마이크로프로세서 설계 무작정 따라하기 (3)

KAIST 전자전산학과 박사과정 배영돈(donny@ics.kaist.ac.kr)

5.2 컨트롤 회로의 설계

가) Fetch unit 의 설계

Fetch unit 은 메모리로부터 명령어를 읽어오는 기능을 수행한다. Fetch unit 의 구조는 명령어의 구조에 따라 결정된다. 명령어 크기(bit width)은 입출력 핀의 개수와 프로그램 메모리의 크기를 결정한다. 쉽게 설명하면 32 비트 명령어보다 16 비트 명령어가 적은 전력을 소비한다. 또한, 가변적인 크기의 명령어를 사용하는 프로세서들도 있다. 예를 들어 자주 사용하는 명령어는 8 비트로 하고, 복잡하고 적게 사용되는 명령어는 16 비트, 24 비트로 만드는 것이다. 그 결과, 명령어를 가져오는(fetch) 과정에서의 전력 소모를 줄이고 프로그램 메모리의 크기를 줄일 수 있다. 저전력을 지향하는 임베디드(embedded) 프로세서는 16 비트 명령어를 가장 많이 사용하고 있다.
SimpleCore 는 단순한 파이프라인 구조를 사용하므로 16 비트 레지스터만으로 Fetch Unit 을 설계할 수 있다. Fetch Unit 의 출력은 Decode unit 으로 전달된다.

Fetch unit 의 RTL 코드

```
module   fetch (
         clk,
         dIn,
         fInst
         );

input    clk;  // main clock
input [15:0]   dIn;  // instruction data
output   [15:0]   fInst;   // fetch data

reg   [15:0]   fInst;

always@(posedge clk)
begin
         fInst = dIn;
end

endmodule
```

나) Decode unit 의 설계

Decode unit 은 명령어를 해석하는 동작을 수행한다. Decoder unit 의 기능을 이해하기 위해, 명령어 구조를 설계하는 방법을 살펴보자.
명령어를 실행하기위해선 많은 양의 정보가 필요하다. 이러한 정보를 모두 명령어에 저장하면 명령어의 크기가 커지게 된다. 앞에서 설명한 것과 같이 명령어의 크기는 작을수록 장점을 갖는다. 필요한 정보를 정해진 명령어의 크기에 저장해야 하므로 명령어는 필요한 정보의 압축된 형태라고 설명 할 수 있다. 따라서, 명령어를 실행하기 위해선 이러한 압축된 정보를 반대로 해석하는 동작이 필요하며 그 기능을

수행하는 것이 Decode unit 이다.
명령어는 크게 두 가지 부분으로 구성되어있다. 즉, 해당 명령어의 실행에 꼭 필요한 정보(예, op code, 레지스터 주소)와 명령어를 서로 구분하기 위한 정보로 구성되어있다.

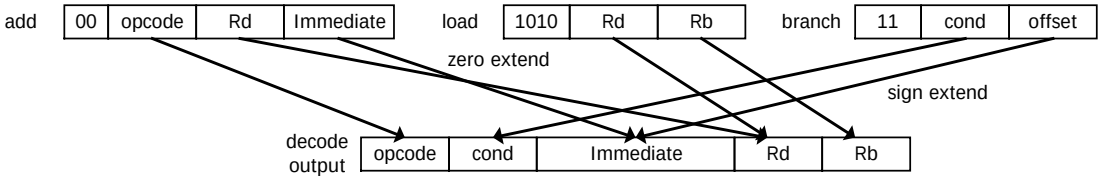

그림 1. 명령어 decode 과정

그림 1 은 명령어의 decode 과정을 설명한 것이다. 그림과 같이 Decode unit 은 모든 명령어로부터 명령어 실행에 필요한 정보를 추출한다. 이 때, 각각의 정보는 필요에 따라 부호확장(sign extend)이 된다. 이 결과(decode output)는 파이프라인 레지스터에 저장되어 execute stage 에서 사용하게 된다.

동일한 목적의 정보가 명령어에 따라 저장된 위치가 달라지는 경우에는 그림 2 와 같이 MUX 를 사용하고 MUX 를 제어하기 위한 선택 신호를 명령어로부터 생성해야 한다.

