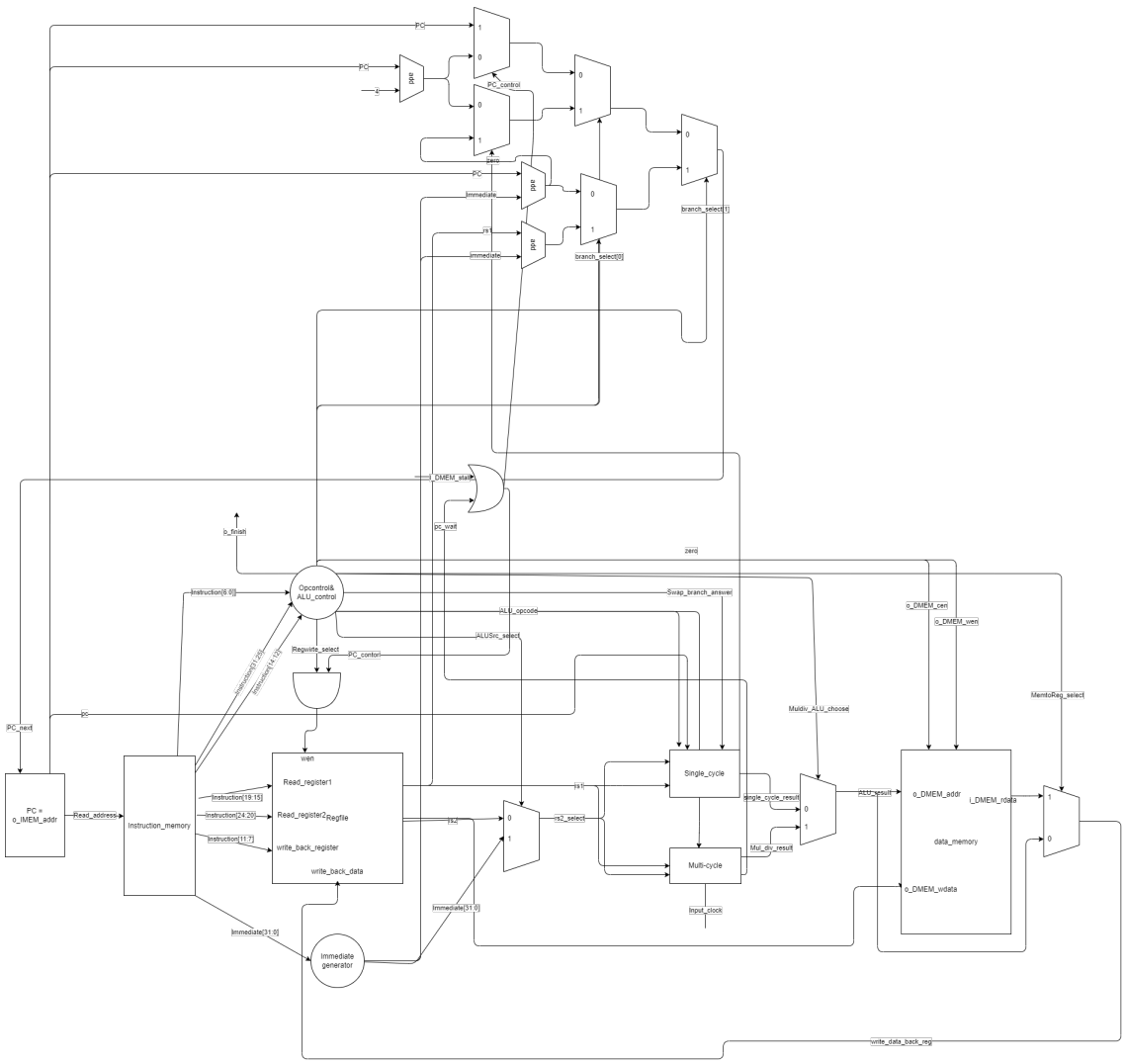

Figure 1: The diagram of the design

Above is the block diagram of our design and we will introduce it separately.

