CA2_TermProject_Analysis&Implement Report

컴퓨터학과 2020320156 허준우
컴퓨터학과 2021320175 박수빈

#1 Analyze given multi-cycle ARM processor source code

<module armreduced>

```
module armreduced(
    input clk,
    input reset,
    output[31:0] pc,
    input[31:0] inst,
    input nIRQ,
    output [3:0] be,
    output[31:0] memaddr,
    output memwrite,
    output memread,
    output[31:0] writedata,
    input[31:0] readdata);
```

모듈의 input

: clock signal(clk), reset signal(reset), interrupt signal(nIRQ), read data(readdata) and instruction(inst)

모듈의 output

: the program counter(pc), branch enable(be), memory address(memaddr), memory write enable(memwrite), memory read enable(memread), write data(writedata)

<wire>

```
    //signals
  wire IRwrite,regwrite,NZCVwrite;
  wire [1:0] regdst,regsrc,ALUsrcA,immsrc,ALUsrcB;
  wire [2:0] ALUop;
  wire [3:0] instop;
  wire regBdst;


  //wires_out
  wire[31:0] imm;

  //wires_in
  wire [3:0] ALUflags;
  wire [31:0] ALUresult,readA,readB;

  wire [3:0] RFdst [2:0];
  wire [31:0] RFsrc [3:0];
  wire [31:0] ALUnum1 [2:0];
  wire [31:0] ALUnum2 [3:0];

  wire [3:0] regBread [1:0];
```

wire signal로 선언하는 것

: Instruction Register Write(IRwrite), register write(regwrite), Negative Zero Carry Overflow Write(NZCVwrite), Register Destination(regdst), Register Source(regsrc), Arithmetic Logic Unit Source A(ALUsrcA), Immediate Source(immsrc), Arithmetic Logic Unit Source B(ALUsrcB), Arithmetic Logic Unit Operation(ALUop), Instruction Operation(instop), Register B Destination(regBdst)

-wire 타입 상수 선언 : immediate value를 담는 32bit 크기의 imm

-wire 타입 변수 선언 : ALU의 연산으로 나온 flag, ALU 연산으로 입력받을 readA와 readB, 결과값 result, 목적지 RegisterFile(RFdst), 소스 RegisterFile(RFsrc), 2개의 ALUnum1, 3개의 ALUnum2

-2개의 Register B를 읽어오는 wire 선언

<registers>

```
//registers
assign be = 4'b1111;
wire[31:0] A,B,instructions,mdr,ALUout;
reg n,v;
wire z,c;
```

-be에 4bit 2진수 1111 연속 할당

-wire 타입 레지스터 A, B, instruction, memory data register(mdr), Arithmetic Logic Unit output(ALUout) 선언

-reg 타입 레지스터 선언 : negative flag(n), overflow flag(v)

-wire 타입 레지스터 선언 : zero flag(z), carry flag(c)

<registers>

```
register MDR(.regin (readdata), .write ('b1), .clk (clk), .reset (reset), .regout (mdr));
register ALUoutRegister(.regin (ALUresult), .write ('b1), .clk (clk), .reset (reset), .regout (ALUout));
register A_Register(.regin (readA), .write ('b1), .clk (clk), .reset (reset), .regout (A));
register B_Register(.regin (readB), .write ('b1), .clk (clk), .reset (reset), .regout (B));
register InstructionRegister(.regin (inst), .write (IRwrite), .clk (clk), .reset (reset), .regout (instructions));
//NZCVregister NZCV(.regin (ALUflags), .write(NZCVwrite), .clk (clk), .reset (reset), .regout({n,z,c,v}));
register_1bit Z(.regin (ALUflags[2]), .reset (reset), .clk (clk), .write (NZCVwrite), .regout (z));
register_1bit C(.regin (ALUflags[1]), .reset (reset), .clk (clk), .write (NZCVwrite), .regout (c));
```

레지스터 instance의 인수들

1. regin: 레지스터에 쓰여질 값을 나타내는 입력 신호
2. write: 레지스터의 쓰기 동작을 제어하는 신호
3. clk: 레지스터의 동작을 동기화하기 위해 사용되는 클락 신호
4. reset: 레지스터를 초기 상태로 되돌리기 위해 사용되는 리셋 신호
5. regout: 레지스터에서 읽어온 값을 저장하는 출력 신호

MDR

: readdata에서 입력 받고 b1에서 wirte 실행유무를 확인 가능하다. 또한 clk를 통해 클럭 신호를 확인 가능하며, reset으로 초기 상태로 되돌리기가 가능하다. 마지막으로 mdr에서 읽어온 값들이 저장되는 신호이다. A_Register B_register, InxtructionRegister, 1bit z, 1bit c들은 같은 방법으로 인스턴스화 하게 된다.

