韓國 OA學會 論文誌
第5卷 第3號, 2000. 9.

2000-5-3-1-3

HDL을 이용한 파이프라인 프로세서의 테스트 벡터 구현에 의한 시뮬레이션

박 두 열*

Simulation on a test vector Implementation of a pipeline processor using a HDL

Doo-Youl Park*

요 약

본 연구에서는 HDL을 이용하여 16-비트의 파이프라인 프로세서를 함수적 레벨에서 기술하여 구현하고, 그 프로세서의 동작을 확인하였다. 구현된 파이프라인 프로세서를 시뮬레이션할 때 그 프로세서 내에서 실행되는 테스트 벡터를 기호로 표시된 명령어로 먼저 설정하여 규정하고, 구현된 명령어 세트를 프로그래밍하여 입력하였다.

따라서 본 연구에서 제시된 테스트 벡터를 이용한 시뮬에이션 방법은 프로세서의 동작을 쉽게 확인할 수 있었으며, 정확한 시뮬레이션을 할 수 있었고, HDL을 이용하므로써 구현시 프로세서의 동작을 문서화하는 것이 간편하였다.

Abstract

In this paper, we implemented by describing a pipeline processor using a HDL in functional level, simulated and verified it's operation. When simulating a implemented processor, We first specify assembly instruction that is performed in the processor, entered by programming using the instruction sets at the experimental framework.

Thus, the procedure that is presented in this paper can easily identify and verify the purpose for implementation and operation of a system by using test vector. Also, it was possible that exactly simulate a system. The method was comfortable that document a system operation to implement.

* 동주대학 컴퓨터정보통신계열 부교수
본 연구는 동주대학 학술연구비 지원에 의한 논문임

Ⅰ. 서 론

메모리와 같은 디지털 회로 설계에서 칩의 밀도와 복잡성이 급격히 증가함에 따라 이런 욕구들을 만족시키기 위해 모듈화된 회로들이 여러 가지 방법으로 설계되어 왔다. 과거의 설계 방법들 중에는 모듈을 하드와이어적(Hardwired)으로 직접 설계를 수행함에 따라 동작의 결과를 확인할 수 있는 방법이라고는 하드웨어적으로 동작시키는 것밖에 없었으며, 설계과정에서 발생한 오류를 검출할 때 그와 같은 설계과정을 반복해야 하는 문제로 인해 설계시간과 비용이 증가하게 되었다. 따라서 프로세서와 같은 비메모리 설계분야에서 이런 문제점들을 해결하고 개선하기 위해 설계자는 실제적인 설계를 수행할 때 설계하고자 하는 하드웨어를 분석하고 그 규격을 문서화하여 구현하는 HDL (Hardware Description Language)들을 이용하여 하드웨어를 소프트웨어적으로 기술하여 구축하고, 그 가상적인 시스템을 시뮬레이션하므로써 동작을 확인하고 설계시 발생하는 오류를 검출하였다.

이와 같은 설계방식에 대한 관심이 고조됨에 따라 근래 많이 사용되고 있는 HDL들로는 CDL[1], DDL[2], HDL[3], ISP[4], PMS[5], AHPL[6], Verlog-HDL[7] 및 VHDL[8-9]등이 있다. 이들 HDL들을 이용하여 하드웨어를 기술할 때는 고급언어의 구문으로 기술하는 기능적 수준의 기술방법과 하드웨어를 하부계층에서 부터 상부계층으로 기술하는 논리적 수준의 기술방법 등을 이용해 왔다.

본 연구에서는 HDL을 이용하여 간단한 명령을 갖는 16비트의 파이프라인 프로세서의 동작을 기능적 수준에서 기술하여 구현하였고, 새로운 기법의 시험벡터 입력방법[10-12]을 이용하여 구현된 프로세서를 시뮬레이션하고 동작을 확인하고자 하였다.

첫째, 구현하고자 하는 프로세서에서 실행되는 명령어의 세트를 먼저 설계하고, 두번째로, 구현된 명령의 실행에 필요한 명령어 서식을 형식화하여 이 명령들을 구현하고자 하는 시스템에 입력하도록 하는 시험환경을 구축하였다. 세번째로, 프로세서 사이클이 명령어의 인출, 실행 및 쓰기의 3단계를 겹치도록 하는 16비트의 파이프라인 프로세서의 사양을 규정하여 전체적인 시스템을 HDL을 이용하여 모듈별로 기술하고 구축하였다. 마지막으로 구현된 프로세서에 명령으로 구성된 시험벡터를 시험환경을 통해 입력하므로써 시뮬레이션하고 그것의 동작을 확인하였다.

16비트의 파이프라인 프로세서를 시뮬레이션하기 위해 본 연구에서 제시된 방법은 일반적으로 대부분의 HDL들을 이용하여 기술할 때의 설계 특징뿐만 아니라 설계가 용이하다는 장점도 역시 갖도록 하였고, 시험벡터를 명령으로 설정하여 시험환경을 통해 입력할 수 있게 하여 정확한 시뮬레이션이 가능하고 동작확인이 용이하다는 이점을 제공해 줄 수 있을 것으로 보았다.