1.1 Instruction memory and Regfile

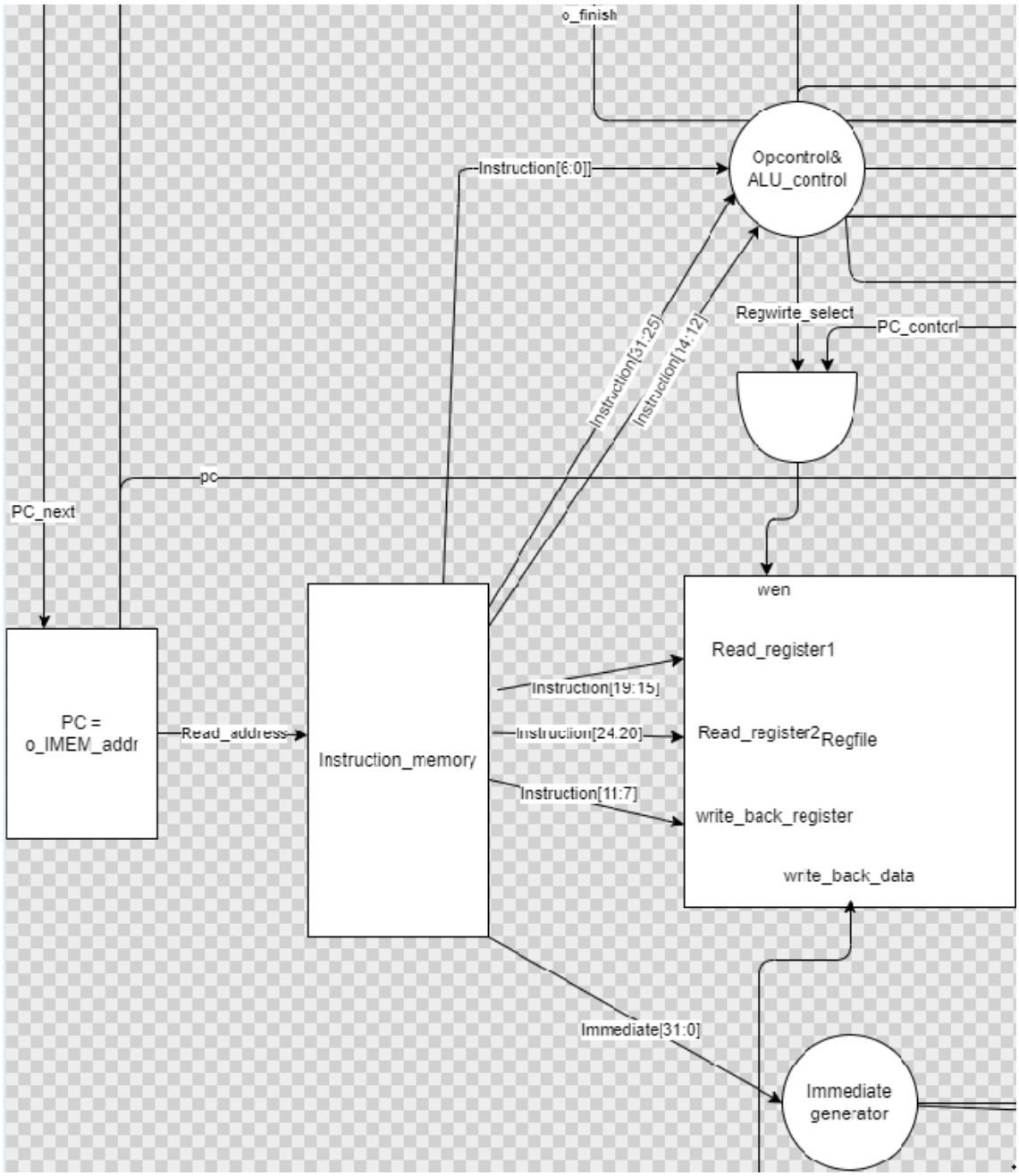

Figure 2: The diagram of the IM and Reg

As you can see, we actually do the same thing taught in class to design our instruction memory and Regfile as usual. However, we synchronize the Opcontorl and ALU control together. Moreover, the signal of Regwrite have to do AND operation with PC_control which is a control signal determine whether to stall our process and we will introduce it later.

