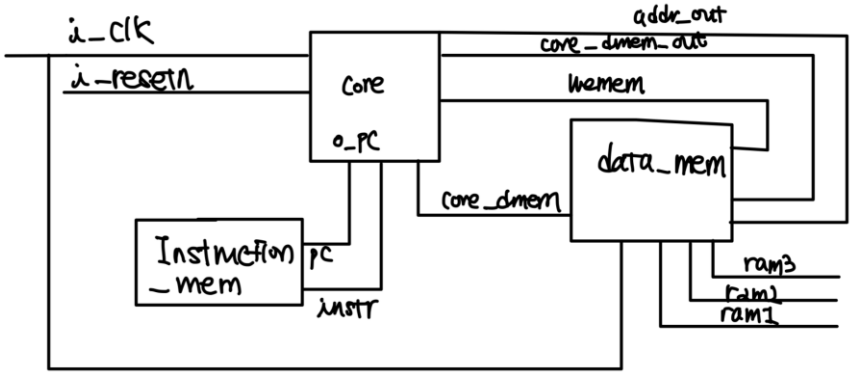
ASIC design report

2017160111 안진모

Chapter 1. Schematic & explain my design in detail

1) CPU design



Overall Schematic은 위와 같다. 여기서, Core part는 당연히 i/o signal을 Processing하는 part이고, 우리가 구현하고자 하는 Havard Architecture에 맞추어 D/I mem을 구분하였다. I_mem의 경우 주어진 구현에 맞추어, $readmemh의 구문을 아래처럼 넣어주었다.

```
$readmemh("test_program2.txt", rom);
```

이를 통해 Instruction을 Simultaneous하게 fetch하고, PC value를 31:2 로 Truncation하여 마치 4가 증가할 때마다, 1줄씩 update되도록 하였다. (이 때 Scenario별로 test_program 1or2가 정해진다.)

다음은 data_mem이다. 사실 우리가 RTL Verification 만으로는 실제 RAM에 접근하기 어려워, 이를 Flip-Flop으로 설계하였다. 또한 여기서 ram1 ,ram2 ,ram3는 [31:0] ram [0:31]중 ram[0], ram[1], ram[2]에 wire한 내용으로, 내부에 저장이 잘 되었는지 확인하기 위해 바깥으로 빼서 놓았다. 따라서 향후 저장 시에, 주소를 0,1,2 에만 저장하는 것을 볼 수 있을 것이다.

(ex. sw x18, 0(x0) , In risc V-> x0 is hardly wired to zero)

마지막으로 core unit이다. 과제의 요청대로, Forwarding unit으로 data-hazard를 방지하였고, Control hazard의 경우, Comparator라는 H/W를 따로 추가하였고, Static-taken mode로 구현하였다. 또한, ALU의 add unit에 CLA(carry look-ahead adder)를 apply하였다. Core unit의 자세한 schematic을 추가하고 싶지만, 이후 Verification이 잘되는 모습으로 이게 제대로 설명되었음을 확인하자.(cause of 5 page limitation)

Chapter 2. Validation Scenarios

1) Scenario_1 – Targeting a control Hazard

- 먼저, c언어로, 변수 초기화를 진행하고, ALU operation을 진행한 후, if문+함수를 통해 프로시저 콜을 하고, 받은 value를 store하는 시나리오를 생각하자. 간단하게 Assembly로 생각하면 register에 0를 apply하고 간단한 ALU 이후 beq/bne로 분기를 한다. 이후, 해당 branch 내부에서 ALU를 진행한 후, 다시 Unconditional branch로 돌아온 후 store를 하면 된다. (control Hazard 해결)
- 이에 해당하는 Scenario는 다음과 같다.

C language

```c
int main (void){
int a = 0;
int b = 0;
a = a +3;
b = b +2;
int c=0;
if (a != b){
  c = jinmo(a,b);
}
return c;
}


int jinmo (int a,int b){
    return a+b;
}
```

Risc-V assembly language (+Instruction address) , (a->x19, b->x20, c->x18)

0 : add x19, x0, x0 /// a=0

4 : add x20, x0, x0 /// b = 0

8 : addi x19, x19, 3 /// a = a+3

12 : addi x20, x20, 2 /// b = b+2

16 : add x18, x0, x0 ///c =0

20 : bne x19, x20, L1 /// if (a != b)

24 : sw x18, 0(x0) // 앞서 설명한 이유로, ram1 에다 store하자.

EXIT:

L1:

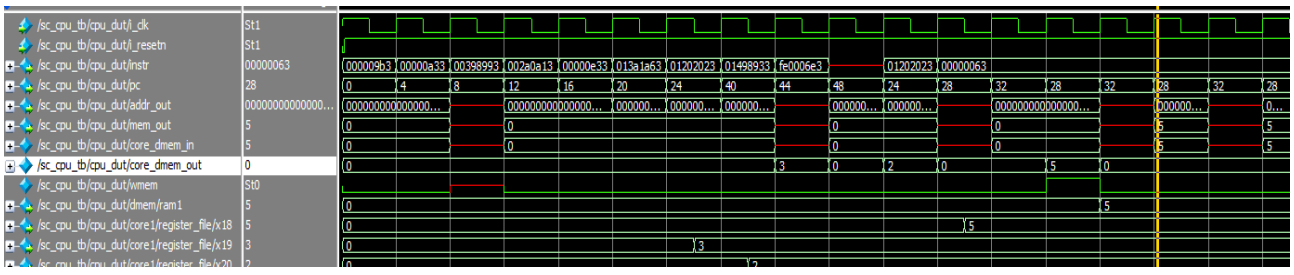
40 (Just In my instruction memory) : add x18 x19 x20

44 beq x0 x0 24 (Loop 24)

- 위의 시나리오를 **Machine language** 로 Compile하면 아래와 같다.

