논문 99-24-5B-3 한국통신학회논문지 '99-5 Vol.24 No.5B

리턴 어드레스의 우선 패취가 가능한 흐름 제어 명령어 처리 구조

정회원 박 주 현*, 김 영 민**

Architecture of processing of flow control instruction capable of prefetching of return address

Ju-Hyun Park*, Young-Min Kim** *Regular Members*

요 약

본 논문은 파이프라인 구조를 갖는 프로세서에서 흐름 제어 명령어 수행의 오버헤드를 줄이기 위해 파이프라인 초기 단계에서 리턴 어드레스를 우선 패취하여 명령어 수행 사이클 수를 줄일 수 있는 방법을 제안한다. 본 논문에서 제안한 구조는 4단의 파이프라인에 기초하고 있으며, 우선 리턴 블록을 가지고 있지 않는 일반적인 구조와 비교하여 파이프라인 프로세서 처리 시간을 25%까지 단축시킬 수 있다. 제안한 흐름 제어 명령어 처리 구조의 각 모듈은 Model Technology Inc.의 V-System을 이용하여 VHDL 시뮬레이터 상에서 서브루틴 호출, 분기, 리턴 명령어를 사용한 간단한 프로그램 수행을 적용하여 검증하였다.

ABSTRACT

In this paper, we propose a method to reduce overhead of processing of flow control instruction in the pipeline architecture by performing pre-fetching in the initial stage of pipeline processing. The basis of pipeline processing is a four-stage configuration. In a program routine with subroutine call and return instructions, a pre-return processing module reduces pipeline processing time by 25% as compared to the conventional processing architecture. Each module of the proposed architecture of processing of flow control instruction is described in VHDL structurally and behaviorally and whether it is working well or not is checked on V-System simulator of Model Technology Inc. by processing a simple program with a call, a jump and a return instruction.

I. 서 론

LIFO(Last In First Out)는 1950년대 이후 컴퓨터에 이용하기 시작했다. 원래 이러한 스택은 고급 언어의 수행 효율을 높이기 위해 추가되곤 하였다. 프로시저 내의 정보의 임시 저장을 위한 메카니즘을 형성하기 때문에 과거에 수행한 루틴으로부터 데이터를 파괴하지 않고 반복되는 프로시저의 수행을 할 수 있도록 한다. 그 이후 하드웨어 설계자에게는 대부분의 컴퓨터에서 2차적인 데이타 처리를 위한 구조 블럭으로 여겨져 왔다[1].

과거와 현재의 스택 구현의 가장 큰 차이는 고속의 전용 스택 메모리를 사용하는 것이 현재는 가격 대비 효율이 높다는 것이다. 이러한 스택은 빠른 서브루틴 호출 성능을 제공하고, 인터럽트 핸들링, 태스크 스위칭(task switching) 등에 있어 높은 성능을 보여 주고 있다. 이러한 특징은 컴퓨터 시스템을 좀 더 빠르고, 효율적인 제작이 가능하게 한다. 하드웨

* 한국전자통신연구원 집적회로설계연구부 VLSI구조팀(pjhyun@etri.re.kr),
** 전남대학교 전자공학과(kym@chonnam.chonnam.ac.kr)
논문번호 : 98182-0423, 접수일자 : 1998년 4월 23일

어로 스택을 구현하면 소프트웨어 구현 보다 훨씬 더 빠르다는 장점이 있다. 명령어의 상당 부분을 스택과 관련된 명령어로 사용하는 프로세서에서는 이와 같은 효율을 증가시키는 것이 고성능의 시스템을 얻는 데 중요하다. 일반적으로 하드웨어 구현은 스택 포인터에 메모리의 위치를 보존해 두는 것이다. 보통 포인터는 전용 하드웨어 레지스터로 구현되며, push/pop 동작에 따라 증가하거나 감소한다. 하드웨어로 스택을 구현하는 또 한 가지 방법은 각 요소를 레지스터로 구현한 레지스터 파일 또는 레지스터 어레이를 이용하는 것이다.

