마이크로프로세서 설계 무작정 따라하기 (2)

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

5.1. 데이터패쓰의 설계

이번 회에서는 SimpleCore 의 데이터패쓰의 주요 기능블록을 Verilog 를 사용하여 설계한다. 모든 설계는 합성 가능한 RTL(Register Transfer Level) 코드를 사용하여 기술한다. RTL 기술은 전체 하드웨어 구조가 정해지고 각 블록의 기능이 결정된 단계에서 실행하며, 하드웨어의 기본적인 동작과 구조를 기술한다. 구체적인 회로는 합성 툴과 합성에 사용되는 입력(최소 동작속도, 회로의 최대크기, 등)에 따라 달라지게 된다.

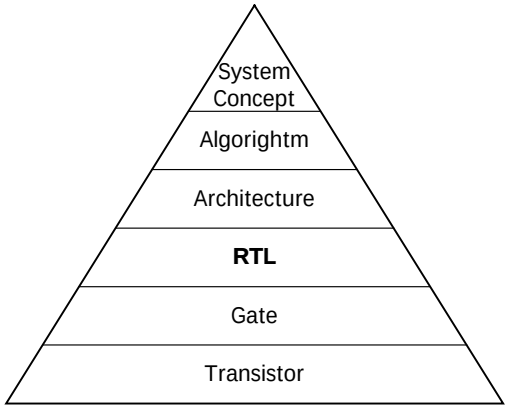

그림 1. Abstraction pyramid

SimpleCore 의 데이터패쓰는 ALU, 쉬프터, 곱셈기 그리고 레지스터파일로 구성되어있다.

가) ALU

ALU(Arithmetic Logic Unit)는 덧셈, 뺄셈과 간단한 논리 연산(AND, OR)을 수행한다. 그림 2 과 같이 16 비트 덧셈기를 이용하여 덧셈과 뺄셈을 모두 수행하는 구조를 가정하였으며, 이에 따라 ALU 의 제어신호(aluCtl)의 값을 결정하였다(표 1).

표 1. ALU 동작

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

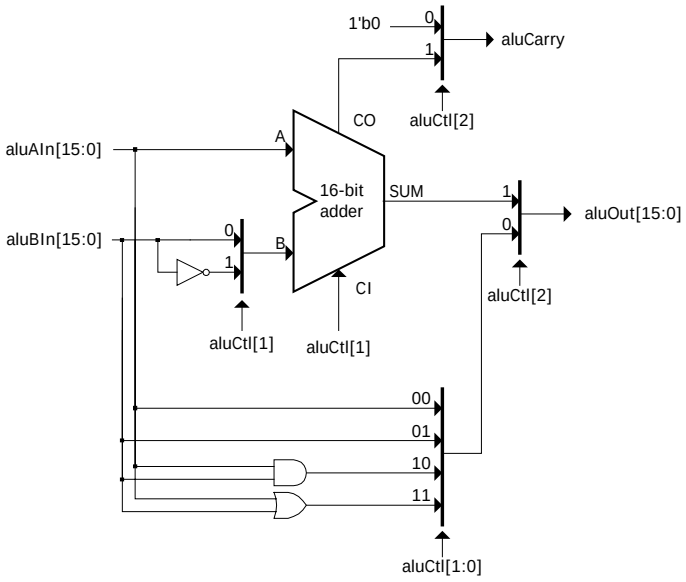

그림 2. ALU 의 구조

ALU 의 RTL 코드

```
module  alu(
       aluAIn,
       aluBIn,
       aluCtl,
       aluOut,
       aluCarry
);

input   [15:0]  aluAIn;   // alu input A
input   [15:0]  aluBIn;   // alu input B
input   [ 2:0]  aluCtl;   // alu control input
output  [15:0]  aluOut;   // alu output
output         aluCarry; // alu carry output

reg     [15:0]  aluOut;
reg            aluCarry;

always @ (aluAIn or aluBIn or aluCtl)
begin
       casex(aluCtl)
              3'b000: // MOVA
                     {aluCarry, aluOut} = {1'b0, aluAIn};
              3'b001: // MOVB
                     {aluCarry, aluOut} = {1'b0, aluBIn};
              3'b010: // AND
                     {aluCarry, aluOut} = {1'b0, aluAIn & aluBIn};
              3'b011: // OR
                     {aluCarry, aluOut} = {1'b0, aluAIn | aluBIn};
              3'b10?: // ADD
                     {aluCarry, aluOut} = aluAIn + aluBIn;
              3'b11?: // SUB
                     {aluCarry, aluOut} = aluAIn - aluBIn;
       endcase
end

endmodule
```

나) 쉬프터

한 싸이클에 임의의 값만큼 쉬프트연산을 하기 위하여 배럴 쉬프터(barrel shifter)구조를 많이 사용한다. 16 비트 배럴 쉬프터의 기본 개념은 그림 3 와 같다. 그림의 사각형은 2:1 MUX 를 나타내고 있다. 각 단에서 1,2,4,8 비트 쉬프트 연산이 이루어지며 shift amount(shitAmt)신호에 따라 MUX 로 선택하는 구조이다.