Ⅱ. 파이프라인 프로세서의 제어

2.1 파이프라인 아키텍쳐

제어 사이클들을 한 클럭 동안 병행 처리하는 파이프라인 프로세서는 명령의 대열로 이동하는 명령 파이프라인으로 처리되었다. 여기서 파이프라인 처리를 위해 페치 사이클과 실행 사이클과 쓰기 사이클을 겹치게 하는 3단계 파이프라인 프로세서를 도입하였다. 그림1에서 3단계의 사이클을 파이프라인 처리와 순차적 처리방식을 비교 설명하고 있다.

단 처리에서 분기명령의 경우에는 미리 페치된 명령들은 버리고 분기된 번지의 명령부터 페치된다.

```
cycle  #without   with pipeline
       pipeline
 1       F1    F1
 2       E1    F2   E1
 3       W1    F3   E2   W1
 4       F2    F4   E3   W2
 5       E2    F5   E4   W3
 6       W2    F6   E5   W4
 7       F3     .
 8       E3     .
 9       W3     .
 10      F4     .
 11      E4     .
 12      W4    F12  E11  W10
 13      F5    F13  E12  W11
 14      E5    F14  E13  W12
 15      W5    F15  E14  W13

Without pipeline 5 instruction
  execute in 15 cycles. (#instruction*3)
With pipeline 5 instruction execute
  in 7 cycles. (#instruction+2)
```

그림 1. 3단계 파이프라인의 표현
Fig 1. Representation of a three-stage pipeline

2.2 파이프라인 프로세서의 기능적 분류

그림2는 그 프로세서의 구동 이벤트를 위한 트리거를 기술하고, 그림3은 일반적인 모듈에 사용되는 파라메타의 선언을 기술한다.

```
always @(posedge clock) begin : phase1_loop
  if (!reset) begin
    fetched=0;
    executed=0;
    if (!queue_full && !mem_access)
      ->do_fetched;
    if (queue_full || mem_access)
      -> do_execute;
    if (result_ready)
      -> do_write_results;
    end
  end
```

그림 2. 3단계 이벤트의 트리거링
Fig 2. Triggering simultaneous events

```
parameter QDEPTH=3;
reg[0:WIDTH-1]
            IR_queue[0:QDEPTH-1], wir;
reg[0:WIDTH] wresult;
reg[0:2]      eptr, fptr, qsize;
            reg, mem_access,
            branch_taken, halt_found;
reg         result_ready;
reg         executed, fetched;
wire        queue_full;
event       do_fetch, do_execute,
            do_write_results;
```

그림 3. 파이프라인 모델의 부가 선언
Fig 3. Additional declarations for pipeline modeling

2.3 페치 타스크의 처리

연속적인 명령을 실행하기 위해 읽어온 명령을 대기시켜 놓는 명령큐 IR_queue의 명령의 개수를 qsize가 지시하고, etpr은 IR_queue의 빈 공간을, fptr은 IR_queue의 채워져 있는 공간을 지시한다. mem_access 플래그는 Load나 Store명령이 들어오면 사이클을 버리도록 페치 타스크에 지시한다. 이 페치 타스크는 아래와 같이 기술된다.

```
task fetch;
begin
  IR_queue[fptr]=mem[pc];
  fetched=1;
end
endtask
```

여기서 fetched 플래그는 한번의 페치 사이클이 수행되었다는 것을 지시한다.

2.4 실행 타스크의 처리

Queue로부터 읽어온 현재의 명령을 해독하여 실행하는 타스크로서 ALU에 입력되는 입력 레지스터 src1과 src2가 처리되고, 처리된 자료는 Output 레지스터 기억된다. 대부분의 명령은 1-사이클 동안 처리되나 2-사이클로 수행되는 Load명령의 모델 세그멘트는 그림4에 기술된다. 첫 사이클에서 명령을 읽고, 실행 사이클에서 한번 쉬게 된다.

```
if (!mem_access)  ir=IR_queue[eptr];
  'LD: begin
       if (mem_access==0)
          mem_access=1;
       else begin
       .....................................
       end
  end
```

그림 4. 로드 명령에 대한 메모리 억세스
Fig 4. Memory access for Load instruction

실행 타스크는 mem_access플래그가 '1'일 때는 명령을 읽지 못하고 앞 사이클의 Load명령을 수행하며, 메모리 전송의 끝에서 실행 타스크는 플래그를 '0'으로 하고 명령 Queue로부터 IR로 명령을 넣고, 메모리로부터 Queue로 명령을 채운다. 특히 읽어온 명령이 Branch이거나 Halt명령일 때는 그림5의 Flush queue 타스크에 의해 Queue전체를 비운다.

```
task flush queue
begin
  fptr=0;
  eptr=0;
  qsize=0;
  branch taken=0;
end
end task
```

그림 5. 파이프라인의 플러쉬
Fig 5. Flushing the pipeline

2.5 쓰기 타스크 처리

실행과 쓰기 단계는 레지스터 wresult 와 wir사이에서 자료를 전달하는 동작을 하는 사이클로서, result의 내용이 복사되고 ir이 wir로 복사된다. 다음 사이클에서 wresult가 목적지로 복사된다. 그림6과 그림7은 동작을 기술하는 copy_results와 write_result 타스크이다.

Result의 결과를 Result로의 복사는 먼저 실행 타스크의 Negative 클럭 사이클동안 Result를 변형하여 쓰기 타스크의 다음 사이클의 Positive Half동안 레지스터 파일로 그것을 복사하는 동작과 파이프라인의 쓰기 사이클을 제거하여 레지스터 파일에 직접 ALU의 결과를 쓰는 두 가지의 방법이 있다.