<assign>

```
//mux
assign RFdst[0]=instructions[15:12];
assign RFdst[1]=4'b1111;
assign RFdst[2]=4'b1110;

assign RFsrc[0]=mdr;
assign RFsrc[1]=ALUout;
assign RFsrc[2]=ALUresult;
assign RFsrc[3]=B;

assign regBread[0]=instructions[3:0];
assign regBread[1]=instructions[15:12];

assign ALUnum1[0]=pc;
assign ALUnum1[1]=A;
assign ALUnum1[2]='h00000000;

assign ALUnum2[0]='h00000004;
assign ALUnum2[1]='h00000008;
assign ALUnum2[2]=imm;
assign ALUnum2[3]=B<<instructions[11:7];

assign writedata=B;
assign memaddr=ALUout;
```

레지스터에 mux로 묶여 있는 변수들 할당

RFdst(목적 레지스터) : 각각 instruction 중 Rd에 해당하는 15~12비트의 4비트, 15, 14 할당

RFsrc(소스 레지스터)에는 mdr, ALUout, ALUresult, B register 할당

regBread(B 레지스터 읽기 소스) : instructions[3:0]와 insructions[15:12] 할당

ALUnum1 : 각각 pc레지스터, A 레지스터, 0 할당

ALUnum2 : 각각 4, 8, immediate value, B레지스터의 값에 instructions[11:7]만큼 시프트연산한 값 할당

→ 이렇게 할당된 값들은 mux에 의해 한가지씩 선택되어 신호를 출력하게 된다.

<signalunit SignalControl>

```
signalunit SignalControl(
    .clk (clk),
    .reset (reset),
    .flags (instructions[31:20]),
    .zero (z),
    .Mwrite (memwrite),
    .IRwrite (IRwrite),
    .Mread (memread),
    .regwrite (regwrite),
    .regdst (regdst),
    .regsrc (regsrc),
    .ALUsrcA (ALUsrcA),
    .ALUsrcB (ALUsrcB),
    .ALUop (instop),
    .NZCVwrite (NZCVwrite),
    .immsrc (immsrc),
    .regbdst (regBdst));
```

-signalunit을 SignalControl으로 인스턴스화

입력 signal

clk: 클럭 신호

reset: 리셋 신호

instructions: 명령어 flag bit

출력 signal

z: zero flag

memwrite: 메모리 write signal

regwrite: 레지스터 write signal

regdst: 레지스터 destination signal

regsrc: 레지스터 source signal

ALUsrcA: ALU A source signal

ALUsrcB: ALU B source signal

instop: 명령어 operation signal

NZCVwrite: NZCV register write signal

immsrc: immediate source signal

regBdst: 레지스터 B destination signal

<registerfile RegisterFile>

```
registerfile RegisterFile(
    .reg1 (instructions[19:16]),
    .reg2 (regBread[regBdst]),
    .regdst (RFdst[regdst]),
    .regsrc (RFsrc[regsrc]),
    .clk (clk),
    .reset (reset),
    .we (regwrite),
    .out1 (readA),
    .out2 (readB),
    .pc (pc));

signextmux Immidiate(instructions[23:0],immsrc,imm);
```

-registerfile을 사용하여 RegisterFile을 정의하고 인스턴스화

-각 포트는 이후의 괄호 부분의 값으로 할당된다.

-마지막 줄은 signextmux라는 하나의 모듈로 인스턴스화를 하는 예시이다. instructions[23:0]의 24 비트 신호를 가지고, immsrc와 imm으로 각각 제어 신호와 결과값을 나타낸다.