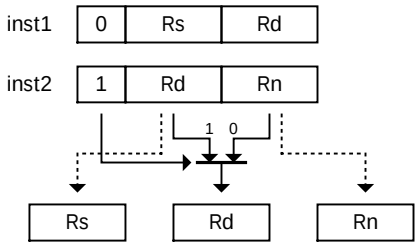

그림 2. 명령어 decode 과정

Simple core 의 경우 최대한 이러한 MUX 가 필요 없도록 명령어 구조를 정의 하였다. 한가지 예외는 바로 ALU immediate 형식의 명령어와 Branch 명령어가 각각 Immediate 와 Offset 이란 정보를 서로 다른 위치에 저장하고 있다. Immediate 와 Offset 은 동시에 사용되지 않는 정보이기 때문에 pipeline register 의 공간을 공유하여 사용할 수 있으며 부호확장 방식의 차이를 제외하고 유사한 목적을 갖고있다.

	15	14	13	12	11	10 ~ 7	6 ~ 4	3 ~ 0	Pseudo code
ALU imm.	0	0	Opcode			Rd	Immediate		Rd = Rd op Immediate
ALU reg.	0	1	Opcode			Rd	Rs2	Rs1	Rd = Rs1 op Rs2
Shift/Rotate	1	0	0	Shift		Rd	Rs1	Rs1	Rd = Rs1 <Shift> Rs2
Load	1	0	1	0	-	Rd	-	Rb	Rd = Mem[Rb]
Store	1	0	1	1	-	Rd	-	Rb	Mem[Rb] = Rd
Branch	1	1	Cond			Offset			if(cond) PC = PC + Offset
Multiply	1	1	1	1	-	Rd	Rs2	Rs1	Rd = Rs1 * Rs2

그림 3. SimpleCore 의 명령어 구조

Decoder unit 을 Verilog 로 기술하는 과정 중 명령어를 구분하는 부분을 살펴보자.
다음과 같이 case 문을 이용하여 간단히 설계할 수 있다.

이제 각 명령어를 구분할 수 있는 명령어 ID(instruction ID)가 만들어 졌다.

Decode unit 의 RTL 코드(명령어 구별)

```
`define  INST_ALUI   3'd0  // ALU immediate operand
`define  INST_ALUR   3'd1  // ALU register operand
`define  INST_SHRO   3'd2  // shift and rotate
`define  INST_LOAD   3'd3  // load
`define  INST_STORE  3'd4  // store
`define  INST_BRANCH 3'd5  // branch
`define  INST_MUL 3'd6  // multiply

input [15:0]   fInst;   // fetch data
output   [ 2:0]   instId;  // instruction ID

/* instruction ID */
always@(fInst)
begin
        casex(fInst[15:12])
                4'b00?? : instId <= `INST_ALUI;  // ALU imm.
                4'b01?? : instId <= `INST_ALUR;  // ALU reg.
                4'b100? : instId <= `INST_SHRO;  // Shift/Rotate
                4'b1010 : instId <= `INST_LOAD;  // Load
                4'b1011 : instId <= `INST_STORE; // Store
                4'b110? : instId <= `INST_BRANCH;   // Branch
                4'b11?0 : instId <= `INST_BRANCH;   // Branch
                4'b1111 : instId <= `INST_MUL;   // Multiply
        endcase
end
```

그럼 이제 명령어 ID 를 사용하여 명령어로부터 실행에 필요한 정보를 추출해보자.
앞서 설명한 것과 같이 Immediate 를 제외한 나머지 경우 assign 문을 사용하여 정해진 위치로부터 추출이 가능하다. instId 가 INST_BRANCH 인 경우, 출력 imm 에 부호확장(sign extension)된 정보가 저장된다. 또한, immediate 사용을 나타내는 immFlag 와 compare 명령에서 writeback 을 하지 않기 위한 cmpFlag, MSR, MRS 명령어에서 status register 를 제어하기 위한 srOEn, srWbEn 등의 정보가 생성된다.

Decode unit 의 RTL 코드(정보 추출)

```
output   [ 2:0]   opcode;     // op code
output   [ 1:0]   shift;      // shift type
output   [ 3:0]   rs1Idx;     // Rb/Rs1 Index
output   [ 3:0]   rs2Idx;     // Rs2 Index
output   [ 3:0]   rdIdx;      // Rd Index
output   [15:0]   imm;     // immediate data
output       immFlag;   // immediate operand flag
output      cmpFlag;    // compare flag (update status register, no writeback)
output      branchFlag; // branch flag
output      exitFlag;      // end of simulation flag
output      srOEn, srWbEn; // status register read/write enable

/* field extraction */
assign   opcode   = fInst[13:11];
assign   shift = fInst[12:11];
assign   rs1Idx   = fInst[ 3: 0];
assign   rs2Idx   = {1'b0, fInst[ 6: 4]};
assign   rdIdx = fInst[10: 7];
assign   imm   = (instId == `INST_BRANCH) ?
                 {{4{fInst[11]}}, fInst[11:0]} :  // sign extension
                 {9'b0, fInst[6:0]};     // zero extension
assign   immFlag  = (instId == `INST_ALUI);
assign   cmpFlag  = (fInst[15:11] == 5'b00101);
assign   branchFlag = (instId == `INST_BRANCH);
assign   exitFlag = (fInst[15:11] == 5'b00111); // end of simulation
assign   srWbEn   = (fInst[15:11] == 5'b00110) || (fInst[15:11] == 5'b00101); // MSR or CMP
assign   srOEn = (fInst[15:11] == 5'b00111); // MRS
```

