CSED311 Computer Architecture

Assignment 4-2 Report:
Pipelined CPU with Control Flow Instructions

TEAM 28

20180154 전제민 jeminjeon@postech.ac.kr

20180916 임경빈 kbini@postech.ac.kr

2023/05/14

1. Introduction

이번 Lab4-2을 수행하며 Pipelined Multi-cycle CPU를 RTL로 구현하였다. Pipelined Multi-cycle CPU는 명령어 수행을 효율적으로 하기 위해 하나의 명령어 수행을 여러 단계로 나누고 각 단계에서 동시에 다른 명령어를 처리하도록 설계된 CPU를 말한다. 이번 랩에서는 control flow 명령어 또한 지원하도록 구현하였고 Branch prediction은 2-bit global predictor와 Gshare predictor로 구현하였으며 discussion 파트에서 이를 비교하였다.

이번 보고서에서는 Pipelined Multi-cycle CPU의 논리 회로 설계를 다루고자 한다.

- **Design 파트에서는** 각 모듈의 설계, 파이프라인 레지스터 설계, hazard detection 설계, data forwarding설계 및 branch predictor설계 에 대해 설명하였다.
- **Implementation 파트에서는** 각 모듈의 구체적인 구현을 설명하였다. 또한 branch predictor가 어떻게 구현되었는지 상세히 설명하였다.
- **Discussion 파트에서는** 2-bit global predictor와 Gshare predictor의 비교를 설명하였다.
- **Conclusion 파트에서는** 이번 구현을 통해 새롭게 알거나 깨닫게 된 점을 간단히 정리하였다.

2. Design

(1) Designs of Each Module

기존 lab 4-1 구현시에 사용된 모듈들을 재사용하였고 새롭게 branch predictor module,

corrected pc module을 추가하였다.

- GsharePredictor: 5비트 BHSR레지스터와 pc[6:2]를 xor 연산한 값으로 BTB를 인덱싱 하도록 설계된 branch predictor이다. (6) Branch prediction design part에서 자세히 설명하였다.
- **Corrected PC**: branch prediction이 틀린 경우에 올바른 타겟 pc로 branch 하는 기능을 하는 모듈이다. JAL, JALR 인스트럭션은 ALU 결과를, branch instruction은 bcond가 1일때는 ID_EX_pc+immediate, bcond가 0일때는 ID_EX_pc+4로 pc를 계산하도록 설계하였다. 여기서 계산된 pc를 사용하는 경우는 틀리게 branch prediction한 경우에만 사용된다.

(2) **Pipeline register design**

각 스테이지 사이에 이전 스테이지에서 결정된 값들을 저장하는 파이프라인 레지스터를 만들어 저장한다. Pc_to_reg control signal을 저장할 파이프라인 레지스터와 잘못 prediction 된 pc로 fetch 된 인스트럭션들을 버블로 만들기 위한 파이프라인 레지스터를 추가하였다.

- IF/ID register:

  잘못 prdiction된 PC임을 판단하고 난 뒤에는 이 PC로 fetch한 수행중인 인스트럭션들을 버블로 만들어야 되므로 IF/ID bubble을 추가하였다.

- ID/EX register:

  IF/ID bubble값이 이동 될 ID/EX bubble 레지스터를 추가하였다.

  ID_EX_pc_to_reg를 추가하였다.

- EX/MEM register:

  EX_MEM_pc_to_reg를 추가하였다.

- MEM/WB register:

  MEM_WB_pc_to_reg를 추가하였다.

(3) **Control signal design**

Lab 4-1 구현시에는 non-control flow insturction들만을 사용하였기 때문에 pc_to_reg control signal을 사용하지 않았으므로 새로 pc_to_reg control signal을 추가하였다. 또한 IsBranch, IsJAL, IsJALR control signal들을 추가하였다.

Pc_to_reg : 레지스터에 pc값을 작성할지 여부를 나타낸다. jal, jalr 명령어의 경우 pc_to_reg가 1이되고 WB stage에서 destination register에 PC+4를 저장한다.

(4) Hazard detection design

Hazard detection 로직은 Lab 4-1 구현시 사용한 로직을 동일하게 사용하였다.