<ALUopdecoder AUopDecoder>

```
ALUopdecoder ALUopDecoder(
    .instop (instop),
    .aluop (ALUop));
```

-ALUopdecoder을 사용하여 ALUopDecoder을 정의하고 인스턴스화

-instop과 ALUop 신호를 입력으로 받아서 적절한 ALU 작업을 수행하기 위한 ALU 제어 신호를 생성한다.

<ALU32bit ALU>

```
ALU32bit ALU(
    .inpa (ALUnum1[ALUsrcA]),
    .inpb (ALUnum2[ALUsrcB]),
    .cin (c),
    .aluop (ALUop),
    .result (ALUresult),
    .negative (ALUflags[3]),
    .zero (ALUflags[2]),
    .cout (ALUflags[1]),
    .overflow (ALUflags[0]));

endmodule
```

-ALU32bit을 사용하여 ALU을 정의하고 인스턴스화

-입력으로 ALUnum1[ALUsrcA]와 ALUnum2[ALUsrcB]를 받아서 c 신호를 캐리 입력으로 사용하고, ALUop을 통해 어떤 연산을 수행할지 결정한다. 그 결과는 ALUresult에 저장되며 ALUflags의 여러 비트를 통해 부호, 제로, 캐리 아웃, 오버플로우 등의 상태를 표시한다.

#2 Analyze Signal unit

<module signalcontrol>

```
module signalcontrol(
    input[11:0] flags,
    input zero,
    output reg [2:0] total,

    output reg [19:0] s2,
    output reg [19:0] s3,
    output reg [19:0] s4);

 always @ (*) begin
  if((flags[11]&flags[10]&flags[9])||(flags[8]^zero)) begin
   if(flags[7]) begin //B, BL
    if(~flags[4]) begin //B
    s2=20'b00010110001001000100;
    s3=20'bxxxxxxxxxxxxxxxxxxxx;
    s4=20'bxxxxxxxxxxxxxxxxxxxx;
    total=2;
    end
   else begin //BL
    s2=20'b00011001001001000100;
    s3=20'b00010101xxxxxxxx0xxx;
    s4=20'bxxxxxxxxxxxxxxxxxxxx;
    total=3;
   end
  end
  else if(flags[6]) begin //LDR, STR
   s2={10'b0001010101,(flags[5]==1 ? 2'b11 : 2'b10),(flags[3]==1 ? 4'b0100 : 4'b0010), 3'b001, (flags[0]==1 ? 1'b0 : 1'b1)};

   if(~flags[0]) begin //STR
    s3=20'b1000xxxxxxxxxxx0xxx;
    s4=20'bxxxxxxxxxxxxxxxxxxxx;
    total=3;
   end
   else begin //LDR
    s3=20'b0010xxxxxxxxxxx0xxx;
    s4=20'b00010000xxxxxxxx0xxx;
    total=4;
   end
  end
```

```
   else //Else
    case(flags[4:1])/*
       0 : //AND
       1 : //EOR
       2 : //SUB
       4 : //ADD
       5 : //ADC
       6 : //SBC
       12 : //ORR*/
     10 : begin //CMP
     s2={10'b0001010101, (flags[5]==1 ? 2'b10 : 2'b11), 8'b00101000};
     s3=20'bxxxxxxxxxxxxxxxxxxxx;
     s4=20'bxxxxxxxxxxxxxxxxxxxx;
     total=2;
     end
     13 : begin //MOV
     s2={10'b0001010110, (flags[5]==1 ? 2'b10 : 2'b11), 4'b0100 ,flags[0], 3'b000};
     s3=20'b00010001xxxxxxxx0xxx;
     s4=20'bxxxxxxxxxxxxxxxxxxxx;
     total=3;
     end
     default : begin //ALU
      s2={10'b0001010101, (flags[5]==1 ? 2'b10 : 2'b11), flags[4:0], 3'b000};
      s3=20'b00010001xxxxxxxx0xxx;
      s4=20'bxxxxxxxxxxxxxxxxxxxx;
      total=3;
     end
    endcase
   end
   else begin //Recovery
    s2=20'b00010101xxxxxxxx0xxx;
    s3=20'bxxxxxxxxxxxxxxxxxxxx;
    s4=20'bxxxxxxxxxxxxxxxxxxxx;
    total=2;
   end
  end
endmodule
```

모듈 역할

: 입력된 flags 신호와 zero 신호를 기반으로 다양한 상태 (s2 , s3 , s4) 및 제어 신호를 생성한다. always @ (*) 블록을 사용하여 조건부 논리를 구현하고, if-else 문과 case 문을 사용하여 조건에 따라 상태를 설정한다.