그러나 스택의 가장 중요한 성질은 최상위 요소만을 접근할 수 있기 때문에 프로그램의 소형화, 하드웨어 단순화, 수행 속도 면에서 성능을 저하시키는 결과를 초래한다. 특히 명령어의 흐름을 혼란시키는 흐름 제어 명령어 수행시 전 단계의 모든 처리 과정을 취소시킬 수도 있으며, 다음 단계에 처리해야 할 명령어를 새로 패취해야 하는 경우가 발생한다. 따라서 파이프라인을 혼란시키는 명령어를 수행할 때 오버헤드가 발생하며, 데이터 처리 속도도 증가하지 않는다. 오버헤드를 제거하기 위해 파이프라인 초기 단계에서 그 흐름을 예측해서 예측된 명령어를 다음 단계에 바로 수행할 수 있는 분기 타겟 버퍼를 이용하는 방법 등이 시도되었다[2]. 그러나 리턴 명령어는 서브루틴 호출 명령어의 어드레스에 종속되어 있기 때문에 그 흐름을 예측하기가 어렵다. 즉 서브루틴 어드레스의 정보가 없다면 예측할 수 없음을 의미한다. 특히 객체 지향 프로그램과 같은 응용에서는 다량의 근거리 분기에 따른 많은 리턴 동작이 발생한다. 이를 효과적으로 예측하고, 처리하기 위해서는 리턴 어드레스를 예측하고, 고속의 연산이 가능하도록 하기 위한 파이프라인 도입이 불가피하다.

본 논문은 리턴 어드레스 스택 버퍼를 이용한 분기 예측이 가능하도록 한다. 특히 객체 지향형 프로그램을 지원할 수 있도록 스택 버퍼를 이용한 콘트롤러를 내장하고 있어서 객체에 기반한 데이터 처리에 효과적이며 네트워크를 기반으로 한 객체 지향형 데이터 처리를 하는 프로세서에 적용이 가능한 구조이다[3].

객체 지향 코드로 작성된 프로그램은 C보다 2배 이상의 클래스로 구성되며, 이는 다량의 근거리 분기 동작과 함께 리턴 동작이 발생함을 가리킨다[4]. 또한 일반적인 프로그램에서 분기 동작은 모든 명령어의 약 15%가량을 차지하며, 객체 지향 프로그램의 경우는 많은 클래스의 사용으로 30% 이상의 리턴 동작이 발생할 수 있기 때문에 시스템 성능에 중요한 영향을 미칠 수 있다. 스택 버퍼는 32개의 스택 요소로 구성되는데, 일반적인 프로그램에서 24개를 기준으로 스택 정보의 1%가 오버플로우 되므로 32개의 스택 요소는 오버플로우에 의한 데이터 누출을 방지할 수 있다[1].

본 논문의 구성은 Ⅱ장에서 본 논문에서 제안하는 파이프라인 구조와 RS(Retrun Stack) 구조를 설명하고, Ⅲ장에서 흐름 제어 명령어의 포맷과 명령어 수행 흐름을 살펴본다. Ⅳ장에서는 VHDL 시뮬레이션 결과와 분석을 하며, Ⅴ장에서 결론을 맺는다.

Ⅱ. 본 론

본 논문의 흐름 제어 명령어 처리 구조는 4단의 파이프라인을 가지고 있다. 파이프라인 구조는 데이터 처리 흐름에 따라 계산 시간이 많은 단계를 나누어 각 단계를 동시에 처리함으로써 한 명령어의 평균 처리 시간을 짧게 하며, 전체 성능을 향상시키는 효과가 있다.

본 구조에서는 리턴 어드레스를 전용으로 저장하는 RS를 둔다. 서브루틴 호출 명령어를 수행할 때 리턴 어드레스를 RS에 저장한 후 리턴 명령어를 수행할 때 디코딩 파이프라인 단계에서 RS로부터 예측한 리턴 어드레스를 우선처리(pre-processing)한다.

1. 파이프라인 동작

본 구조의 파이프라인은 명령어를 패취하기 위한 F(Instruction Fetch) 단계, 명령어 디코딩을 수행하는 D(Decode) 단계, 오퍼랜드 어드레스를 계산하는 A(Address generate)단계, 명령어를 수행하는 E(Execute) 단계 등 4단계로 나눈다[5][6]. 이는 본 논문의 명령어 처리 구조는 연산이나 메모리 억세스가 E 단계에서 가능한 프로세서에 적용이 될 예정이며[3], 호출 및 리턴 명령어 또한 4 사이클 내에서 모두 처리가 가능하기 때문이다.

F 단계에서 D 단계로 전달되는 정보는 명령어 코드 그 자체이다. D 단계에서 A 단계로 전달되는 정보는 명령어에 따른 동작 신호와 오퍼랜드 어드레스 계산에 필요한 정보를 전달한다. A 단계에서 E 단계로 전달되는 정보는 프로그램 메모리의 어드레스 및 파라미터 값과 RS에 백업을 위한 신호 등을 전달한다.