(5) Data forwarding design

Data Forwarding 로직은 Lab 4-1 구현시 사용한 로직을 동일하게 사용하였다.

(6) Branch prediction design

Branch predictor는 Gshare predictor로 구현하였다. BHSR 레지스터가 모든 브랜치 인스트럭션들의 Taken, Not Taken 여부를 기록하도록 구현하였다. BHSR 은 총 5비트이므로 최근 5개 인스트럭션들의 Taken, Not taken여부를 가지고 있다. 그리고 그리고 BTB를 구현하였는데 BTB는 총 32 entries로 구현하였다. 각 entry별로 tag, BHT, data, valid를 가지고 있다. Tag는 상위 25비트, BHT는 2비트로 , data는 32비트 pc값을 저장하고, valid는 1비트로 설계하였다. BHT는 2-bit saturation counter로 설계하였다. 그리고 pc[6:2] 와 BHSR [4:0]을 xor연산하여 구한 인덱스로 BTB를 인덱싱하여 태그 비교 후 일치할 경우 bht의 예측값을 사용해 저장된 pc data를 사용하도록 설계하였다. BTB는 처음 reset 신호가 왔을 때 모두 0으로 초기화 되게 설계하였다.

다음 PC를 예측할 때는 BTB를 BHSR[4:0], pc[6:2]을 xor 연산하여 구한 인덱스로 인덱싱하고 만약 태그가 일치하고 valid가 1이라면 BTB에서 가져온 data를 next_pc로 사용하고 아니면 PC+4를 next_pc로 하였다. 예측이 맞는지 여부는 EX stage에서 ALU의 bcond를 통해 확인하게 된다. BHSR 레지스터 값의 업데이트는 BHSR 레지스터의 오른쪽 4비트를 1비트 left shift하고 ALU에서 계산된 Taken, not taken여부를 가장 오른쪽 비트에 추가한다. 만약 잘못 prediction 되었다면 IF_ID_bubble, ID_EX_bubble을 1로 write하여 IF, ID stage에서 실행중인 명령어들을 버블로 만들고 next_pc로 올바르게 분기된 pc값인 corrected pc를 사용하도록 설계하였다. 따라서 misprediction을 하게 되면 2 bubble이 생기고 올바르게 분기된 pc를 다음 cycle부터 사용되게 된다. 또한 이처럼 브랜치 예측이 잘못된 경우에는 BTB를 제대로 예측 된 값으로 태그비트 값과 데이터(pc)를 수정하도록 설계하였다.

3. Implementation

(1) Exceptions of Identical Modules

이전 Lab 4-1과 구현이 동일한 모듈인 PC, ALU, ALUControlUnit, ForwardingUnit, , HazardControlUnit, ImmediateGenerator, Memory, RegisterFile에 대한 내용은 제외하였다.

(2) Pipeline Registers

Control Flow Instruction에 대한 구현을 위해 파이프라인 레지스터에 필요한 데이터를 추가하였다. 추가된 파이프라인 레지스터는 다음과 같다.

- PC: 현재 스테이지에 해당하는 PC를 나타낸다. 이후 PC를 이용한 연산에 사용된다.
- predicted_pc: Branch Predictor에서 예상한 PC를 저장한다.
- predicted_taken: Branch Predictor에서 예상한 taken 결과를 저장한다. taken으로 저장
- bubble: 현재 스테이지가 bubble이 되어야 할 경우 이를 표시한다. bubble이 된 스테이지는 MEM 단계 이후로 레지스터파일과 메모리에 write하는 시그널이 모두 disable된다.
- pc_to_reg: JAL과 JALR 명령어의 경우 PC+4 값이 레지스터에 write되기 위해 이를 수행하도록 하는 signal이다.
- IsBranch: 현재 스테이지에 해당하는 명령어가 Branch 타입임을 나타낸다.
- IsJAL: 현재 스테이지에 해당하는 명령어가 JAL 타입임을 나타낸다.
- IsJALR: 현재 스테이지에 해당하는 명령어가 JALR 타입임을 나타낸다.