입력 신호

flags : 12비트 신호

zero : 1비트 신호

출력 신호

total : 사용된 output의 수(3비트)

상태 s2 , s3 , s4 : control signal을 나타낸다. (각각 20비트)

내부 동작

input flags[11:8]가 111x 이거나 xxx1일 때 동작 수행

→flags[7]가 1일 때 : flags[4]가 0 : B ----------flags[4]는 Link bit
flags[4]가 1 : BL

→flags[7]가 0일 때 : flags[6]가 0일 때 : flags[4:1]가 0000 : AND -------flags[4:1]는 OPCode
flags[4:1]가 0001 : EOR
flags[4:1]가 0010 : SUB
flags[4:1]가 0100 : ADD
flags[4:1]가 0101 : ADC
flags[4:1]가 0110 : SBC
flags[4:1]가 1100 : ORR
flags[6]가 1일 때 : flags[0]가 0 : STR ---------- flags[0]는 Load/Store bit
flags[0]가 1 : LDR

<module oneAdder>

```
module oneAdder(
    input clk,
    input reset,
    input [2:0] current,
    output reg[2:0] regout);

    wire last;

    assign last=current==regout;

    always @ (posedge clk,posedge reset)
    if(reset)
        regout<='b000;
    else if(last)
        regout<='b000;
    else
        regout<=regout+'b001;
endmodule
```

모듈역할

: 카운터의 현재 값과 Total 값을 비교하여 마지막 Step에 도달했을 경우 0으로 초기화하고, 다른 경우에는 1씩 증가시키는 역할을 수행한다.

입력신호

clk : 클럭 reg

reset : 리셋 reg

current : Total 카운터를 나타내는 3비트 reg

출력신호

regout : 현재 카운터를 나타내는 3비트 출력 reg

내부신호

last : current와 regout이 같은 지를 검사하여, 카운터가 마지막에 도달했는지 아닌지를 나타내는 wire

내부동작

always @ (posedge clk,posedge reset)

-동작조건: clk 과 reset 의 상승 edge 발생시 동작 ctrlSig.v 2

-동작

reset == 1 : regout 은 'b000으로 초기화

reset == 0, last ==1 : regout 은 'b000으로 초기화

reset == 0, last ==0 : regout 은 regout 에 'b001을 더한 값으로 갱신

<module signalunit>

```
module signalunit
    (input clk,
     input reset,
     input[11:0] flags,
     input zero,
     output Mwrite,
     output IRwrite,
     output Mread,
     output regwrite,
     output[1:0] regdst,
     output[1:0] regsrc,
     output[1:0] ALUsrcA,
     output[1:0] ALUsrcB,
     output[3:0] ALUop,
     output NZCVwrite,
     output [1:0] immsrc,
     output regbdst);

  wire [19:0] s [4:0];

  assign s[0] = 20'b01110110000101000xxx;
  assign s[1] = 20'b0000xxxx000000100xxx;

  wire [2:0] total;
  wire [2:0] step;

  oneAdder Step(.clk (clk), .reset (reset), .current (total), .regout (step));

  signalcontrol bringSignal(
    .flags (flags),
    .zero (zero),
    .total (total),
    .s2 (s[2]),
    .s3 (s[3]),
    .s4 (s[4]));

  assign Mwrite=s[step][19];
  assign IRwrite=s[step][18];
  assign Mread=s[step][17];
  assign regwrite=s[step][16];
  assign regdst=s[step][15:14];
  assign regsrc=s[step][13:12];
  assign ALUsrcA=s[step][11:10];
  assign ALUsrcB=s[step][9:8];
  assign ALUop=s[step][7:4];
  assign NZCVwrite=s[step][3];
  assign immsrc=s[step][2:1];
  assign regbdst=s[step][0];

endmodule
```

모듈역할

Instruction flag를 받아 해당하는 instruction에 맞는 control signal들을 반환하는 역할을 수행한다.