2.6 Phase-2의 제어

파이프라인 처리는 동기 설계로서 클럭 사이클의 Positive Half에서 동작하며, Negative Half에서는 읽어온 명령이 Branch일 때는 PC를 변경하여 그 명령 이후의 새로운 결과로부터 조건코드를 세트한다. 역시 Negative Half에서는 IR과Result가 각각 wir과 wresult에 복사되고, 명령 큐에 대한 페치와 실행 참조 포인터 (fptr, eptr)가 변경된다.

```
task copy results;
begin
   if (('OPCODE)='ADD) &&
      ('OPCODE('HLT)) begin
       setcondcode(result);
       wresult=result;
       wir=ir;
       result ready=1;
   end
end
endtask
```

그림 6. 결과와 명령의 복사
Fig 6. copying results and instruction

```
task write result;
begin
  if (('WOPCODE)='ADD) &&
     ('WOPCODE('HLT)) begin
     if ('WDSTTYPE=='REGTYPE);
        RFILE('WDST)=wresult;
     else MEM('WDST)=wresult;
     result ready=0;
     end
  end
end
endtask
```

그림 7. 결과를 레지스터 파일에 쓰는 동작
Fig 7. writing results in the register file

그림8은 타스크 set_pointer와 Phase-2의 처리의 동작을 기술하는 것으로 Branch명령을 만났을 때는 파이프라인 속의 Branch이전의 명령은 수행되어야 하기 때문에 Negative Half에서 명령큐는 비워진다. 그리고 쓰기 단계에서는 실행단계가 halt_found 플래그를 세트할 때 다음 사이클의 positive half에서 이전의 명령의 결과를 저장한다.

```
task set pointers;
begin
  case ({fetched, executed})
    2'b00::
    2'01: begin
            qsize=qsize-1;
            eptr=(eptr+1) % QDEPTH;
          end
    2'b10: begin
            qsize=qsize+1;
            fptr=(fptr+1) % QDEPTH;
          end
    2'b11: begin
            eptr=(eptr+1) % QDEPTH;
            fptr=(fptr+1) % QDEPTH;
          end
  endcase
end
end task

always @ (negedge clock) begin:
          phase_2 loop
   if (!reset) begin
     if (!mem_access && !branch_taken)
        copy_results;
     if (branch_taken) pc='DST;
     else if (!mem_access) pc=pc+1;
     if (branch taken ||halt found)
        flush queue;
     else set pointers;
     if (halt found) begin
        $stop;
        halt found=0;
     end
   end
end
```

그림 8. Phase2의 제어동작
Fig 8. Phase-2 control operations

2.7 Interlock 문제

파이프라인 구조에서는 레지스터의 내용을 변경시키는 명령들이 동일한 레지스터를 쓰고 읽는 동작이 동시에 수행될 때 발생하는 인터락 문제가 발생한다

예) I3: ADD R1,R2 // R1=R1+R2
I4: CMP R3,R1 // R3=~R1

위의 예에서 R1과 R2를 더하는 동작을 수행한 후에 그 결과를 R1에 쓰는 시간에 R1의 보수를 R3에 넣는 것으로 인해 R1이 인터락된다. 즉 실행단계와 쓰기 단계의 병렬성 때문에 I4의 실행단계는 I3의 쓰기 단계가 수행되기 전에 이전의 R1을 먼저 읽게 되므로 잘못된 R1을 갖고서 수행된다. 이 인터락 문제를 해결하기 위해서는 소프트웨어적으로 해결하는 방법과 하드웨어적으로 해결하는 방법이 있다. 소프트웨어적으로는 NOP 명령을 두 명령 사이에 삽입한다.

I3: ADD R1,R2 // R1=R1+R2
IX: NOP
I4: CMP R3,R1 // R3=~R1

하드웨어적으로 이 문제를 피하는 방법은 부가적으로 논리설계를 추가하는 것으로 그림9에 보여진다.

```
reg bypass;
function[0:31] getsrc;
input[0:31] i;
begin
   if (bypass) getsrc=result;
   else if ('SRCTYPE===REGTYPE)
      getsrc=RFILE['SRC];
   else getsrc='SRC;
end
end function

function[0:31] getdst;
 input[0:31] i;
begin
   if (bypass) getdst=result;
   else if ('DSTTYPE===REGTYPE)
      getdst=RFILE['DST];
   else $diplay ("Errror : immediate data
                  cannot be destination");
end
end function
always @ (do_execute) begin :
          execute_block
   if (!mem_access) begin
      ir=IR_Queue[eptr];
      bypass=(('SRC=='WDST) |
              ('DST=='WDST));
   end
   execute;
   if (!mem_access) executed=1;
end
```

그림 9. Bypass를 위한 함수의 변형과 처리
Fig 9. Modified functions and execute process for bypass

Ⅲ. 명령어 세트의 모델링과 명령의 실행

3.1 SISC의 블록 다이어그램

설계하고자 하는 SISC(Simple Instructional Set Computer)의 블록 다이어그램은 그림10에 보여진다. 그 시스템에서 PSR은 프로세서 수행 후의 조건코드 플래그를 세트하는 5비트 레지스터이다. 그리고 SISC에서 사용되는 모든 레지스터의 비트 폭은 표1에 보여진다.