1.2 ALU and Data memory

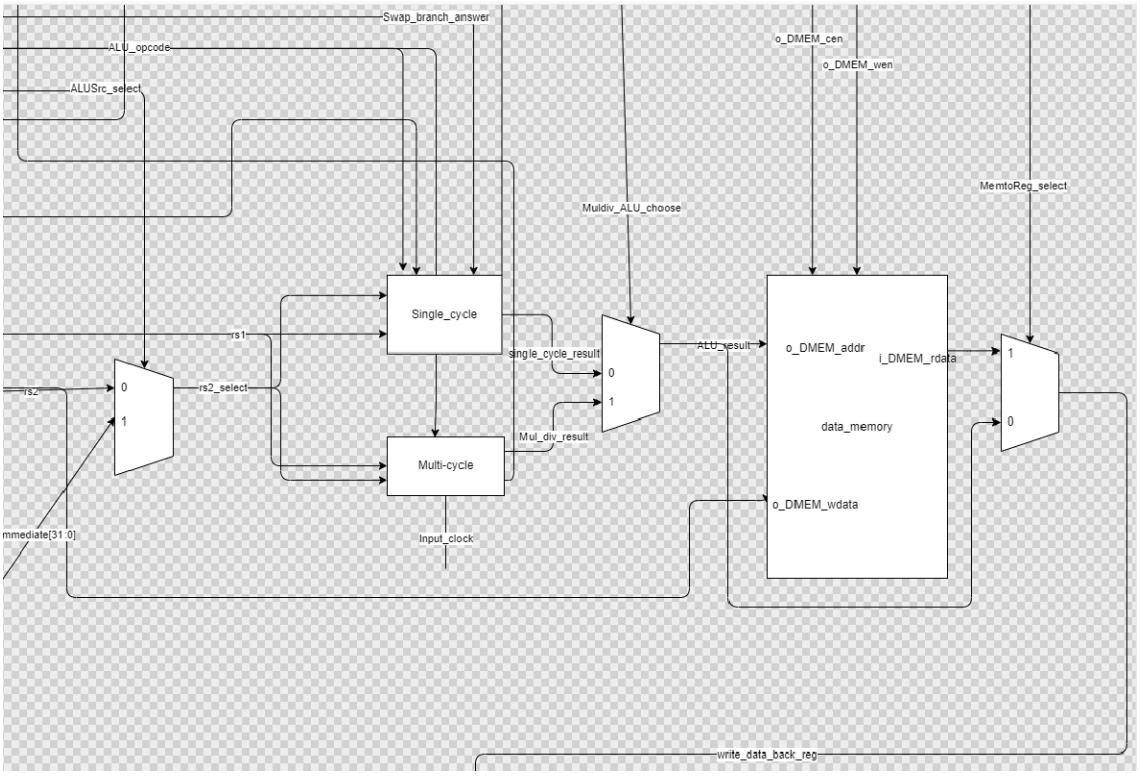

Figure 3: The diagram of the IM and DM

To realize the single cycle control for simple instruction such as add and subtract and the multiple cycle control for multiplication. We separate our ALU design into two part Single_ALU and MUlti_ALU and the ALU_opcode will help us to determine which ALU have to be run up. Finally, there is MULDIV_ALU_choose signal to help up determine which output result to pick.

When we have to do multiplication the pc_wait signal will rise to 1 to tell PC to wait and let the other write back register operation stop.

For the single cycle ALU, we add a swap_branch_answer to tell the ALU whether to swap the output zero signal. For instance, we use subtract operation both for beq and bne. When the rs1 and rs2 are equal we will check the swap_branch_answer. If it is 1 then it actually tell us the branch instruction is bne, hence zero will become 0, same logic beq, blt, bge.

1.3 PC_control

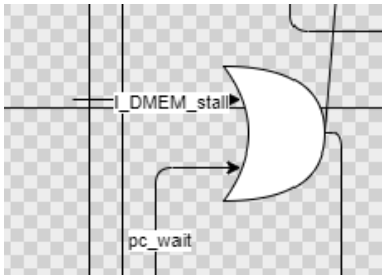

Figure 4: The diagram of the Control

Above is how we determine PC_control signal. If we are accessing memory (I_DEM_stall) or the current instruction is MUL (which send pc_wait) – both require multicycle to process – then we have to told PC to stall.