→ (12bit instruction) x (5 step) x (20bit Flag)

입력신호

clk : 클럭 reg

reset : 리셋 reg

flags : 12비트 Instruction flag를 나타내는 reg (ALU, Branch, Load, Store 등)

zero : 제로 reg

출력신호

총 20비트의 control signal을 각각 reg에 나누어 반환한다.

Mwrite : 메모리에 쓰기 동작

IRwrite : IR 레지스터에 쓰기 동작

Mread : 메모리에서 읽기 동작
regwrite : 레지스터에 쓰기 동작
regdst : 레지스터 목적지
regsrc : 레지스터 소스
ALUsrcA : ALU의 A 소스
ALUsrcB : ALU의 B 소스
ALUop : ALU의 연산 종류
NZCVwrite : NZCV 레지스터에 쓰기 동작
ctrlSig.v 4 immsrc : immediate 값 소스
regbdst : 레지스터 B 목적지

내부신호

s : 최대 5개의 각 step별 20비트 Control Signal을 나타내는 wire → 0,1 번째 step에 해당하는 control signal인 s[0]과 s[1] 은 내부 모듈에 의해 hard coding되어있다.
total : Instruction별 최대 step을 나타내는 wire
step : 현재 step을 나타내는 wire

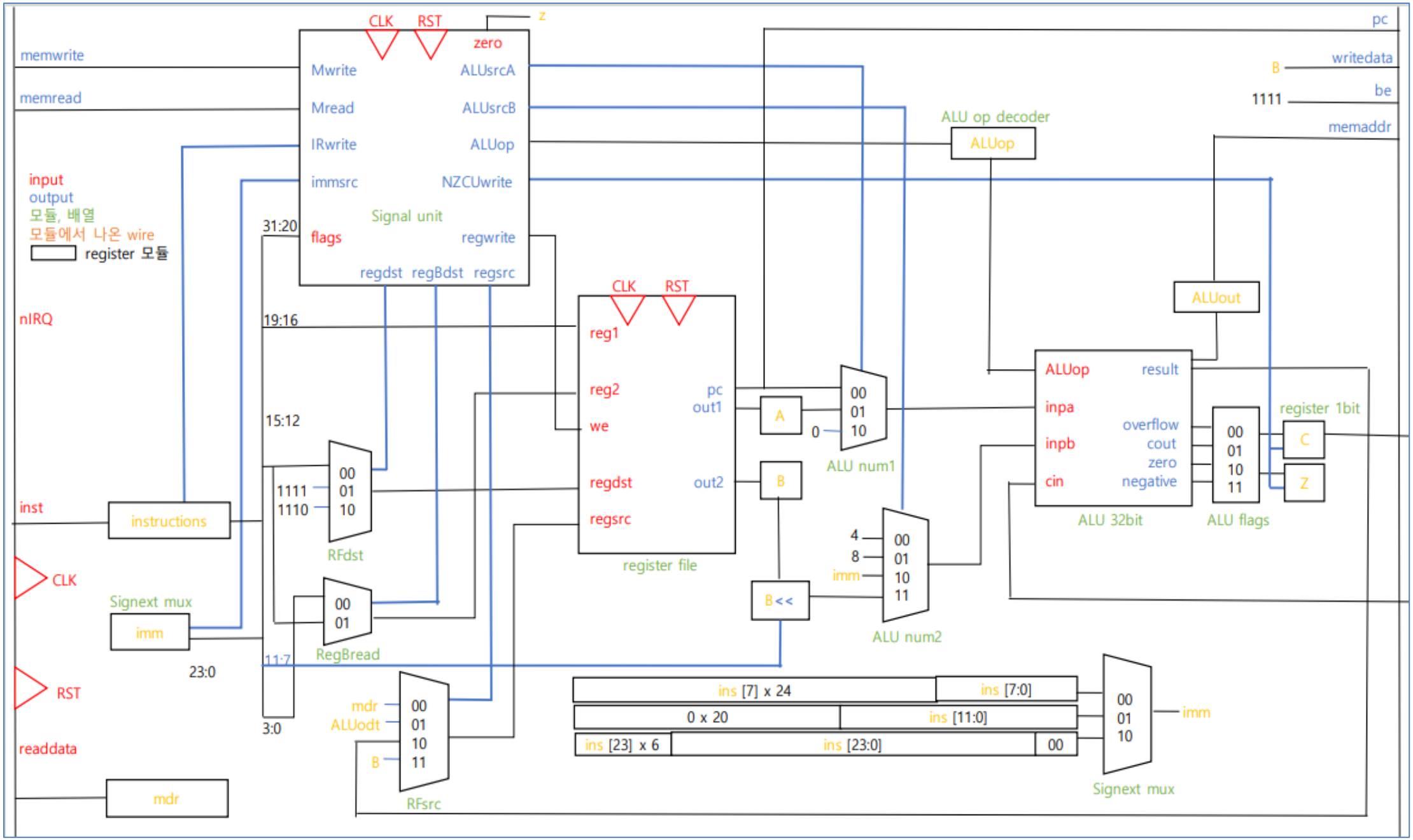
내부동작

step : 앞서 기술한 oneAdder 모듈을 이용해 step을 증가시키거나, total step에 도달한 경우 초기화한다.
bringSignal : 앞서 기술한 signalcontrol 모듈을 이용해 flags에 해당하는 instruction에 대한 정보를 가져온다

→ total instruction의 최대 step을 가져온다.

→ s2, s3, s4 : instruction에 대해 2,3,4번째 스텝에 해당하는 control signal을 가져온다
assign : instruction과 step에 해당하는 flag를 출력하되, 각각의 reg에 assign하여 출력한다

#3 Draw block diagram of multi-cycle ARM processor



#4 Explain How the Test program works on the given processor

- Instruction [공통]

▼ `Decode`

- `signextmux`

- `ALUop decoder`

ctrlSig	0010	0100
ALUop	000	001
mean	SUB	ADD