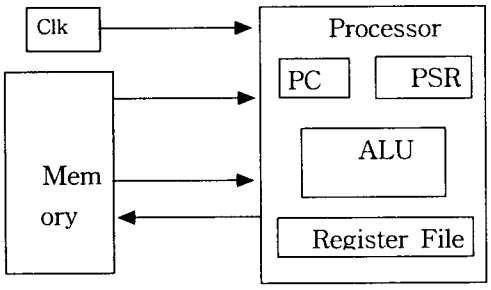

그림 10. SISC의 블럭 다이어그램
Fig 10. Block diagram of SISC

표 1. SISC의 레지스터 규격
Table 1. A register's specification for SISC

register	specification	Function
MEM	212×32	32-bit Word Memory
RFILE	32-bit	16개의 32-bit Reg
ir	32-bit	Instruction Reg
src1	32-bit	Operand1 Reg
src2	32-bit	Operand2 Reg
result	33-bit	결과를 Hold하는 Reg
PC	12-bit	Program Counter
psr	5-bit	Status Reg
dir	1-bit	Shift/Rot의 방향 Reg
reset	1-bit	System Initial Reg

3.2 Instruction Format

명령 레지스터는 각 비트의 필드마다 주어진 2진 코드에 따라 명령의 종류나 명령의 실행시에 요구되는 자

료의 소스나 목적지 형식과 번지의 소스나 목적지 등의 정보들을 갖도록 구성된다. ir의 필드의 코드값에 따른 기능은 다음과 같이 구분되어 설명된다. 명령어 서식을 명령어 코드 형식으로 표현하면 그림11처럼 도시된다.

각각의 명령은 다음의 표2에서 보여주는 바와 같이 6비트의 2진코드로 할당하였다. 각 명령에서의 동작코드와 오퍼란드의 구성은 항상 Op-Code, 그리고 Oprand1과 Operand2로 이루어지도록 하였다.

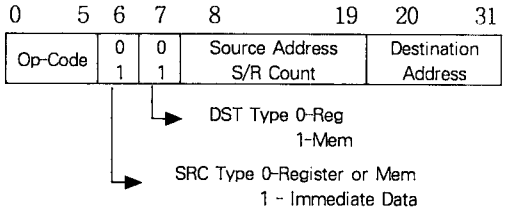

그림 11. 명령어 서식
Fig 11. Instruction Format

표 2. 명령어의 2진 코드
Table 2. Binary code for Instructions

Instruction	Binary Code	Mnemonic
Load	0 0 0 0 1 0	LD
Store	0 0 0 0 1 1	STR
Add	0 0 0 1 0 0	ADD
Multiply	0 0 0 1 0 1	MUL
Complement	0 0 0 1 1 0	CMP
Shift	0 0 0 1 1 1	SHF
Rotate	0 0 1 0 0 0	ROT
No-operation	0 0 0 0 0 0	NOP
Halt	0 0 1 0 0 1	HLT
Branch	0 0 0 0 0 1	BRA

Ⅳ. SISC의 구성과 기술

4.1 SISC 파라메타와 레지스터의 선언

SISC내의 모든 모듈에 공통으로 사용되는 파라메타라든가 레지스터 등의 규격을 우선하여 선언하고 정의한다. 그 정의는 시스템기술에서 정의하고 선언한다.

4.2 Task와 Function의 기술

```
function[0:WIDTH-1] getsrc;
input[0:WIDTH-1] in;
begin
   if ('SRCTYPE===='REGTYPE) begin
      getsrc=RFILE['SRC];
   end
end
end function
```

이 동작은 읽어온 명령코드 중 ir[6]의 SRCTYPE을 검사하여 소스 형식을 결정하고 getsrc에 넣는다.

```
function[0:WIDTH-1] getdst;
input[0:WIDTH-1] in;
begin
   if ('DSTTYPE===='REGTYPE) begin
      getdst=RFILE['DST];
   end
   else begin
   $diplay ("Error : Immediate data can't
          be dsestination");
   end
end
end function
```

Function getsrc와 마찬가지로 읽어온 명령중 ir[7]의 DSTTYPE을 검사하여 목적지의 형식을 결정하여 getdst에 넣고 그것이 '1'이면 Error를 표시한다.

```
function checkcond;
input[0:4] ccode;
begin
   case (ccode)
      'CCC : checkcond='CARRY;
      'CCE : checkcond='EVEN;
      'CCP : checkcond='PARITY;
      'CCZ : checkcond='ZERO;
      'CCN : checkcond='NEG;
      'CCA : checkcond='1;
   end case
end function
```

이 동작은 ccode를 받아서 각 조건비트를 세트한다.

새로운 동작을 수행하고자 할 때는 이전의 동작에 의한 psr의 내용을 리세트시켜야 하는데 이때 psr을 리세트시키는 타스크는 clearcondcode이다.

```
task clearcondcode;
begin
  'psr;
end
endtask
```

이 타스크에서는 조건코드값을 기억하는 psr의 5개 비트를 '0'으로 리세트시켜 준다. 반면에 Function checkcond를 수행한 후에는 그것의 각 상태비트를 세트하는 동작은 타스크 setcondcode이다.

```
task setcondcode;
input[0:WIDTH] res
begin
   'CARRY = res[0];
   'EVEN = ~res[32];
   'PARITY = res;
   'ZERO = ~(|res);
   'NEG = res[1];
end
endtask
```

여기에서는 한 개의 명령을 수행한 후의 결과는 res에 복사되고, 이 res의 내용을 갖고서 상태비트의 플래그를 세트-리세트하게 된다.