1.4 PC_next determination

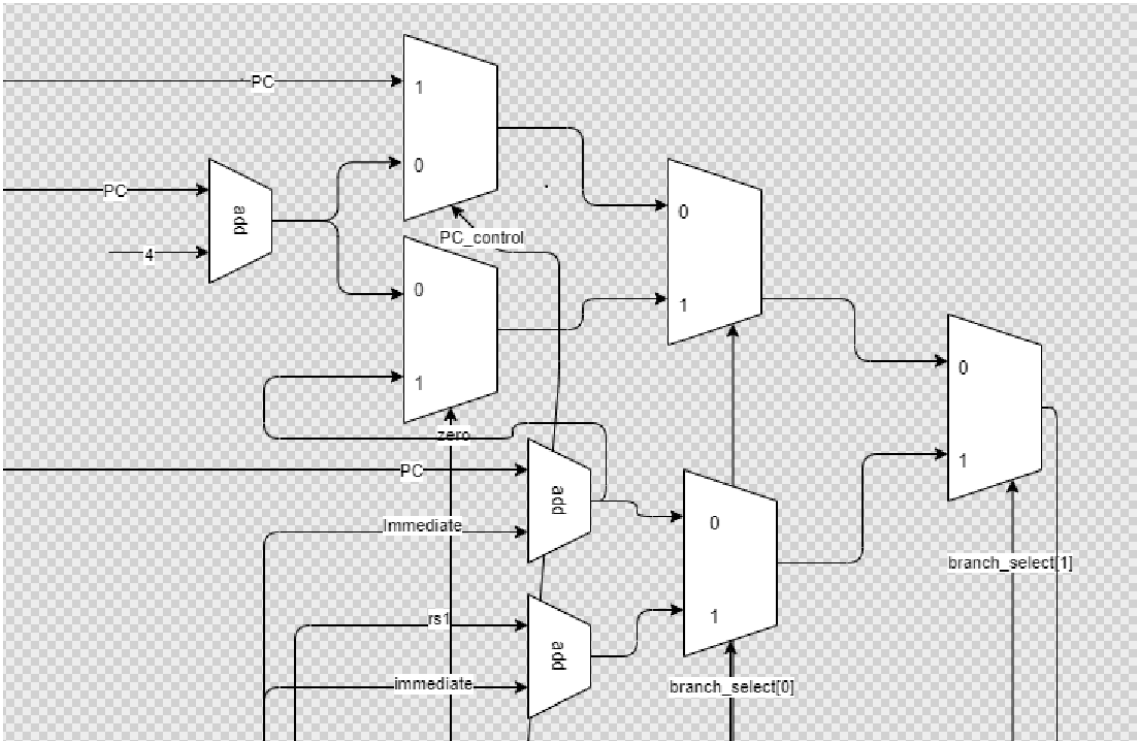

Figure 5: The diagram of the PC

To determine what the current operation is, we use a branch_select signal from op_control. If the signal is 11, the instruction is JALR, hence PC_next is rs1+immediate. If the signal is 10, it means the instruction is JAL, hence PC_next is PC+immediate. If the signal is 01, it means we are in B-type instruction; therefore, we have to check zero which is the output signal of ALU to determine whether branch successes. If zero is 1, then branch successes, then PC_next is PC+immediate, otherwise is PC+4. If the signal is 00, then we may be in store, load or multiplication. Therefore, we have to read PC_control signal to determine whether we have to wait until the process finish, otherwise we can just jump to PC+4.