다) Execute unit 의 설계

Execute stage 의 대부분의 동작은 datapath 에서 수행되며, control 회로는 datapath 의 각 블록을 제어하기 위한 제어신호를 생성한다. 이 제어 신호들은 명령어에 따라 달라지게 된다. 표 1 은 명령어에 따른 제어신호를 나타내고 있다. 이렇게 명령어에 따라 제어신호를 결정하는 과정이 Execute unit 의 설계를 의미 하며, 제어신호가 모두 결정되면 Execute unit 의 설계는 거의 완성된 것이다.

표 1. 명령어에 따른 제어신호

명령어	ALU 동작	ALU 출력	shifter 출력	mul. 출력	operand A	operand B
ALU immediate	op code	enable	disable	disable	Rd	Immediate
ALU register	op code	enable	disable	disable	Rs1	Rs2
Shift and rotate	don't care	disable	enable	disable	Rs1	Rs2
Load	MOV	disable	disable	disable	Rb (Rs1)	don't care
Store	MOV	disable	disable	disable	Rb (Rs1)	Rd
Branch	ADD	enable	disable	disable	PC	Immediate
Multiply	don't care	disable	disable	enable	Rs1	Rs2

이제 RTL 코드를 기술해보도록 하자. case 문을 사용하여 표 1 의 내용을 그대로 옮기는 작업에 불과하다.

Execute unit 의 RTL 코드(명령어에 따른 제어신호 생성)

```
// operand A control
`define  ARD   2'b00 // operand A : Rd
`define  ARS1  2'b01 // operand A : Rs1
`define  APC   2'b10 // operand A : PC

// operand B control
`define  BIM   2'b00 // operand B : Immediate
`define  BRS2  2'b01 // operand B : Rs2
`define  BRD   2'b10 // operand B : Rd

always@(instId or opcode)
begin
casex(instId)
// control signals :          |opcode   |aluOEn |shiftOEn  |mulOEn|opA  |opB  |wbEn  |iAreg |dAreg  |dInCtl  |dOutCtl
    `INST_ALUI   : control <= { opcode,    1'b1,     1'b0,    1'b0, `ARD , `BIM ,  1'b1,   1'b0,   1'b0,    1'b0,    1'b0};
    `INST_ALUR   : control <= { opcode,    1'b1,     1'b0,    1'b0, `ARS1, `BRS2,  1'b1,   1'b0,   1'b0,    1'b0,    1'b0};
    `INST_SHRO   : control <= { 3'bx,      1'b0,     1'b1,    1'b0, `ARS1, `BRS2,  1'b1,   1'b0,   1'b0,    1'b0,    1'b0};
    `INST_LOAD   : control <= {`OP_MOV,    1'b0,     1'b0,    1'b0, `ARS1, 2'bx,   1'b1,   1'b0,   1'b1,    1'b1,    1'b0};
    `INST_STORE  : control <= {`OP_MOV,    1'b0,     1'b0,    1'b0, `ARS1, `BRD ,  1'b0,   1'b0,   1'b1,    1'b0,    1'b1};
    `INST_BRANCH : control <= {`OP_ADD,    1'b1,     1'b0,    1'b0, `APC , `BIM ,  1'b0,   1'b1,   1'b0,    1'b0,    1'b0};
    `INST_MUL    : control <= { 3'bx,      1'b0,     1'b0,    1'b1, `ARS1, `BRS2,  1'b1,   1'b0,   1'b0,    1'b0,    1'b0};
endcase
end

assign  {opcodeTmp, aluOEn, shiftOEn, mulOEn, opA, opB, wbEnTmp, iAregCtl, dAregCtl,dInCtl, dOutCtl}
= control;
```

자 이제 datapath 의 각 블록을 제어할 수 있는 신호까지 완성이 되었다. 그러나, 이 제어 신호들은 datapath 에 그대로 인가될 수 없다. 예를 들어 opcode 신호를 본 강좌의 지난 회(2 회)에서 정의된 ALU 의 제어신호에 맞추어야 ALU 블록이 올바르게 동작하게 된다. shifter 의 제어신호와 operand A,B 의 제어신호도 적절히 변환시켜주어야 한다.(표 2, 표 3)

그러면, 왜 Decode unit 에서부터 datapath 에 직접 인가될 수 있는 신호를 생성하지 않고 이러한 변환과정을 거치는가. 물론 처음부터 datapath 에 맞추어 제어신호를 정의 하는 것도 설계방법 중 한가지이다. SimpleCore 의 경우 RTL 코드의 가독성(redability)를 높이고 디버깅이 용이한 코드를 만들기 위하여 이와 같은 방법을 선택하였다.