그림 3 는 왼쪽으로 5 비트 쉬프트하는 경우를 나타내고 있다.

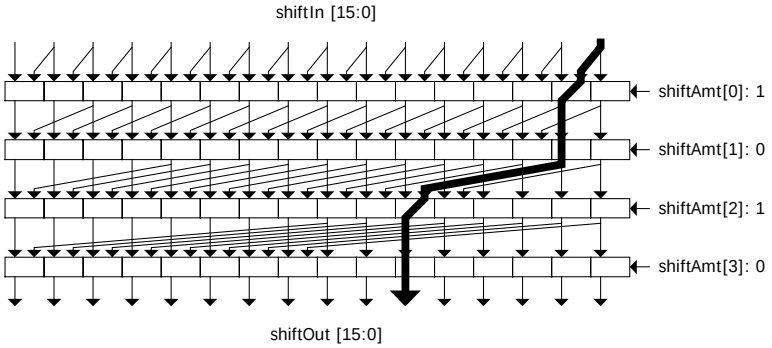

그림 3. 배럴 쉬프터의 구조 (왼쪽으로 5 비트 쉬프트하는 예제)

Verilog RTL 로 그림 3 의 쉬프터 구조를 기술하는 방법은 다음과 같다.

```
input   [15:0]  shiftIn;
input   [ 3:0]  shiftAmt;
wire    [15:0]  stage1, stage2, stage3, stage4;
output  [15:0]  shiftOut;

assign stage1 = shiftAmt[0] ? shiftIn[15:0] : {shiftIn[14:0], 1'b0};
assign stage2 = shiftAmt[1] ? stage1 [15:0] : {stage1 [13:0], 2'b0};
assign stage3 = shiftAmt[2] ? stage2 [15:0] : {stage2 [11:0], 4'b0};
assign stage4 = shiftAmt[3] ? stage3 [15:0] : {stage3 [ 7:0], 8'b0};

assign shiftOut = stage4;
```

표 2. 쉬프터의 동작

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

실제 쉬프터의 동작은 표 2 와 같은 동작을 수행한다. 따라서 앞서 설명한 내용보다 복잡한 구조를 갖게 된다. 또한 실제 구현방법도 다양하다. 본 강좌에서는 이해가 쉬운 단순하고 합성 가능한 코드를 사용하였다.

쉬프터의 RTL 코드

```
module  shifter(
        shiftIn,
        shiftAmt,
        shiftCtl,
        shiftOut
);

input   [15:0]  shiftIn;  // shifter input
input   [ 3:0]  shiftAmt; // shift amount
input   [ 2:0]  shiftCtl; // control input {left/right, arith/logic, rotate}
output  [15:0]  shiftOut; // shifter output

reg     [15:0]  shiftOut;

always @ (shiftIn or shiftAmt or shiftCtl)
begin
        casex(shiftCtl)
                3'b0?? : // LSL (logical shift left)
                        shiftOut = shiftIn << shiftAmt;
```

```
            3'b100 : // LSR (logical shift right)
                  shiftOut = shiftIn >> shiftAmt;
            3'b101 : // ASR (arithmetic shift right)
                  shiftOut = ({16{shiftIn[15]}}<<(16-shiftAmt)) | (shiftIn >> shiftAmt);
            3'b11? : // ROR (rotate right)
                  shiftOut = (shiftIn >> shiftAmt) | (shiftIn << (16-shiftAmt));
      endcase
end

endmodule
```

다) 곱셈기

곱셈기는 많은 면적을 차지하고 다른 기능블록에 비해 동작속도가 느리기 때문에 다양한 방법을 사용하여 성능을 높인다.(예, Modified Booth 방식). 그러나, 구체적인 곱셈기의 설계방법은 매우 복잡하므로, 본 강좌는 합성 툴의 기능에 의존하여 곱셈기를 설계하고자 한다. 합성 툴을 이용할 경우, 단순히 *(곱셈 연산부호)를 사용하는 것만으로도 곱셈기를 설계할 수 있다. 지난 강좌에 설명한 것과 같이 최적화 설계를 위해서는 데이터패쓰합성기(datapath compiler), 모듈 생성기(module generator), 등을 이용해야 한다.