4.3 파이프라인 모듈의 제어

지금까지는 시스템을 구성하여 동작이 개시되기 전에 사용되는 변수, 파라메타, 레지스터 및 그들의 폭 등을 선언하는 동작을 기술하였으며, 역시 명령형식을 결정하고 명령코드를 기술하였다. 따라서 이제부터는 프로세스를 모델링하고 그것의 동작을 모듈별로 기술한다.

파이프라인 제어를 모델링하는 모듈에서는 모든 파라메타의 선언과 함수와 타스크를 먼저 기술하여야 한다. 또한 파이프라인 제어를 수행하는 데 필요한 레지스터나 제어라인들도 역시 부가적으로 선언해야 하며, 결과를 저장하기 위해 필요한 명령필드들도 정의해야 한다.

파라메타 선언	제어 레지스터 정의
HALFCYCLE= (CYCLE/2) QDEPTH=3	reg[0:WIDTH] IR QUEUE[0:QDEPTH-1] wir reg[0:2] eptr, fptr, qsize reg clock reg[0:WIDTH] wresult

여기에서 QDEPTH는 명령큐의 깊이이고, wir은 쓰기 단계에서의 명령 레지스터이며, eptr, fptr, qsize 등은 포인터들을 검사하는 플래그이다. clock은 시스템 클럭을 나타낸다. queue_full에 대한 연속적인 입력은 assign queue_full=(qsize== QDEPTH)로 기술된다.

4.4 동작개시 후의 Function과 Task의 기술

동작개시 후의 타스크들은 페치 사이클과 모든 명령에 대한 실행 사이클과 결과를 기억시켜 놓는 쓰기사이클 등에 대한 시스템이 동작할 때 요구되는 순서대로 기술하고 있으며, 그 외에 결과를 복사하고 출력하는 동작들이 기술된다.

페치 타스크는 프로그램 카운터가 지시하는 메모리 번지의 내용을 명령 IR_Queue[fptr]에 넣는 동작을 하며, 한 개의 명령을 페치한 다음 실행 타스크로 제어를 넘겨준다. 여기서 fptr은 큐의 깊이를 나타내는 파라메타이다.

```
Task fetch;
begin
   IR_Queue[fptr]=MEM[pc];
   fetched=1;
end
endtask
```

실행 타스크는 eptr이 지시하는 IR_Queue의 내용을 ir로 읽어들이고, 이 ir의 명령은 각각 동작코드에 따라 분류되어 명령에 맞는 동작을 수행하도록 제어를 옮기게 된다. 즉, mem_access를 검사하여 '0'이면 eptr이 지시하는 IR_Queue의 명령을 ir로 넣는다.

No-operation은 파이프라인 처리에서 인터락 문제를 해결하기 위해 사용하는 명령으로 debug만 검사하여 "Nop...."를 표시한다.

```
'NOP: begin
      if (debug) $display("Nop...");
      end
```

Branch명령은 조건코드를 검사하여 '1'이면 읽어온 명령의 'DST를 pc로 변경하고 pc가 지시하는 위치로 분기 동작을 하고, 명령을 수행한 후에 Branch_taken을 '1'로 세트한다.

```
'BRA: begin
      if (debug) $write ("Branch.");
         if (checkcond ('CCODE)==1) begin
            pc='DST
```

```
            branch_taken=1;
          end
        end
```

Load명령에 대한 타스크는 'SRC를 'DST가 지시하는 레지스터 파일에 넣거나 그 'SRC가 지시하는 메모리 번지의 내용을 'DST가 지시하는 레지스터 파일에 넣는다. 그 다음에 mem_access를 '0'으로 리세트시켜 둔다.

```
'LD: begin
    if (mem_access==0) begin
      mem_access=1'
    end
    clearcondcode;
    if ('SRCTYPE) begin
      RFILE['DST]='SRC;
    else RFILE['DST]=MEM['SRC];
      setcondcode ({'1b0, RFILE['DST]});
      mem_access=0;
    end
    end
```

Store명령에 대한 타스크는 명령의 'DST가 지시하는 메모리 번지에 'SRC를 저장하거나 'SRC가 지시하는 레지스터의 내용을 'DST가 지시하는 메모리 번지에 저장한다. 이 동작을 수행한 후에는 mem_access를 '0'으로 리세트시켜 둔다.

```
'STR: begin
      if (mem_access==0) begin
        mem_access=1;
      end
      else begin
        if (debug) $dis[lay (Store...");
      clearcondcode;
      if ('SRCTYPE) begin
        MEM['DST]='SRC;
      end
      else MEM['DST]=RFILE['SRC];
        mem_access=0;
      end
      end
```

Add명령의 타스크는 getsrc 타스크로부터 ir의 소스를 src1으로, getdst로부터 ir의 목적지를 src2로 두고서 scr1과 src2를 더하여 결과를 result에 넣는다. 이 result의 결과에 의해 조건코드를 세트한다.

```
ADD: begin
     clearcondcode;
     src1=getsrc(ir);
     src2=getdst(ir);
     result=src1+src2;
       setcodcode(result);
     end
```