- `signalunit MUX`

	regdst (w 목적지)	regsrc (w source)	regbdst (r out2)
00	`instruction`[15:12]	`mdr`	`instruction`[15:12]
01	1111(`pc`)	`ALUout`	`instruction`[3:0]
10	1110(reg[14]	`result`	
11		`B`	

▼ Step [0]

- `signalunit`

IRwrite	Mwrite/Mread	regwrite/regdst/regsrc	regbdst	ALUsrcA/ALUsrcB	ALUop	NZCVwrite/i
1	0/1	1/01/10	x	00/01	0100	x

- `register`

	Instructions	A	B	ALUout	imm	mdr
value	`inst`	x	x	x	x	`readdata`
change	o					o

- `registerfile`

reg	reg2	we	regdst	regsrc
x	x	1	1111	10
		pc ← pc+8	`pc`	`result`(pc+8)

out1	out2	pc
x	x	`pc+8`

- `ALU32bit`

aluop	inpa	inpb	cin
001	00	01	x
ADD	`pc`	8	

result	NZCU
`pc+8`	x

▼ Step [1]

- `signalunit`

IRwrite	Mwrite/Mread	regwrite/regdst/regsrc	regbdst	ALUsrcA/ALUsrcB	ALUop	NZCVwrite/i
0	00	x	x	00/00	0010	x

- `register`

	Instructions	A	B	ALUout	imm	mdr
value	`inst`	reg[ins[19:16]]	x	`pc+8`	x	`readdata`
change		o		o		

- `registerfile`

reg1	reg2	we	regdst	regsrc
ins[19:16]	x	x	x	x

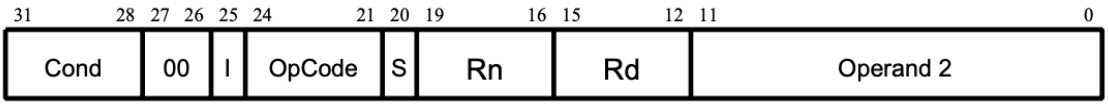
out1	out2	pc
reg[ins[19:16]]	x	`pc+8`

- `ALU32bit`

aluop	inpa	inpb	cin
000	00	00	x
SUB	`pc`	4	

result	NZCU
`pc-4`	x

- MOV r1, #0
  - ▼ `instructions`



    - [31:20] `flags` == 1110 00 1 1101 0
    - [19:16] `Rn` == x
    - [15:12] `Rd` == 0001
    - [11:0] `op2` == #0
  - ▼ Step [2]
    - `signalunit`