F 단계는 외부 프로그램 메모리로부터 하나의 명령어를 패취하여 그 값을 IR(Instruction Register)에 저장한다. 다음에 패취할 명령어 어드레스는 분기나 점프 동작이 일어날 때는 PC 연산 블록에서 새로운 PC값을 연산하여 다음 사이클에 패취를 한다.

D 단계는 F 단계에서 입력받은 명령어 코드를 디코딩한다. 각 명령어 코드는 32비트로 구성한다. 이 단계에서는 각 명령어의 PC 연산 블록 제어, 우선분기 프로세싱 등을 수행한다. 우선분기 프로세싱은 조건부(conditional) 분기 명령어나 무조건(unconditional) 분기 명령어의 리턴 어드레스를 예측한다.

A 단계는 메모리를 접근하는 명령어의 경우 어드레스의 연산 과정이 필요하며, 이때 단일 사이클에 어드레스를 생성한다. 서브루틴 호출 명령어에서는 리턴 어드레스를 연산하여 레지스터에 저장하고, 분기 타겟 명령어를 연산한다. 리턴 명령어에서는 외부 데이터 메모리 내의 스택 영역을 참조하기 위한 어드레스를 연산한다.

E 단계는 명령어 수행을 통해 데이터를 처리하는 단계이다. ALU, 배럴 쉬프터, 곱셈기 등이 사용된다. 메모리를 접근하는 명령어인 경우 단일 사이클 동안 레지스터 값을 메모리에 저장하거나 메모리에서 값을 읽어 레지스터에 저장한다. 분기 명령어는 조건 분기 여부를 판단하고, PC값을 갱신하거나 RS 및 외부 스택 메모리에 리턴 어드레스를 저장한다. RS에 저장할 때는 유효 비트(valid bit)를 세팅하므로써 리턴 명령어 수행시 예측 여부를 판단한다. 또한 리턴 명령어는 유효 비트가 '0'일 경우 RS에 저장되어 있는 리턴 어드레스가 정확하지 않음을 의미하기 때문에 A단계에서 이미 연산한 외부 스택 메모리 어드레스를 참조하여 리턴 어드레스를 가져온다.

2. 흐름 제어 명령어의 우선 처리

리턴 명령어 수행에 따른 파이프라인의 혼란을 막기 위해 서브루틴 리턴 명령어의 D 단계에서 우선분기 처리를 수행한다. 그림 1은 흐름 제어 명령어를 수행하기 위한 구조이다.

SP(Stack Pointer)는 외부 데이터 메모리 내 일부 스택 영역을 참조할 수 있는 리턴 어드레스를 저장하고 있으며, EP(Encoding Pointer)는 6비트 길이로 리턴 명령어 수행시 RS에 저장되어 있는 리턴 어드레스를 출력시키는 역할을 한다. DP(Decoding Pointer)는 EP와 같이 RS내 어드레스를 출력하기 위한 포인터이다. EP는 실행 단계에서 로드가 되며, DP는 우선리턴 동작을 하기 위해 디코딩 단계에서 동작한다.