(3) ControlUnit

IsBranch, IsJAL, IsJALR 레지스터가 추가되었으므로 ID 스테이지에서 현재 인스트럭션이 branch 또는 JAL, JALR 타입인지 확인하고 알맞은 시그널을 output으로 내보내는 로직이 추가되었다.

```
if(opcode == `BRANCH)
    IsBranch = 1;

if(opcode == `JAL)
    IsJAL = 1;

if(opcode == `JALR)
    IsJALR = 1;

if(opcode ==`STORE)begin
    mem_write=1;
end
```

(4) TwoBitGlobalPredictor

아래는 2-bit Global Predictor를 구현한 모습이다. 먼저 BTB와 2-bit saturation state를 나타내는 레지스터 predictor_state를 선언하였다. current pc를 입력받으면 tag와 index를 분리하고 bcond의 값에 따라 taken 값을 set함으로써 이후 로직에 필요한 레지스터 값을 세팅한다.

태그의 값과 해당 index 엔트리의 BTB에 저장된 태그 값을 비교하여 이 둘이 같고, valid bit가 1이며 현재 2-bit saturation state가 2'b10 또는 2'b11로 taken을 가리키고 있다면 predicted_taken은 1이 되고, 즉 현재 branch가 taken 된 것으로 예측하고 타겟 pc는 BTB에 저장된 값으로 하여 output으로 내보낸다. 만약 위 조건을 충족하지 않는다면 not taken으로 예측하고 타겟 pc는 현재 pc+4가 된다.

이후에는 실제 branch의 taken 결과에 따라 BTB를 업데이트하는 로직이 있다. 만약 EX에서 살펴본 결과 해당 branch가 실제로 taken 되었다면 valid bit를 1로 바꾸고, 실제 target pc로 BTB를 업데이트 한다. 또한 해당 결과에 따라 2-bit saturation predictor의 state를 수정하게 된다.

```
//    reg [57:0] BTB [0:31];
    reg [59:0] BTB [0:31];
    reg [1:0] predictor_state;
    integer i;

    reg [24:0] tag;
    reg [4:0] index;
    reg taken;
    reg correct_prediction;

    always @(*) begin

        // setting proper reg values

        tag = current_pc[31:7];
        index = ID_EX_pc[6:2];

        if (IsJAL || IsJALR || (IsBranch && bcond))
            taken = 1;
        else
            taken = 0;

        // actual_pc == ID_EX_predicted_pc
        if (actual_pc == ID_EX_predicted_pc)
            correct_prediction = 1;
        else
            correct_prediction = 0;


    end
```

```
if((IsBranch || IsJAL || IsJALR) && !is_stall ) begin

    //BTB[index][57] <= 1;
    //BTB[index][56:32] <= tag;

    //if(correct_prediction)
        //BTB[index][31:0] <= actual_pc;
    //else
        //BTB[index][31:0] <= ID_EX_predicted_pc;

    if(taken) begin
        BTB[index][57] <= 1;
        //BTB[index][56:32] <= tag;
        BTB[index][56:32] <= ID_EX_pc[31:7];
        BTB[index][31:0] <= actual_pc;
    end

    case(predictor_state)
        2'b00: begin
            if(taken)predictor_state <= 2'b01;
            else predictor_state <= 2'b00;
        end
        2'b01: begin
            if(taken) predictor_state <= 2'b10;
            else predictor_state <= 2'b00;
        end
        2'b10: begin
            if(taken) predictor_state <= 2'b11;
            else predictor_state <= 2'b01;
        end
        2'b11: begin
            if(taken) predictor_state <= 2'b11;
            else predictor_state <= 2'b10;
        end
    endcase
end
```

```
always @(*) begin
    if((current_pc[31:7] == BTB[current_pc[6:2]][56:32]) && (predictor_state == 2'b10 || predictor_state == 2'b11) && (BTB[current_pc[6:2]][57] == 1)) begin
        predicted_pc = BTB[current_pc[6:2]][31:0];
        predicted_taken = 1;
    end
    else begin
       predicted_pc = current_pc + 4;
       predicted_taken = 0;
    end
end
```