IRwrite	Mwrite/Mread	regwrite/regdst/regsrc	regbdst	ALUsrcA/ALUsrcB	ALUop	NZCVw
0	00	1/01/01	0	10/10	0100	x/00

    - `register`

	Instructions	A	B	ALUout	imm	mdr
value	`inst`	reg[Rn] (의미없음)	x	`pc-4`	ins[7:0]	`readdata`
change				o	o	

    - `registerfile`

reg	reg2	we	regdst	regsrc
x	x	1	15	01
			`pc`	

		pc ← pc-4	`ALUout` (pc-4)

out1	out2	pc
x	x	pc-4

- `ALU32bit`

aluop	inpa	inpb	cin
001	10	10	x
ADD	0	`imm` (ins[7:0])	

result	NZCU
ins[7:0] = 0	x

▼ Step [3]

- `signalunit`

IRwrite	Mwrite/Mread	regwrite/regdst/regsrc	regbdst	ALUsrcA/ALUsrcB	ALUop	NZCVw
0	00	1/00/01	x	x	x	x

- `register`

	Instructions	A	B	ALUout	imm	mdr
value	`inst`	x	x	ins[7:0] = 0	ins[7:0]	`readdata`
change				o		

- `registerfile`

reg	reg2	we	regdst	regsrc
x	x	1	ins[15:12]	01
		reg[1] ← `imm` (0)	`Rd` ($1)	`ALUout` (0)

out1	out2	pc
x	x	x

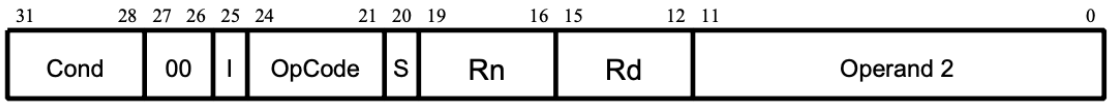
- `ALU32bit`

aluop	inpa	inpb	cin
x	x	x	x

result	NZCU
x	x

- ADD r1, r1, #1

▼ `instructions`



- [31:20] `flags` == 1110 00 1 0100 0
- [19:16] `Rn` == 0001
- [15:12] `Rd` == 0001
- [11:0] `op2` == #1

▼ Step [2]

- `signalunit`

IRwrite	Mwrite/Mread	regwrite/regdst/regsrc	regbdst	ALUsrcA/ALUsrcB	ALUop	NZCVw
0	00	1/01/01	0	01/10	0100	0/00

- `register`

	Instructions	A	B	ALUout	imm	mdr
value	`inst`	reg[1]	x	`pc-4`	#1	`readdata`
change				o	o	

- `registerfile`

reg	reg2	we	regdst	regsrc
x	x	1	15	01
		pc ← pc-4	`pc`	`ALUout`(pc-4)

out1	out2	pc
x	x	pc-4

- `ALU32bit`

aluop	inpa	inpb	cin
001	01	10	x
ADD	reg[1]	`imm`(#1)	

result	NZCU
reg[1] + 1	

▼ Step [3]

- `signalunit`

IRwrite	Mwrite/Mread	regwrite/regdst/regsrc	regbdst	ALUsrcA/ALUsrcB	ALUop	NZCVw
0	00	1/00/01	x	x	x	0/x

- `register`

	Instructions	A	B	ALUout	imm	mdr
value	`inst`	reg[1]	x	reg[1] + 1	#1	`readdata`
change				o		