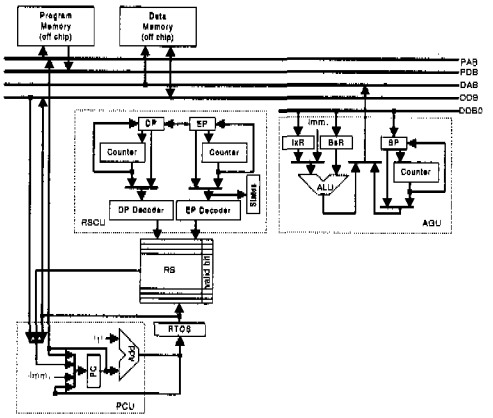


그림 1. 흐름 제어 명령어 처리 블록도

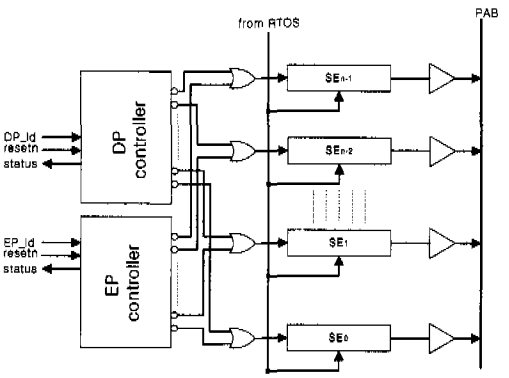


그림 2. RS 구조

RTOS(Return Top OF Stack)는 리턴 어드레스를 RS에 저장하기 전에 일시 저장하고 있는 레지스터이며, 조건 분기 명령어의 조건이 맞지 않았을 때 프로그램 메모리에서 다음 명령어를 패취하기 위한 어드레스로 사용한다. IxR(Index Register), BsR (Base Register)는 외부 데이터 메모리의 어드레스를 발생시키기 위해 필요한 레지스터이다. AGU (Address Generate Unit)는 외부 데이터 메모리 어드레스를 발생시키는 블록이다. PCU(Program Counter Unit)는 PC값을 연산하기 위한 블록이다. PAB(Program Address Bus), PDB(Program Data Bus), DDB(Dada Data Bus), DAB(Data Address Bus) 등은 외부 메모리 및 내부 각 블록과 데이터

를 주고 받는 버스이다. 그림 2는 RS 구조를 보여주고 있다.

PAB는 PC값과 연결되어 있다. 즉 각 SE(Stack Element)는 리턴 어드레스를 저장하고 있기 때문에 명령어에 따라 새로운 PC값이 되는 것이다. SE의 각 로드 신호는 DP, EP 컨트롤러에서 출력하는 디코딩 신호에 따른다. SE 입력은 최초로 리턴 어드레스 값을 저장하고 있는 RTOS로부터 입력된다.

본 논문에서는 CALLcond(conditional CALL), JUMPcond(conditional JUMP), RET(RETurn)를 흐름 제어 명령어로 제공한다. 서브루틴 호출 명령어를 수행할 때 서브루틴으로부터 리턴 어드레스를 연산하여 실행 단계에 RS에 저장한다. 또한 리턴 명령어를 디코딩할 때 RS에 저장한 어드레스로 우선분기 처리를 한다. 따라서 분기 처리를 파이프라인 초기 단계인 디코딩 단계에서 수행함으로써 서브루틴 리턴 명령어 수행으로 초래된 파이프라인 장애를 크게 줄일 수 있다. 실행 단계에서 우선분기 된 어드레스의 유효 비트가 '0'이면 외부 스택 메모리로부터 실제 리턴 어드레스를 읽어들인다.

RS는 프로그램 서브루틴을 수행하기 위해 32개 하드웨어 스택을 갖는다. RS는 LIFO 레지스터 파일이다. 포인터는 6비트 크기를 가지고 있으며, 5비트 LSB(Least Significant Bits)는 SE를 선택하는데 사용하며, 1비트 MSB(Most Significant Bit)는 스택의 오버플로우, 언더플로우을 찾기 위해 사용한다. 표 1은 포인터 동작을 보여주고 있다.

표 1. 스택 포인터 동작.

Resetn	push/pop	pointer(old)	pointer(new)	O/U
1	×	×	111111(empty)	none
0	none	Q	Q	none
0	push	111111(empty)	000000	none
0	push	000000	000001	none
0	push	011111(full)	100000	O
0	pop	011111(full)	011110	none
0	pop	000000	111111(empty)	none
0	pop	111111(empty)	111110	U

리셋 상태에서 포인터는 '111111'으로 초기화한다. 오버플로우와 언더플로우 탐색은 다음과 같은 연산으로 가능하다.

$$O = (pointer[5]='1') \text{ AND } (pointer[4:0]='00000') \quad (1)$$

$$U = (pointer[5]='1') \text{ AND } (pointer[4:0]='11110')$$

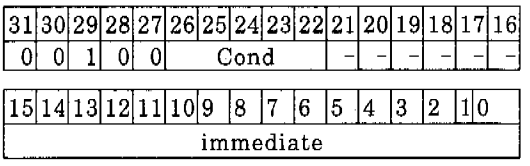
초기화 상태의 스택 포인터는 스택이 빈 상태임을 나타내는 '111111' 값을 가지고 있다. 스택 포인터가 '111111'에서 '111110'로 될 때 스택은 언더플로우가 되며, '011111'에서 '100000'로 되면 스택 오버플로우가 된다.