아래는 BTB의 각 엔트리 별로 BHT가 존재하는 경우의 구현이다.

```
//    reg [57:0] BTB [0:31];
    reg [59:0] BTB [0:31];
    reg [1:0] predictor_state;
    integer i;

    reg [24:0] tag;
    reg [4:0] index;
    reg taken;
    reg correct_prediction;

    always @(*) begin

        // setting proper reg values

        tag = current_pc[31:7];
        index = ID_EX_pc[6:2];

        if (IsJAL || IsJALR || (IsBranch && bcond))
            taken = 1;
        else
            taken = 0;

        // actual_pc == ID_EX_predicted_pc
        if (actual_pc == ID_EX_predicted_pc)
            correct_prediction = 1;
        else
            correct_prediction = 0;


    end
```

```
if((IsBranch || IsJAL || IsJALR) && !is_stall ) begin

    if(taken) begin
        BTB[index][57] <= 1;
        //BTB[index][56:32] <= tag;
        BTB[index][56:32] <= ID_EX_pc[31:7];
        BTB[index][31:0] <= actual_pc;
    end

    case(BTB[index][58:57])
        2'b00: begin
            if(taken)BTB[index][58:57] <= 2'b01;
            else BTB[index][58:57] <= 2'b00;
        end
        2'b01: begin
            if(taken) BTB[index][58:57] <= 2'b10;
            else BTB[index][58:57] <= 2'b00;
        end
        2'b10: begin
            if(taken) BTB[index][58:57] <= 2'b11;
            else BTB[index][58:57] <= 2'b01;
        end
        2'b11: begin
            if(taken) BTB[index][58:57] <= 2'b11;
            else BTB[index][58:57] <= 2'b10;
        end
    endcase
end
```

```
always @(*) begin
    if((current_pc[31:7] == BTB[current_pc[6:2]][56:32]) && (BTB[current_pc[6:2]][58:57] == 2'b10 || BTB[current_pc[6:2]][58:57] == 2'b11)
          && (BTB[current_pc[6:2]][57] == 1)) begin
          predicted_pc = BTB[current_pc[6:2]][31:0];
          predicted_taken = 1;
    end
    else begin
       predicted_pc = current_pc + 4;
       predicted_taken = 0;
    end
end
```

(5) GsharePredictor

Gshare predictor의 경우 input으로 들어온 index에 BHSR 에 저장된 이전 branch taken 기록들이 XOR 연산되어 BTB의 인덱스로 사용된다. 이전 2-bit Global Predictor과 달리 XOR 연산을 이용한 index를 구하는 로직이 추가되었음을 알 수 있다. 또한 output을 결정하는 로직에서 current_XORindex와 ID_EX_XORindex를 이용해 인덱싱을 하는 것을 알 수 있다. BTB를 업데이트하는 로직도 이전과 마찬가지로 EX에서 resolve한 branch taken result에 따라 BTB 해당 엔트리의 valid를 1로 set하고 실제 타겟 pc와 2-bit saturation predictor state를 업데이트한다. 다만 BHSR에서 실제 branch 결과를 left shift하는 과정이 추가되었다. 또한 각 entry별로 BHT를 가지도록 구현하였다.

```
always @(*) begin

    // setting proper reg values

    current_tag = current_pc[31:7];
    ID_EX_tag = ID_EX_pc[31:7];
    current_index = current_pc[6:2];
    ID_EX_index = ID_EX_pc[6:2];
    current_XORindex = current_index ^ BHSR;
    ID_EX_XORindex = ID_EX_index ^ BHSR;

    if (IsJAL || IsJALR || (IsBranch && bcond))
        taken = 1;
    else
        taken = 0;


end
```

```
if((IsBranch || IsJAL || IsJALR) && !is_stall ) begin

    if(taken) begin

        BTB[ID_EX_XORindex][62:58] <= ID_EX_pc[6:2];

        BTB[ID_EX_XORindex][57] <= 1;
        BTB[ID_EX_XORindex][56:32] <= ID_EX_tag;
        BTB[ID_EX_XORindex][31:0] <= actual_pc;
    end


    if(taken)begin
        BHSR <= {BHSR[3:0],1'b1};
    end
    else begin
        BHSR <= {BHSR[3:0],1'b0};
    end
    case(BTB[ID_EX_XORindex][64:63])
        2'b00: begin
            if(taken) BTB[ID_EX_XORindex][64:63] <= 2'b01;
            else BTB[ID_EX_XORindex][64:63] <= 2'b00;
        end
        2'b01: begin
            if(taken) BTB[ID_EX_XORindex][64:63] <= 2'b10;
            else BTB[ID_EX_XORindex][64:63] <= 2'b00;
        end
        2'b10: begin
            if(taken) BTB[ID_EX_XORindex][64:63] <= 2'b11;
            else BTB[ID_EX_XORindex][64:63] <= 2'b01;
        end
        2'b11: begin
            if(taken) BTB[ID_EX_XORindex][64:63] <= 2'b11;
            else BTB[ID_EX_XORindex][64:63] <= 2'b10;
        end
    endcase
```

```
always @(*) begin
    if((current_tag == BTB[current_XORindex][56:32]) && (BTB[ID_EX_XORindex][64:63] == 2'b10 || BTB[ID_EX_XORindex][64:63] == 2'b1
       && (BTB[current_XORindex][57] == 1)&&(BTB[current_XORindex][62:58]==current_pc[6:2] )) begin
            predicted_pc = BTB[current_XORindex][31:0];
            predicted_taken = 1;
    end
    else begin
       predicted_pc = current_pc + 4;
       predicted_taken = 0;
    end
end
```