Multiplay명령의 타스크는 읽어온 명령의 ir중에서 타스크 getsrc와 gestdst로부터 각각 src와 dst를 src1과 src2로 넣고 그것을 곱하여 result에 넣고 조건코드를 세트한다.

```
MUL: begin
     clearcondcode;
     src1=getsrc(ir);
     src2=getdst(ir);
     result=src1*src2;
     setcondcode(result);
     end
```

Complement명령의 타스크는 getsrc에 의해 ir의 src를 src1으로 전송하고, 그 src1의 전체를 1의 보수를 취하여 결과를 result에 넣는다.

```
CMP: begin
     clearcondcode;
     src1=getsrc(ir);
     result=~src1;
     setcondcode(result);
     end
```

Shift명령의 타스크는 gestsrc로부터 ir의 소스를 src1에, getdst로부터 ir의 목적지를 src2에 넣고, src2의 i값만큼 우측으로 시프트하거나 좌측으로 시프트하여 result에 결과를 넣는다. 그 다음에 조건코드를 세트한다.

```
SHF: begin
     src1=getsrc(ir);
     src2=getdst(ir);
     i=src1[0:ADDRSIZE-1];
     result=(i>=0) ? (src2>>i) : (src2<<-i);
     setcondcode(result);
     end
```

Rotate명령의 타스크는 getsrc로부터 src1의 20번 비트의 부호를 검사하여 좌우측 로테이트를 수행시킨다. 여기서 로테이트 동작은 1비트의 로테이트 동작을 src2의 값만큼 반복시켜 로테이트의 횟수를 결정하고 결과는 result에 넣은 후 조건코드를 세트한다.

```
ROT: begin
```

```
clearcondcode;
src1=getsrc(ir);
src2=getdst(ir);
dir=(src1[0]==0) ? 'RIGHT: 'LEFT;
i=(src1[ADDRSIZE-1]==0) ?
  src1: -src1[ADDRSIZE-1:0];
while (i>0) begin
  if (dir=='RIGHT) begin
    result=src2>>1;
    result[1]=src2[20];
  end
  else begin
    result=src2<<1;
    result[32]=src2[20];
  end
  i=i-1;
  src2=result;
end
setcondcode(result);
end
```

Halt명령의 타스크는 먼저 halt_found=1로 세트해 두고 프로그램의 수행을 중지시킨다.

```
HLT: begin
     $display("Halt...");
     halt_found=1;
     end
```

지금까지 기술한 각각의 타스크 중 한 개의 명령을 수행할 때 잘못된 명령이었다면 다음에 기술된 것처럼 에러 메시지를 표시하도록 하였다.

```
default: $display("Error: Wrong Op-code in instruction.");
        if (!mem_access) executed=1;
```

여기서는 실행 타스크를 완료하고 마지막으로 mem_access가 '1'이 아니면 executed를 '1'로 세트하고 실행 사이클의 끝남을 지시한다.

결과저장 타스크는 실행 타스크에서 실행한 결과를 원하는 메모리나 레지스터에 저장하는 동작을 수행한다. 수행한 명령코드 wir[0:5]를 검사하여 메모리 또는 레지스터에 저장하고, 그 동작을 한 후에는 result_ready를 '0'으로 리세트시키고는 타스크를 완료한다.

```
task write_result;
begin
if (('WOPCODE>='ADD) &&
   ('WOPCODE<'HLT)) begin
      If('WDSTTYPE==′REGTYPE)
        RFILE['EDST]=wresult;
    else MEM['WDST]=wresult;
result_ready=0;
  end
end
end task
```

Flush_queue 타스크는 현재의 큐를 비워두는 동작으로 fptr, eptr, qsize를 모두 '0'으로 만드는 동작을 한다. 그리고는 branch_taken을 '0'으로 리세트한다. pc는 이미 Branch명령에 의해 변형되어 있는 상태이다.

```
task flush_queue;
begin
   fptr=0;
   eptr=0;
   qsize=0;
   branch_taken=0;
end
endtask
```

copy_result 타스크는 쓰기 타스크 이전에 result의 내용을 복사해 두는 동작과 현재의 명령의 ir을 wir로 전송해 두는 동작을 수행하며, 동작코드를 검사하여 result를 wresult로, ir을 wir로 전송하는 동작을 한다. 그 다음에 result_ready를 '1'로 세트한다.

```
task copy_result;
begin
   if (('OPCODE>='ADD) &&
      ('OPCODE<'HLT)) begin
      setcondcode(result);
      wresult=result;
      wir=ir;
      result_ready=1;
   end
endtask
```

set_pointer 타스크는 큐의 Full과 Empty 상태를 표시하는 3개의 eptr과 fptr의 상태를 세트하고, 또 qsize를 증감시키는 동작을 한다. 이 동작은 파이프라인의 병렬처리와 연관되므로 eptr과 fptr은 각각 eptr+1, fptr+1을 QDEPTH로 나눈 나머지 값으로 세트된다.

```
task set_pointer;
begin
   case ({fetched, executed})
      2'b00::
      2'b01: begin
```

```
            qsize=qsize-1;
            eptr=(eptr+1) % QDEPTH;
            end
        2'b10: begin
            qsize=qsize+1;
            fptr=(fptr+1) % QDEPTH;
            end
        2'b11: begin
            eptr=(eptr+1) % QDEPTH;
            fptr=(fptr+1) % QDEPTH;
            end
    endcase
endtask
```

disprim 타스크는 레지스터와 메모리의 내용을 표시하는 동작을 수행하는 것으로서 입력의 adr1과 adr2를 읽어 들여 rm을 검사하여 adr1이 지정하는 레지스터나 메모리의 내용을 표시하고는 adr1을 증가시킨다.

```
task disprim
input rm;
input[ADDRSIZE-1:0] adr1, adr2;
begin
  if (rm=='REGTYPE) begin
       while(adr2>=adr1) begin
         $display ("REGFILE[%d]=%d\n", adr1,
    RFILE[adr1]);
          adr1=adr1+1;
          end
  end
  else begin
       while(adr2>=adr1) begin
         $display      ("MEM[%d]=%d\n",
     adr1, MEM[adr1]);
       end
  end
end
endtask
```