2 Cache Implementation

2.1 Cache Architecture

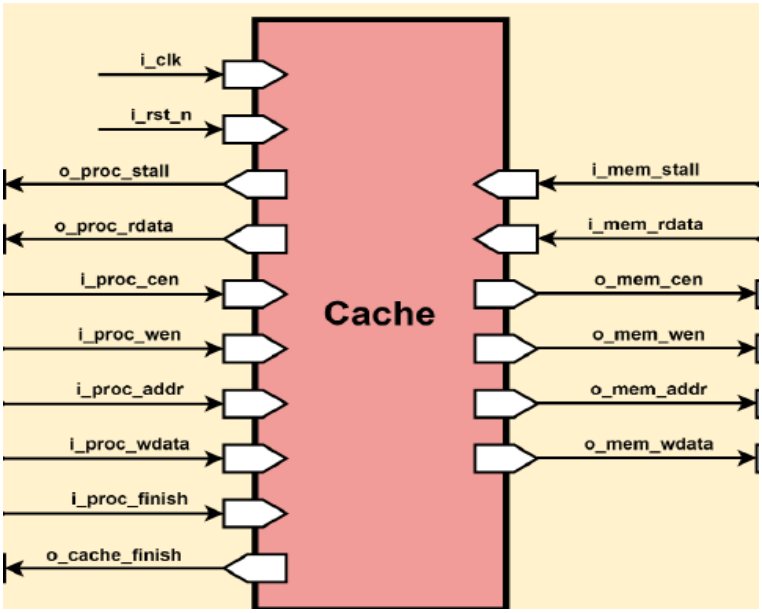

Figure 6: cache wires

We design our cache to be a write-back cache with 16 blocks(4 words/block), direct-mapped, so [31:8] is tag, [7:4] is index, [3:0] is block offset and byte offset. Our cache is not much different from the write-back cache taught in class, except for subtracting i_proc_addr with i_offset so that the address seen in cache starts from 0(to handle invalid data address). Then we add i_offset back whenever there's need to access memory in order to output right address to memory.

2.2 Stategraph

Our cache states is based on the one in textbook. We added one more state to handle the write back when the program finishes(enter this state whenever we receive i_proc_finish and leave the state when all datas are written back, then set o_cache_finish to 1). Also, we put the set valid, set tag part in Allocate state.

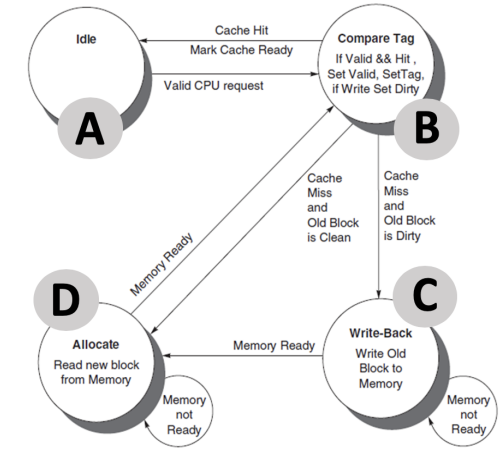

Figure 7: The state graph in textbook

3 Performance

3.1 Without Cache

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

Warning-[STASKW_EMFW20] Exceeds maximum field width of 20
../00_TB/tb.v, 244
  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 :                  77
```

Figure 8: The snapshot of the I0 without cache

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

Warning-[STASKW_EMFW20] Exceeds maximum field width of 20
../00_TB/tb.v, 244
  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 :                 440
-------------------------------------------------------------
```