(6) CorrectedPC

만약 IF 단계에서 수행한 Branch prediction이 틀렸을 경우, 즉 taken 여부가 틀렸거나 taken 여부가 맞았어도 target pc가 틀렸을 경우 올바른 타겟 PC를 전달하는 목적으로 구현한 모듈이다. 먼저 JAL, JALR 타입의 경우 EX 스테이지에서 나온 ALU result를 타겟 pc로 전달한다. 만약 Branch 타입이라면 bcond가 1일 경우 ID_EX_pc + immediate를, 0일 경우 ID_EX_pc+4를 전달한다. next_pc가 CorrectedPC에서 나올지 Predictor에서 나올지에 대한 로직은 CPU에서 결정된다.

```
module CorrectedPC(
                    input [31:0] alu_result,
                    input [31:0] current_pc,
                    input [31:0] immediate,
                    input [31:0] ID_EX_predicted_pc,
                    input isJAL,
                    input isJALR,
                    input isBranch,
                    input bcond,
                    input is_stall,
                    output reg [31:0] corrected_pc
                    );

    always @(*) begin
        if(isJAL || isJALR) begin
            corrected_pc = alu_result;
        end
        else if(isBranch) begin
            if(bcond)
                corrected_pc = current_pc + immediate;
            else
                corrected_pc = current_pc + 4;
        end
        else begin
            corrected_pc = 10'b0101010101; // for debug
        end
    end

endmodule
```

(7) CPU

위에서 새롭게 추가하거나 수정된 Module를 하나로 link하는 과정을 수행하였다. 코드가 길고 수정이 일어난 곳이 sparse하기 때문에 코드 첨부는 생략하였다.

4. Discussion

(1) Comparison of Gshare to 2-bit global prediction

2-bit global predictor를 이용하여 테스트 벤치를 실행한 결과 loop_mem.txt를 적용했을 때 299 cycles, recursive_mem.txt를 적용했을 때 1070 cycles가 소요되었다.

loop_mem.txt		recursive_mem.txt	
TOTAL CYCLE	299	TOTAL CYCLE	1070
0	00000000	0	00000000
1	00000000	1	00000000
2	00002ffc	2	00002ffc
3	00000000	3	00000000
4	00000000	4	00000000
5	00000000	5	00000000
6	00000000	6	00000000
7	00000000	7	00000000
8	00000000	8	00000000
9	00000000	9	00000000
10	00000000	10	0000000d
11	00000000	11	00000000
12	00000000	12	00000000
13	00000000	13	00000000
14	0000000a	14	00000001
15	00000009	15	0000000d
16	0000005a	16	00000015
17	0000000a	17	0000000a
18	00000000	18	00000000
19	00000000	19	00000000
20	00000000	20	00000000
21	00000000	21	00000022
22	00000000	22	00000000
23	00000000	23	00000037
24	00000000	24	00000059
25	00000000	25	00000000
26	00000000	26	00000000
27	00000000	27	00000000
28	00000000	28	00000000
29	00000000	29	00000000
30	00000000	30	00000000
31	00000000	31	00000000