표 2. ALU 의 제어신호

aluCtl 값	ALU 동작
000	aluOut = aluAIn
001	aluOut = aluBIn
010	aluOut = aluAIn and aluBIn
011	aluOut = aluAIn or aluBIn
10x	aluOut = aluAIn + aluBIn
11x	aluOut = aluAIn - aluBIn

표 3. shifter 의 제어신호

shiftCtl 값	쉬프터 동작
0xx	logical shift left (LSL)
100	logical shift right (LSR)
101	arithmetic shift right (ASR)
11x	rotate (ROR)

Execute unit 의 RTL 코드(Datapath 제어신호의 생성)

```
assign   rdAOEn = ~srOEn;
assign   rdBOEn = (opB != `BIM);
assign   wbEn   = (cmpFlag == 1'b0 && wbEnTmp == 1'b1);// no writeback for compare instructions
assign   immOEn = (opB == `BIM);

// opAIdx
always@(opA or rdIdx or rs1Idx)
begin
   casex(opA)
      `ARD  : opAIdx <= rdIdx;
      `ARS1 : opAIdx <= rs1Idx;
      `APC  : opAIdx <= 4'b1111;
      default: opAIdx <= 4'bx;
   endcase
end

// opBIdx
always@(opB or rdIdx or rs2Idx)
begin
   casex(opB)
      `BIM  : opBIdx <= 4'bx;
      `BRS2 : opBIdx <= rs2Idx;
      `BRD  : opBIdx <= rdIdx;
      default: opBIdx <= 4'bx;
   endcase
end

assign   wbIdx = rdIdx;

// aluCtl
always@(opcodeTmp or immFlag)
begin
   casex({opcodeTmp, immFlag})
      {`OP_MOV,1'b0} : aluCtl <= 3'b000;
      {`OP_MOV,1'b1} : aluCtl <= 3'b001;
      {`OP_ADD,1'b?} : aluCtl <= 3'b100;
      {`OP_SUB,1'b?} : aluCtl <= 3'b11x;
      {`OP_AND,1'b?} : aluCtl <= 3'b01x;
      {`OP_OR ,1'b?} : aluCtl <= 3'b011;
      {`OP_CMP,1'b?} : aluCtl <= 3'b110;
      {`OP_MSR,1'b?} : aluCtl <= 3'b000;
      {`OP_MRS,1'b?} : aluCtl <= 3'b000;
   endcase
end

always@(shift)
begin
   casex(shift)
      `SH_LSL  : shiftCtl <= 3'b0xx;
```

```
      `SH_LSR  : shiftCtl <= 3'b100;
      `SH_ASR  : shiftCtl <= 3'b101;
      `SH_ROR  : shiftCtl <= 3'b11x;
   endcase
end
```

이제 Execute unit 은 Datapath 를 위한 제어신호를 출력한다. 이제 컨트롤 회로설계의 마지막 단계가 남았다. 바로 파이프라인이다.

라) 파이프라인 설계

SimpleCore 는 그림 4 와 같은 파이프라인 구조를 가지고 있다.
기본적으로 각 stage 와 stage 사이에 파이프라인 레지스터를 삽입하는 것으로 시작한다.

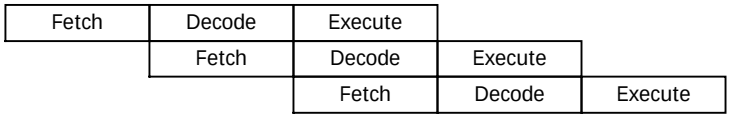

그림 4. SimpleCore 의 파이프라인 구조

그림 5 는 파이프라인 레지스터를 삽입한 구조를 나타내고 있다. 하지만, 이 구조에서 한가지 개선점을 찾을 수 있다. 바로 execute stage 에 수행할 일이 너무 많다는 것이다. 파이프라인 구조를 3 stage 로 단순화하면서 이미 execute stage 를 수행하기 위해 상대적으로 긴 시간이 필요하게 되었다. 그림 5 의 점선은 execute stage 에 critical path 를 나타내고 있다. Execute unit 에서 제어 신호들이 생성되고 나서 비로서 datapath 가 올바른 동작을 하게 되기 때문이다.