Ⅲ. 서브루틴 호출, 분기 및 리턴 명령어 동작

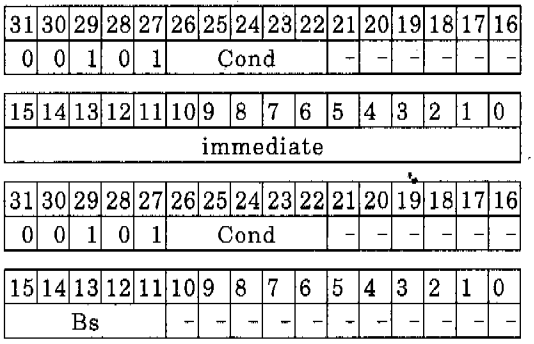
본 논문의 명령어 포맷은 객체 지향 프로그램 처리가 가능한 프로세서에 응용될 수 있도록 32비트 길이를 가지고 있다[5]. 그림 3은 본 구조에서 동작할 수 있는 흐름 제어 명령어 포맷이다.

CALLcond명령어는 직접 모드(immediate mode) 어드레싱만을 지원한다[7]. 31-29는 흐름 제어 명령어 그룹을 나타내는 필드이며, 28-27은 오피코드이다. 26-22는 조건을 나타내며, 표 2는 명령어 조건 니모닉을 보여주고 있다. JUMPcond 명령어는 분기 타겟 어드레스로 직접 모드 및 레지스터 모드로 동작이 가능하다.

니모닉 'O', 'U'는 RS의 오버플로우와 언더플로우를 나타내는 니모닉이다. 그림 4는 CALLcond 명령어 수행의 흐름도를 보여주고 있다.



(a) CALLcond 명령어 포맷



(b) JUMPcond 명령어 포맷

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	1	1	0	Cond					-	-	-	-	-	-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

(c) RET 명령어 포맷

그림 3. 흐름 제어 명령어 포맷.

표 2. 명령어 조건 니모닉.

Code	Mnemonic	Flags	Meaning
00000	AL	Don't care	ALways
00001	LT	N	Less Than
00010	LE	N∨Z	Less than or Equal to
00011	GT	~N∧~Z	Greater Than
00100	GE	~N	Greater than or Equal to
00101	EQ	Z	EQual
00110	NE	~Z	Not Equal to
00111	Z	Z	Zero
01000	NZ	~Z	Not Zero
01001	P	~N∧~Z	Positive
01010	N	N	Negative
01011	NN	~N	Not Negative
01100	NC	~C	Not Carry
01101	C	C	Carry
01110	V	V	oVerflow
01111	NV	~V	Not oVerflow
10000	O	O	stack Overflow
10001	NO	~O	stack Not Overflow
10010	U	U	stack Underflow
10011	NU	~U	stack Not Underflow

CALLcond 명령어는 D 단계에서 분기 타켓 어드레스를 나타내는 상수 값을 디코딩하고, DP를 감소시킨다. 이는 리턴 명령어를 D 단계에서 우선분기 처리할 수 있도록 하기 위함이다. E 단계에서 조건을 만족하면 리턴 어드레스를 RS에 저장한다. 이때 EP도 '1'이 감소하며, 서브루틴이 끝날 때 리턴 명령어가 디코딩되면 DP를 디코딩하여 RS에 저장된 리턴 어드레스를 찾는다.

A 단계에서는 분기 타켓 어드레스를 계산하여 PC에 저장하고, 리턴 어드레스는 PC에 '1'을 더하여 RTOS에 저장한다. 또한 AGU에서 리턴 어드레스 값을 저장할 외부 메모리 어드레스를 계산한다. 분기 타켓 어드레스, 리턴 어드레스와 외부 데이터 메모리의 어드레스를 계산하는 블록은 서로 독립적이기 때문에 동시에 처리가 가능하다.

E 단계에서는 조건을 만족하는지 않는지 판단한다. 만약 조건을 만족하지 않으면 리턴 어드레스 값을 저장하고 있는 RTOS 값을 PC에 저장하여 다음 사이클에 외부 프로그램 메모리로부터 명령어 필드를 패취할 수 있게 한다. 반면에 조건을 만족하면 EP를 '1'만큼 감소시켜 리턴 어드레스를 저장하고 있는 RTOS값을 RS에 저장한다. 그리고 유효 비트를 '1'로 채운다. 또한 분기 타켓 어드레스를 저장하고 있는 PC값은 A단계에서 이미 계산되었으므로 E 단계에서는 새로운 명령어를 패취한다. 또한 RTOS 값을 A 단계에서 계산한 외부 메모리 어드레스에 저장한다.