Gshare predictor를 이용하여 테스트 벤치를 실행한 결과 loop_mem.txt를 적용했을 때 323 cycles, recursive_mem.txt를 적용했을 때 1209 cycles가 소요되었다.

loop_mem.txt		recursive_mem.txt	
TOTAL CYCLE	323	TOTAL CYCLE	1209
0	00000000	0	00000000
1	00000000	1	00000000
2	00002ffc	2	00002ffc
3	00000000	3	00000000
4	00000000	4	00000000
5	00000000	5	00000000
6	00000000	6	00000000
7	00000000	7	00000000
8	00000000	8	00000000
9	00000000	9	00000000
10	00000000	10	0000000d
11	00000000	11	00000000

12 00000000	12 00000000
13 00000000	13 00000000
14 0000000a	14 00000001
15 00000009	15 0000000d
16 0000005a	16 00000015
17 0000000a	17 0000000a
18 00000000	18 00000000
19 00000000	19 00000000
20 00000000	20 00000000
21 00000000	21 00000022
22 00000000	22 00000000
23 00000000	23 00000037
24 00000000	24 00000059
25 00000000	25 00000000
26 00000000	26 00000000
27 00000000	27 00000000
28 00000000	28 00000000
29 00000000	29 00000000
30 00000000	30 00000000
31 00000000	31 00000000

cycle 수만 놓고 판단하였을 때 2-bit global predictor가 Gshare predictor에 비해 좀 더 좋은 성능을 보여주고 있다. 2-bit global predictor의 경우 프로그램 전체에 대하여 branch taken/not taken 여부에 따라 state가 달라지게 된다. 또한 PC의 [6:2] index에 대하여 태그가 정해지고 해당 태그에 대한 타겟 버퍼에서 예상되는 PC를 가져오게 된다.

한편 Gshare의 경우 GSHR 레지스터에 기록된 최근 Taken 여부 패턴과 PC의 인덱스가 XOR 연산되어 이것을 이용하여 BTB의 해당 엔트리에 인덱싱하여 타겟 버퍼에서 예상되는 PC를 가져오는 구조인데, 프로그램 특성 상 최근 Taken 패턴이 앞으로의 branch target을 예측하는데 그다지 도움이 되지 못해 penalty가 늘어나 비교적 낮은 성능을 낸 것으로 보인다.

(2) Improvement of Predictors

이번 Lab에서는 Gshare predictor, two-bit global branch predictor를 구현하였는데 각 엔트리마다 BHT를 가지는 two-bit branch predictor로 구현한 경우에는 사이클 수가 1060사이클로 two-bit global brach predictor, Gshare predictor보다 적은 사이클이 걸렸다. 만약 이처럼 two-level Branch predictor 등을 구현한다면 반복되는 경향을 더 잘 반영하여 조금 더 높은 분기 예측률을 가질 수 있을 것으로 기대가 된다.

(3) Managing Pipeline Register Data

저번 Lab 4-1에 이어서 파이프라인 CPU를 구현하였는데, 파이프라인 레지스터에 저장해야 할 데이터가 늘어나다 보니 각각의 데이터에 대해 초기화와 데이터 path 관리를 해주는 것이 까다로웠다. 실제로 이번에 새로 추가한 pc_to_reg 컨트롤 시그널의 경우 중간에 데이터 path 관리를 제대로 하지 못해 JAL 명령어가 레지스터에 잘못된 값을 write하는 오류가 발생하기

도 하였다. 결국 모든 파이프라인 레지스터를 관리하는 리스트를 만들고 각각의 레지스터가 어디서 값을 가져오는지 추적하여 문제를 해결할 수 있었다.

5. Conclusion

이번 Lab에서는 Control Flow insturction들을 지원하는 Pipelined CPU를 구현하였다. Pipeline register를 통해 스테이지별로 다른 인스터력션을 수행함으로써 throuput을 늘릴 수 있음을 알게 되었다. Branch predictor는 2-bit global predictor와 Gshare predictor로 구현하였는데 어떤 branch predictor 를 사용하여 다음 pc를 predict하는지에 따라 cycle 수가 많이 날 수 있음을 알게 되었다. Branch prediction 정확도가 성능에 영향을 미칠 수 있음을 알게 되었다.