- `registerfile`

reg	reg2	we	regdst	regsrc
x	x	1	ins[15:12]	01
		reg[1] ← += 1)	`Rd`($1)	`ALUout` (reg[1]+1)

out1	out2	pc
x	x	x

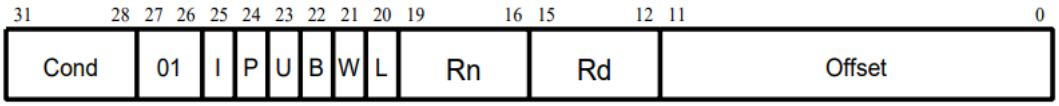
- `ALU32bit`

aluop	inpa	inpb	cin
x	x	x	x

result	NZCU
x	x

- LDR r1, [r0, #0xC]

▼ `instructions`



- [31:20] `flags` == 1110 01 0 1 1 0 0 1
- [19:16] `Rn` == 0000
- [15:12] `Rd` == 0001
- [11:0] `op2` == 0000 0000 1100

▼ Step [2]

- `signalunit`

IRwrite	Mwrite/Mread	regwrite/regdst/regsrc	regbdst	ALUsrcA/ALUsrcB	ALUop	NZCVw
0	00	1/01/01	1	01/10	0100	0/01

- `register`

	Instructions	A	B	ALUout	imm	mdr
value	`inst`	reg[0]	x	`pc-4`	12	`readdata`
change				o	o	

- `registerfile`

reg	reg2	we	regdst	regsrc
x	ins[15:12]	1	15	01
	Rd(0001)	pc ← pc-4	`pc`	`ALUout`(pc-4)

out1	out2	pc
x	Rd(0001)	pc-4

- `ALU32bit`

aluop	inpa	inpb	cin
001	ALUnum1[01]	imm	x
ADD	`A`(reg[0])	12	

result	NZCU
reg[0]+12	x

▼ Step [3]

- `Module`
  - output
    - `memread` = 1
    - `memaddr` = `ALUout` = reg[0] + 12
  - input
    - `readdata`
- `signalunit`

IRwrite	Mwrite/Mread	regwrite/regdst/regsrc	regbdst	ALUsrcA/ALUsrcB	ALUop	NZCVw
0	01	x	x	x	x	0/xx

- `register`

	Instructions	A	B	ALUout	imm	mdr
value	`inst`	x	Rd(0001)	reg[0]+12	12	`readdata`
change						o

- `registerfile`

reg	reg2	we	regdst	regsrc
x	x	x	x	x

out1	out2	pc
x	x	x

- `ALU32bit`

aluop	inpa	inpb	cin
x	x	x	x

result	NZCU
x	x

▼ Step [4]

- `signalunit`

IRwrite	Mwrite/Mread	regwrite/regdst/regsrc	regbdst	ALUsrcA/ALUsrcB	ALUop	NZCVw
0	00	1/00/00	x	x	x	0/x

- `register`

	Instructions	A	B	ALUout	imm	mdr
value	`inst`	x	x	x	12	`readdata`
change						

- `registerfile`

reg	reg2	we	regdst	regsrc
x	x	1	ins[15:12]	00
			Rd(0001)	mdr(`readdata`)

out1	out2	pc
x	x	x

- `ALU32bit`

aluop	inpa	inpb	cin
x	x	x	x

result	NZCU
x	x

- BEQ reset

▼ Step [2]

- `signalunit`

IRwrite	Mwrite/Mread	regwrite/regdst/regsrc	regbdst	ALUsrcA/ALUsrcB	ALUop	NZCVwrite/i
1	00	1/01/10	0	00/10	0100	0/10

- `register`

	Instructions	A	B	ALUout	imm	mdr
value	`inst`	reg[ins[19:16]]	x	`pc-4`	inst[23] * 6 + inst[23:0] + '00'	`readdata`
change						

- `registerfile`

reg	reg2	we	regdst	regsrc
x	ins[3:0]	1	15	ALUresult (pc + imm)

out1	out2	pc
x	registers[reg2]	x

- `ALU32bit`

aluop	inpa	inpb	cin
x	x	x	x

result	NZCU
pc + imm	x

--> module signalcontrol에서
zero == 1 (set) : branch 구문으로 넘어가 reset에 해당하는 label 구간으로 분기한다.
zero == 0 (clear) : recovery로 넘어가 pc + 4로 진행하며 분기하지 않고 바로 다음 명령어로 넘어간다.