1. 000009B3
2. 00000A33
3. 00398993
4. 002A0A13
5. 00000E33
6. 013A1A63 (Branch destination = Pc + sign-ext(imm[12:1] , 0)
7. 01202023
8. 00000063 //dummies
9. 00000063
10. 00000063
11. 01498933
12. FE0006E3 (44-> 24이므로 -20 -> sign extension 시, 1 1111 1110 1100)

위와 같은 Test bench를 돌려보자. 모듈의 설명은 앞에서 진행했으므로, 생략한다. 따라서 txt file에 위의 instruction을 그대로 얹어주고, 총 cycle은 20사이클로 설정한 후, testbench를 돌리고 필요한 정보만 사진으로 캡처하면,



위와 같이, 우리가 생각한대로, PC shift를 하고, ALU를 진행한 후, ram1에 store하는 모습을 볼 수 있었다. (사진/tb file은 zip file로 같이 첨부하겠습니다.)

(모든 과정에서, instr은 Hexadecimal, 그외 register value 등은 Unsigned로 처리했습니다.)

2) Scenario_2 - Targeting a Data Hazard & verifying every instruction in PPT

- 이번에는 C언어 없이 간단하게 assembly로만 생각하자. 먼저 어떠한 register에 하나의 값을 저장하고, 이후 그것을 ram에 저장, 이후 그것을 load하는 과정을 생각해보자. 그렇게 어렵지 않다. 이때, (Data Hazard를 check 하기위해, ALU operation을 연속으로 해보자.)

  0 : add x19, x0, x0

  4 : add x20, x0, x0

  8 : addi x19, x19, 13

  12 : addi x20, x20, 5

  16 : sub x19,x19,x20

  20 : add x19,x19,x20

  24 : sw x19,0(x0)

  28 : lw x18,0(x0)

  32 : and x18, x18,x20

  36 : or x17,x18,x19

  40 : sll x16,x18,x19

  44 : srl x16,x18,x19

  48 : sra x16,x18,x19

  52 : andi x16,x18,2

  56 : ori x16, x18, 2

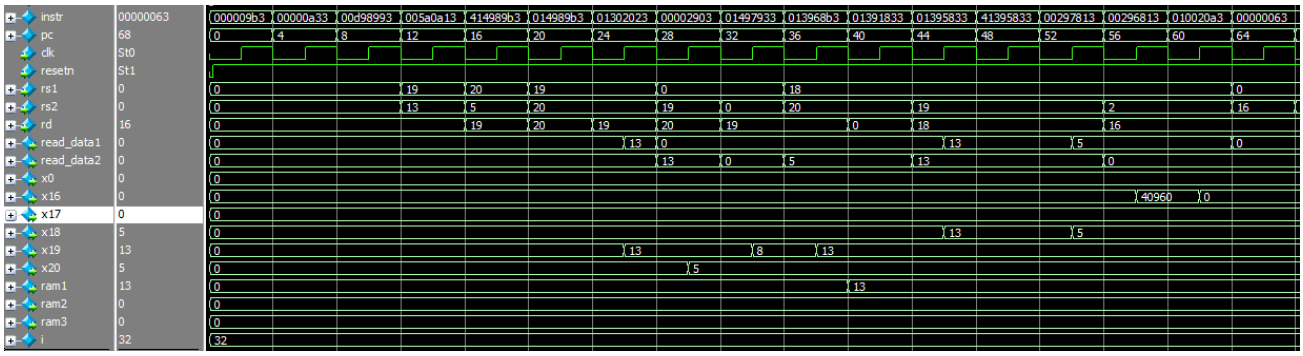
  60 : sw x16,1(x0)

- 위를 Machine language 로 compile 해주면,

  1. 000009B3
  2. 00000A33
  3. 00D98993
  4. 005A0A13
  5. 414989B3

6. 014989B3
7. 01302023
8. 00002903
9. 01497933
10. 013968B3
11. 01391833
12. 01395833
13. 41395833
14. 00297813
15. 00296813
16. 010020A3

과 같고, 이에 대하여 test bench를 진행하면,



와 같이 아주 정확하게 맞아 떨어지며, Forwarding unit이 정확하게 작동하여 위와 같이 CPU가 잘 동작하는 것을 확인할 수 있다. 또한 ram1, ram2에 정확히 저장이 된다. 이를 통해 모든 functionality와 data-Hazard 를 targeting한 processor에 대한 Validation이 완료되었다.

Chapter 3. Reference

- T.W.Seo (2021). *Computer Architecture*. Seoul, Korea: Hong pup science.
- J.H. Kung (2023) Asic design Course materials