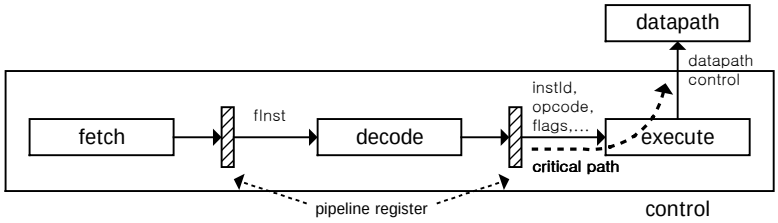

그림 5. 파이프라인 레지스터의 삽입

이러한 문제를 해결하기 위하여 Execute unit 을 decode stage 로 이동하였다(그림 6). 즉, decode stage 에서 execute stage 에 필요한 제어 신호들을 미리 만들어놓고 execute stage 에서는 바로 datapath 를 동작시키게 되는 것이다. 그 결과 프로세서의 동작 속도를 높일 수 있게 된다.

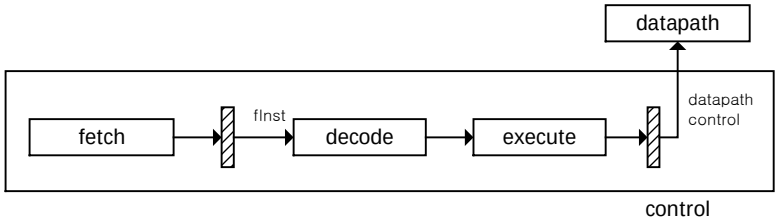

그림 6. 파이프라인 레지스터의 삽입(동작 속도의 개선)

파이프라인 설계의 가장 큰 걸림돌은 바로 해저드(hazard)이다. 해저드에는 크게 분기 해저드(branch

hazard)와 데이터 해저드(data hazard)가 있다. 데이터 해저드는 전방 전달(forwarding)을 이용하여 해결하는데 다소 까다로운 작업이다. SimpleCore 의 경우 데이터 해저드가 발생하지 않는다. 그 이유는 레지스터를 읽는 동작과 쓰는 동작인 한 stage 에서 이루어지기 때문이다. 파이프라인 설계를 쉽고 이해하기 쉽도록 하기 위하여 SimpleCore 의 구조를 3 단계 파이프라인으로 결정한 것이다. 다음으로 분기 해저드에 대하여 살펴보자. 그림 7 은 branch 명령어의 수행을 나타낸 것이다. 명령어 주소(address)가 불연속적인 순간이 발생하기 때문에 미리 fetch, decode 된 명령어를 수행하지 않고, 새로운 명령어를 읽어와야 한다(pipeline refill). (비교. 지연분기(delayed branch)) 이러한 동작을 위하여 두 가지 제어신호(flush, refill)를 생성하였다(그림 7).

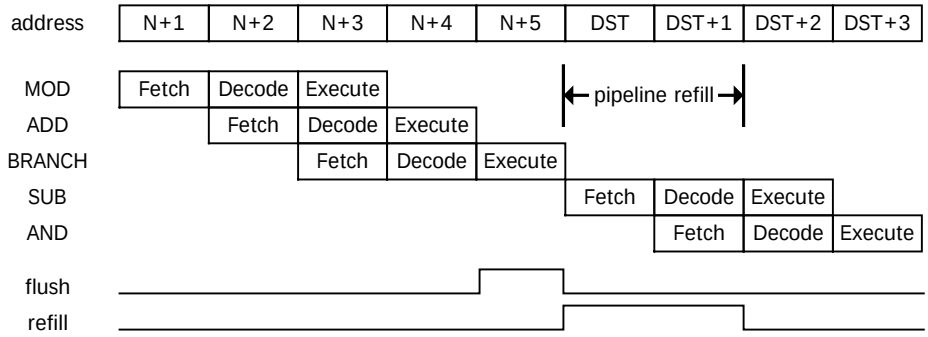

그림 7. Branch 명령어의 수행

지금까지 컨트롤 회로의 설계에 대하여 알아보았다. 지면관계로 RTL 코드 전체를 넣지 못했으며, 대신 홈페이지(http://www.donny.co.kr/simplecore)를 통해 제공하고자 한다.

다음 회에 계속됩니다. 다음 회에서는 프로세서 전체의 시뮬레이션 방법을 설명합니다.