Apply_reset 타스크는 주로 시스템의 플래그나 프로그램 카운터를 리세트하는 동작을 수행한다.

```
task apply_reset;
begin
  reset=1;
  @(negedge clock) #1;
  reset=0;
  pc=0;
  mem_access=0;
  branch_taken=0;
  halt_found=0;
  qsize=0;
  result_ready=0;
  eptr=0; fptr=0;
  executed=0; fetched=0;
end
endtask
```

4.5 Initial 블록과 Always 블록

구성한 시스템을 초기에 구동할려면 지금까지 기술한 타스크나 Function을 MEM에 로드하여야 하는데, 이 로드 동작이 $readmemb에 의해 수행되고, 그리고 apply_reset 타스크를 호출하여 시스템을 모두 리세트 시켜 두게 된다.

```
initial begin: setup_and_prog_load
  'DEBUG_OFF;
  waves;
  $readmemb ("RISC.prog", MEM);
  apply_reset;
end
```

그리고 로드되고 리세트된 시스템에는 클럭이 다음과 같이 주어진다.

```
initial begin: clock_loop
  clock=1;
  forever begin
    #(HALFCYCLE clock=~clock;
  end
end
```

리세트된 시스템의 동작은 주어진 클럭에 따라 동기되어 동작하므로 역시 클럭도 일종의 초기블럭에서 주어져야 한다. 따라서 Positive 클럭은 초기에 '1'로 설정되어 phase1_loop로, Negative 클럭은 '0'(~clock)으로 phase2_loop로 분류되어 요구하는 동작을 동기시켜 준다. phase1_loop (Positive clock)는 do_fetch로 제어를 옮겨 수행하거나 do_execute로 가서 수행하거나 do_write_result로 제어를 옮겨 수행한다.

```
always @(posedge clock) begin: phase1_loop
 if (!reset) begin
  fetched=0; execute=0;
    if (!queue_full && !mem_access) →
      do_fetch;
    if (qsize || mem_access) → do_execute;
      if (result_ready) → do_write;
  end
end
```

phase2_loop(Negedge clock)는 타스크 copy_result를 수행하고, 조건코드를 세트시켜 주게 되며, 명령의 'DST를 pc로 치환하거나 pc를 하나 증가시키게 된다. 역시 타스크 flush_queue를 수행하거나 타스크 set_pointer를 수행한다. 또한 시스템의 동작을 중지하고 halt_found를 '0'으로 리세트하여 동작을 끝낸다.

```
always @(negedge clock) begin: phase2_loop
  if (!reset) begin
    if (!mem_access && !branch_taken)
    copy_results;
    if (branch_taken) pc='DST;
    else if (!mem_access) pc=pc+1;
    if (branch_taken || halt_found)
   flush_queue
    else set_pointer;
  if (halt_found) begin
    $stop;
    halt_found=0;
  end
  end
end
```

마지막으로 항상 수행해야 하는 동작으로는 Fetch, Execute와 Write의 3개의 이벤트에 의해 제어되는 3-단계 파이프라인이다. 따라서 각 단계의 동작을 기술하면 다음과 같다. 이 단계들은 Posedge 클럭의 Phase1_loop에서 조건에 따라 각각 호출되어 수행되는 타스크들이다.

```
Fetch stage
always @(do_fetch) begin: fetch_block
      fetch;
end
Execute stage
always @(do_execute) begin: execute_block
      execute;
end
Write stage
always @(do_write_results) begin
      write_results;
end
```

지금까지 기술한 모듈별 동작의 제어흐름과 그들을 상호 연결을 구현하였고, 구축된 파이프라인 프로세서의 동작을 시뮬레이션하기 위해 시험벡터를 입력하여 그 동작을 확인하도록 하였다.

V. 테스터 벡터의 입력과 시뮬레이션

아래의 테스트 벡터를 프로그램하여 입력한다.

```
prog
init
sto 3, 30
lod 30, 1
sto 50, 31
lod 31, 2
add 1, 2
sto 2, 32
hlt
```