곱셈기의 RTL 코드

```
module  mul(
      mulAIn,
      mulBIn,
      mulOut
);

input   [15:0]  mulAIn; // multiplier input A
input   [15:0]  mulBIn; // multiplier input B
output  [15:0]  mulOut; // multiplier output

assign  mulOut = mulAIn * mulBIn;

endmodule
```

라) 레지스터 파일

SimpleCore 의 레지스터 파일은 그림 4 와 같이 두 개의 읽기 포트와 한 개의 쓰기 포트를 갖고있다. 명령어로부터 읽기 제어신호 (rdAIdx, rdBIdx, rdAOEn, rdBOEn)와 쓰기 제어신호(wbIdx, wbEn)를 생성하여 레지스터의 동작을 제어한다. 이를 위하여 4 비트의 쓰기주소(wbIdx)를 16 개의 선택신호로 변환하는 부분과 16 개의 16 비트 레지스터 그리고 16 개의 레지스터값을 읽기주소(rdAIdx, rdBIdx)에 따라 선택하는 부분으로 구성되어있다.

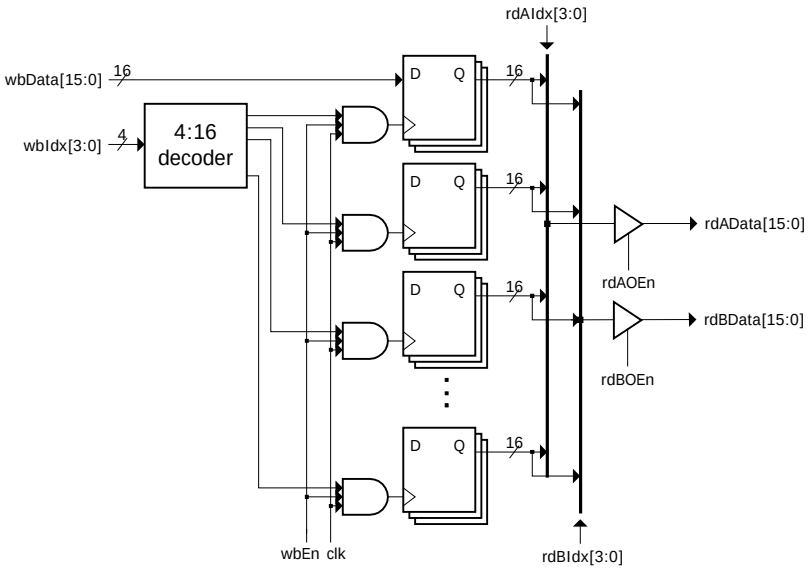

그림 4. 레지스터 파일

레지스터 파일의 RTL 코드

```
module  gpr (
       clk,
       wbEn,
       wbData,
       rdData
);

input         clk;
input         wbEn;
input  [15:0]  wbData;
output [15:0]  rdData;

reg    [15:0]  rdData;

always  @(posedge clk)
       if(wbEn)
             rdData= wbData;

endmodule


module  regSel (
       rdIdx,
       gpr0 , gpr1 , gpr2 , gpr3 , gpr4 , gpr5 , gpr6 , gpr7,
       gpr8 , gpr9 , gpr10, gpr11, gpr12, gpr13, gpr14, gpr15,
       rdData
);

input  [ 3:0]  rdIdx;
input  [15:0]  gpr0 , gpr1 , gpr2 , gpr3 , gpr4 , gpr5 , gpr6 , gpr7;
input  [15:0]  gpr8 , gpr9 , gpr10, gpr11, gpr12, gpr13, gpr14, gpr15;
output [15:0]  rdData;

reg    [15:0]  rdData;

always @ (rdIdx or gpr0  or gpr1  or gpr2  or gpr3  or gpr4  or gpr5  or gpr6  or gpr7  or
       gpr8  or gpr9  or gpr10 or gpr11 or gpr12 or gpr13 or gpr14 or gpr15)
begin
       casex(rdIdx)
             4'd0 :  rdData = gpr0 ;
             4'd1 :  rdData = gpr1 ;
             4'd2 :  rdData = gpr2 ;
             4'd3 :  rdData = gpr3 ;
             4'd4 :  rdData = gpr4 ;
             4'd5 :  rdData = gpr5 ;
             4'd6 :  rdData = gpr6 ;
             4'd7 :  rdData = gpr7 ;
             4'd8 :  rdData = gpr8 ;
             4'd9 :  rdData = gpr9 ;
             4'd10:  rdData = gpr10;
             4'd11:  rdData = gpr11;
             4'd12:  rdData = gpr12;
```

```
            4'd13:  rdData = gpr13;
            4'd14:  rdData = gpr14;
            4'd15:  rdData = gpr15;
      endcase
end

endmodule


module  regFile (
      clk,
      rdAIdx,
      rdBIdx,
      rdAOEn,
      rdBOEn,
      wbIdx,
      wbData,
      rdAData,
      rdBData
);

input         clk;
input   [ 3:0]  rdAIdx;  // read index A
input   [ 3:0]  rdBIdx;  // read index B