Figure 9: The snapshot of the I1 without cache

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

Warning-[STASKW_EMFW20] Exceeds maximum field width of 20
../00_TB/tb.v, 244
  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 :                 397
=============================================================
```

Figure 10: The snapshot of the I2 without cache

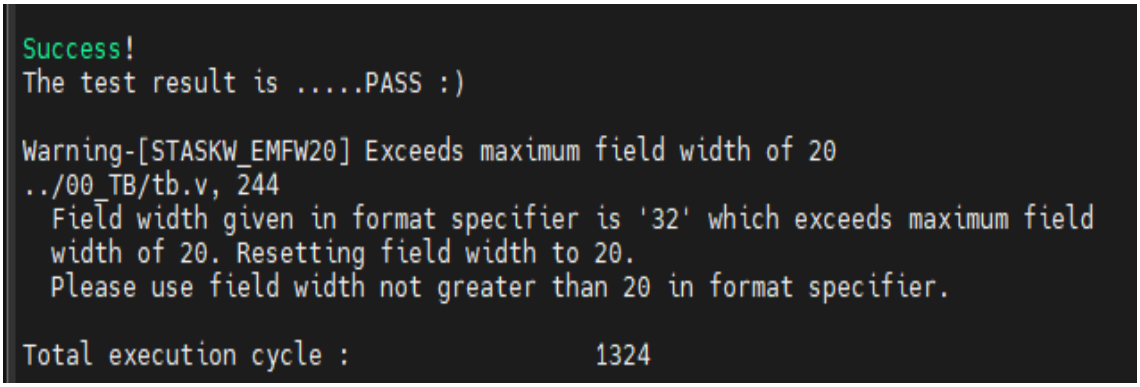

Figure 11: The snapshot of the I3 without cache

3.2 With Cache

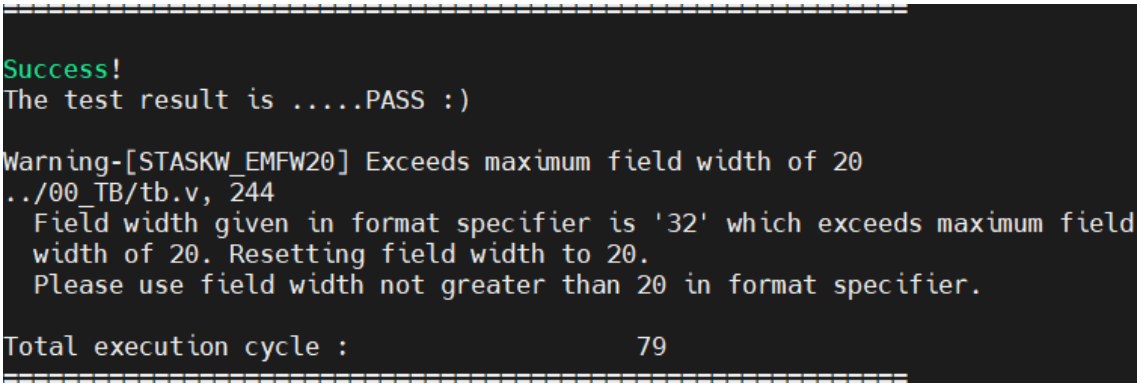

Figure 12: The snapshot of the I0 with cache

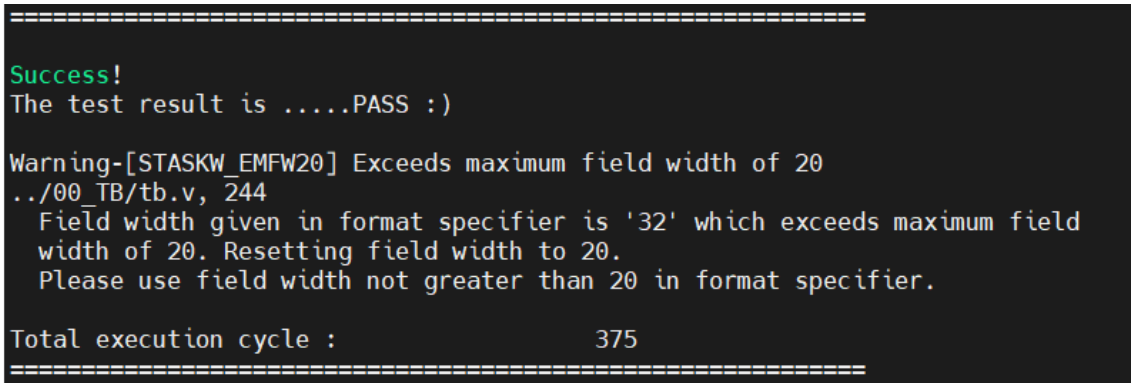

Figure 13: The snapshot of the I1 with cache

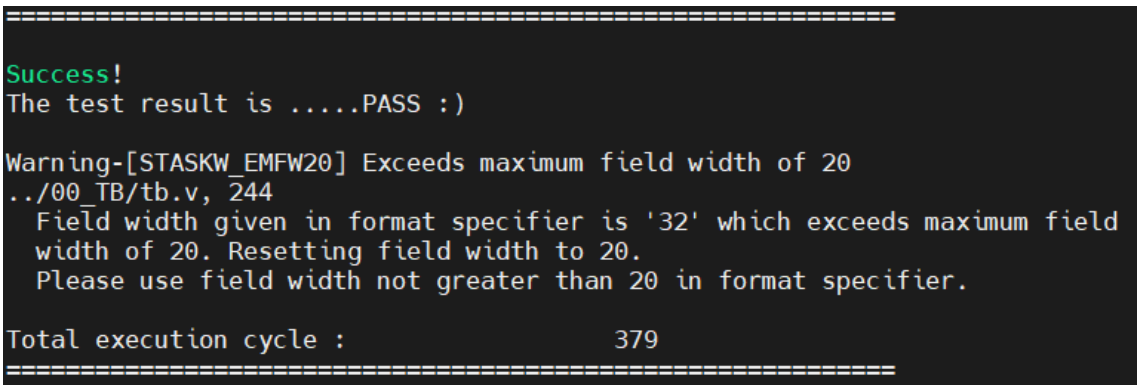

Figure 14: The snapshot of the I2 with cache

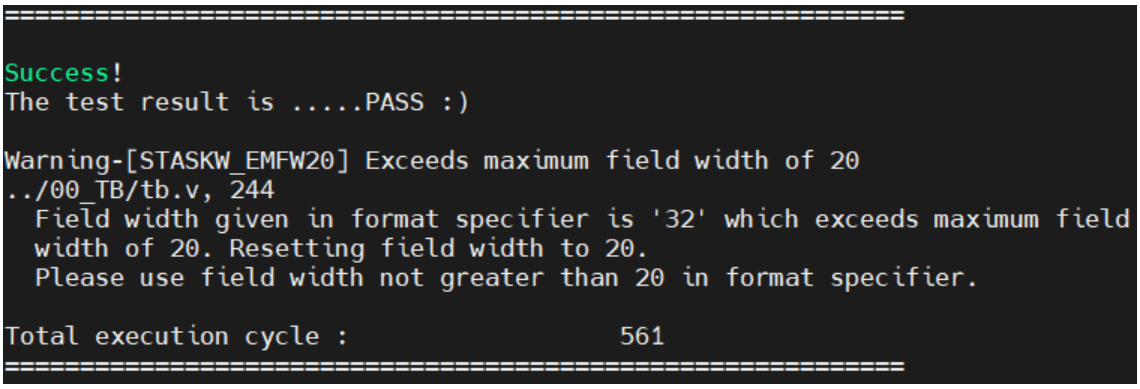

Figure 15: The snapshot of the I3 with cache

3.3 Cache Speedup

Instruction set	Without Cache	With Cache	Speedup
I0	77	79	0.97
I1	440	375	1.17
I2	397	379	1.05
I3	1324	561	2.36

Table 1: Performance table.

We can see the speedup in I3 is significant. The reason is that our cache implementation is a write-back cache, which can reduce the cycle sw instruction needs, and in this case I3 processes many sw. Another point worth to mention is that our I0 speedup is less than 1, which is the due to the fact that we need to write our data in each block(in our case 16 blocks) back into memory in our write-back cache, and each block requires one cycle to write into memory. Hence the real end of the program is the total execution cycle - 16, if we neglect the last write-back cycles, our performance is better than that with cache.(The solution to this is that we can write dirty blocks back instead of all blocks, but we didn't realize it.)