위의 프로그램에 의한 실행은 프로그램 카운터의 순서에 따라 레지스터에 저장되는 값과 실행되는 타스크들과 중간 결과들을 중심으로 시뮬레이션되고 있으며, 그리고 각 프로그램카운터에 의한 3단계 파이프라인의 동작도 확인되고 있다.

```
      pc          — d'100
[ 1] fetch        — ir <- h'0F00301E,
     execute no,
     write_results no
[ 2] copy_results, set_pointers
     pc          -- d'101
[ 3] fetch        — ir <- h'0901E001,
     execute sto  —  ram[30] <- h'00000003,
     write_results no
[ 4] copy_results, set_pointers
     pc          -- d'102
[ 5] fetch        -- ir <- h'0F03201F,
     execute lod  --  rfile[1] <- h'00000003,
     write_results —  result <- h'000000000
[ 6] copy_results, set_pointers
     pc          — d'103
[ 7] fetch        — ir <- h'0901F002,
     execute sto  —  ram[31] <- h'00000032,
     write_results — result <- h'000000000
[ 8] copy_results, set_pointers
     pc          -- d'104
[ 9] fetch        — ir <- h'10001002,
     execute lod  --  rfile[2] <- h'00000032,
     write_results — result <- h'000000000
[10] copy_results, set_pointers
     pc          -- d'105
[11] fetch        -- ir <- h'0D002020,
```

```
      execute add,
      write_results -- result <- h'000000000
[12]  copy_results, set_pointers
      pc          -- d'106
[13]  fetch       -- ir <- h'24000000,
      execute sto -- ram[32] <- h'00000032,
      write_results -- result <- h'000000035
[14]  copy_results, set_pointers
      pc          -- d'107
[15]  fetch       -- ir <- h'00000000,
      execute hlt,
      write_results -- result <- h'000000035
[16]  copy_results, set_pointers
```

Ⅵ. 결론

본 연구에서는 파이프라인 프로세서의 시스템을 HDL을 이용하여 기능적 수준에서 기술하고 동작을 시뮬레이션하고 확인하였다. 그 프로세서를 구현할 때 모든 동작을 Function과 타스크별로 구현하여 주 시스템에서 호출하여 사용할 수 있도록 하였다. 구현된 프로세서를 시뮬레이션하기 위해서 프로세서에서 수행되는 명령어를 기호언어로 구현하고, 구현된 명령을 제시된 실험체제의 환경하에서 테스트 벡터를 입력하여 그 시스템의 동작을 시뮬레이션하였다. 그리고 동작을 확인할 때 파이프라인 동작특성의 수행과정도 역시 클럭 순서별로 확인하였다.

이제까지 알려진 시스템의 시뮬레이션의 방법들은 HDL을 이용하여 시스템을 문서화하고 동작을 기술하는 과정은 대부분 비슷하였으나, 그 시스템을 시뮬레이션할 때 입력되는 자료나 명령을 2진 스트링으로 입력하므로 해서 어떤 동작을 위한 시뮬레이션 자료인지를 식별하기 어려웠고, 동작을 확인하기도 어려웠다. 따라서 본 연구에서는 시스템을 기술하고 구현하는 과정은 일반적인 HDL을 갖고서 수행하는 것과 동일한 방법으로 처리한 반면에, 시뮬레이션을 하기 위해 입력하는 테스트 벡터를 기호언어로 구현하여 시뮬레이션시키므로 써 시스템의 동작에 대한 목적과 과정을 쉽게 식별하고 확인할 수 있도록 하는 장점을 가지도록 하였고, 실험체제를 통해 기호화된 명령어를 프로그램하여 입력하므로 써 정확한 시뮬레이션을 수행할 수 있도록 하였다.

참고문헌

[1] Y.Chu, "Structure of CDL programs", Dep. computer science, Univ. Maryland, Tech. not pp.74-58, May 1974.

[2] J.R.Duley and D.L.Dietmeyer, "Translation of a DDL digital system specification to boolena equation", IEEE Trans. on computers vol. C-18, pp.305-313, 1970.

[3] S.J.Piatz, "HDL: A comprehensive H/W design language", Proc. of the 17th Hawaii International conference on system science 1984, vol. 2, Han. 1984.

[4] M.R.Barbacci, C.G.Bell and A.Newell, "ISP: A language to describe instruction sets and other register transfer systems", In Proc. IEEE computer conf., Compconf-72 San Francisco, Calif., Sept., pp. 219-222, 1972.

[5] C.G.Bell and A.Newell, "The PMS and ISP ve systems for computer structures", In Proc. 1970. Spring joint computer Conf.(AFIPS), pp. 351-374.

[6] R.E.Swanson, Z.Navabi and F.J.Hill, "An AHPL compiler/simulator system", In Proc. sixth Texas conf. computer system., pp.1-11, 1977.

[7] E.Sternheim, R.Singh and Y.Trivedi, "Digital design with Verilog HDL", Automata Publishing company, 1990.

[8] S.Sjoholm and L.Lindh, "VHDL for designers", Prentice-Hall, 1997.

[9] Y.C.Hsu, K.F.Tsai, J.T.Liu and E.S.Lin, "VHDL modeling for digital design synthesis", Toppan Kluwer. 1997.

[10] 박두열, "IDL을 이용한 16-비트 ISP의 설계와 시뮬레이션에 관한 연구", 한국통신학회, 제15권 1호, pp. 29-42, 1990.1.
[11] 박두열, "APL을 이용한 소형명령 컴퓨터의 설계 와 시뮬레이션에 관한 연구", 동아대학 대학원 박사학위논문, 1990.
[12] 박두열, "논리언어를 이용한 파이프라인 프로세서의 Modeling과 시험에 관한 연구", 동주대학논문집, 제16권, pp.363-408.

저 자 소 개

박두열
1980.2.:동아대학교 공과대학 전자공학과 졸업
1982.2.:동아대학교 대학원 공학석사
1989.9.:동아대학교 대학원 공학박사
1985.3~현재:동주대학 부교수 재직
관심부야: 컴퓨터 H/W Description, Computer Vision,Multimedia