input         rdAOEn;  // read A output enable
input         rdBOEn;  // read A output enable
input   [ 3:0]  wbIdx;   // writeback index
input   [15:0]  wbData;   // writeback data
output  [15:0]  rdAData;  // read data A
output  [15:0]  rdBData;  // read data B

wire    [15:0]  gpr0 , gpr1 , gpr2 , gpr3 , gpr4 , gpr5 , gpr6 , gpr7;
wire    [15:0]  gpr8 , gpr9 , gpr10, gpr11, gpr12, gpr13, gpr14, gpr15;
wire    [15:0]  rdADataTmp, rdBDataTmp;

gpr     Igpr0 (.clk(clk), .wbEn(wbIdx == 4'd0),  .wbData(wbData), .rdData(gpr0 ));
gpr     Igpr1 (.clk(clk), .wbEn(wbIdx == 4'd1),  .wbData(wbData), .rdData(gpr1 ));
gpr     Igpr2 (.clk(clk), .wbEn(wbIdx == 4'd2),  .wbData(wbData), .rdData(gpr2 ));
gpr     Igpr3 (.clk(clk), .wbEn(wbIdx == 4'd3),  .wbData(wbData), .rdData(gpr3 ));
gpr     Igpr4 (.clk(clk), .wbEn(wbIdx == 4'd4),  .wbData(wbData), .rdData(gpr4 ));
gpr     Igpr5 (.clk(clk), .wbEn(wbIdx == 4'd5),  .wbData(wbData), .rdData(gpr5 ));
gpr     Igpr6 (.clk(clk), .wbEn(wbIdx == 4'd6),  .wbData(wbData), .rdData(gpr6 ));
gpr     Igpr7 (.clk(clk), .wbEn(wbIdx == 4'd7),  .wbData(wbData), .rdData(gpr7 ));
gpr     Igpr8 (.clk(clk), .wbEn(wbIdx == 4'd8),  .wbData(wbData), .rdData(gpr8 ));
gpr     Igpr9 (.clk(clk), .wbEn(wbIdx == 4'd9),  .wbData(wbData), .rdData(gpr9 ));
gpr     Igpr10(.clk(clk), .wbEn(wbIdx == 4'd10), .wbData(wbData), .rdData(gpr10));
gpr     Igpr11(.clk(clk), .wbEn(wbIdx == 4'd11), .wbData(wbData), .rdData(gpr11));
gpr     Igpr12(.clk(clk), .wbEn(wbIdx == 4'd12), .wbData(wbData), .rdData(gpr12));
gpr     Igpr13(.clk(clk), .wbEn(wbIdx == 4'd13), .wbData(wbData), .rdData(gpr13));
gpr     Igpr14(.clk(clk), .wbEn(wbIdx == 4'd14), .wbData(wbData), .rdData(gpr14));
gpr     Igpr15(.clk(clk), .wbEn(wbIdx == 4'd15), .wbData(wbData), .rdData(gpr15));

regSel  regASel (
      rdAIdx,
      gpr0 , gpr1 , gpr2 , gpr3 , gpr4 , gpr5 , gpr6 , gpr7,
      gpr8 , gpr9 , gpr10, gpr11, gpr12, gpr13, gpr14, gpr15,
      rdADataTmp
);

regSel  regBSel (
      rdBIdx,
      gpr0 , gpr1 , gpr2 , gpr3 , gpr4 , gpr5 , gpr6 , gpr7,
      gpr8 , gpr9 , gpr10, gpr11, gpr12, gpr13, gpr14, gpr15,
      rdBDataTmp
);

assign  rdAData = rdAOEn ? rdADataTmp : 16'hz;
assign  rdBData = rdBOEn ? rdBDataTmp : 16'hz;

endmodule
```

다음 회에 계속됩니다. 다음 회에서는 컨트롤 회로의 설계 및 시뮬레이션 방법을 내용을 설명합니다.

본 강좌에 사용된 예제는 홈페이지(http://www.donny.co.kr/simplecore)를 통해 제공됩니다.