4 Register Table

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
PC_reg	Flip-flop	31	Y	N	Y	N	N	N	N
PC_reg	Flip-flop	1	N	N	N	Y	N	N	N

Figure 16: registers in chip

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
cnt_reg	Flip-flop	6	Y	N	N	N	N	N	N
result_reg	Flip-flop	64	Y	N	N	N	N	N	N
operand_b_reg	Flip-flop	32	Y	N	N	N	N	N	N

Figure 17: registers in MULDIV_unit

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

Figure 18: registers in Reg_file

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
counter_reg	Flip-flop	4	Y	N	N	N	N	N	N

Figure 19: registers in counter

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
state_reg	Flip-flop	3	Y	N	Y	N	N	N	N
cen_reg	Flip-flop	1	N	N	Y	N	N	N	N
wen_reg	Flip-flop	1	N	N	Y	N	N	N	N
dirty_reg	Flip-flop	16	Y	N	Y	N	N	N	N
data_reg	Flip-flop	2048	Y	N	Y	N	N	N	N
tag_reg	Flip-flop	384	Y	N	Y	N	N	N	N
valid_reg	Flip-flop	16	Y	N	Y	N	N	N	N

Figure 20: registers in cache

5 Observation

While the design of the memory in this project is quite different to the memory taught in the class. Hence there are several reasons that reduce the improvement of the memory with cache. First of all, the data memory is stored right after the instruction memory, and the requirement of the instruction memory will be denied. Therefore, we have to adjust the requiring address of the cache, which can be achieve through offset sent my memory.

In the Chip file, since we want to minimize the operation that ALU support, we have to add several signal to make this. Such as swap signal added to recognize beq and bne mentioned above.

6 Work distribution

組員	Chip	Cache	Report
林育正	v	v	v
詹仲睿	v	v	v

Table 2: An example table.