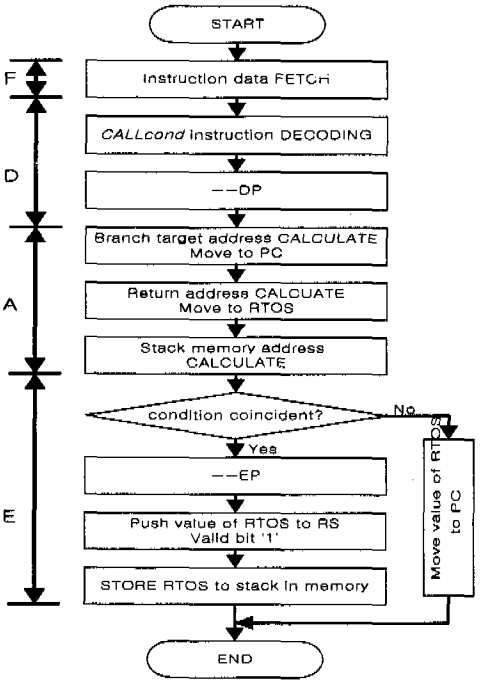


그림 4. CALLcond 명령어 수행 흐름도.

본 구조는 외부 데이터 메모리의 일정영역을 스택으로 사용할 수 있는데 이 메모리는 스택 전용 메모리가 아니기 때문에, 즉 저장 명령어 등에 의해 다른 데이터도 저장이 되므로 프로그래머가 데이타의 충돌이 생기지 않도록 관리해야 한다. 그림 5는 JUMPcond 명령어 수행 흐름도를 보여주고 있다.

JUMPcond 명령어는 CALLcond 명령어와 유사

하나 리턴 어드레스를 RS와 외부 데이터 메모리에 저장하지 않는다. 그러나 조건을 만족하지 않을 경우 리턴 어드레스로 분기 해야 하므로 A 단계에서 분기 타켓 어드레스와 리턴 어드레스를 각각 PC, RTOS에 저장한다. E단계에서는 조건을 조사하여 조건을 만족하면 프로그램 스퀀스를 그대로 진행하고, 조건을 만족하지 않으면 리턴 어드레스를 저장하고 있는 RTOS 값을 PC에 저장한다. 그림 6은 RET 명령어수행 흐름도이다[8].

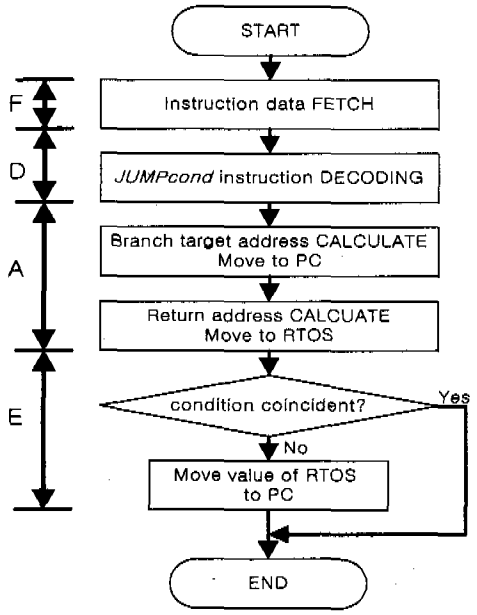


그림 5. JUMPcond 명령어 수행 흐름도

RET 명령어는 서브루틴으로부터 리턴 동작을 수행하며, 스택에 저장되어 있는 리턴 어드레스로 분기하는 명령어이다. D 단계에서는 DP가 가르키는 RS내 값을 PC에 저장한다. 그리고 나서 DP 값을 '1'증가시킨다. 이때의 DP값은 이미 CALLcond 명령어의 D단계 수행시 '1'을 감소시켜 놓았기 때문에 CALLcond의 E단계에서 RS에 저장할 때의 EP 값과 같아서 RS에 저장한 리턴 어드레스 값을 PC에 저장하는 것과 같다.

A 단계에서는 외부 데이터 메모리 어드레스를 구하기 위한 연산을 한다. 이는 RS에 저장되어 있는 리턴 어드레스가 실제 어드레스가 아닌 경우 외부 데이터 메모리에서 가져오기 위하여 외부 데이터 메모리의 어드레스를 발생시키는 것이다.

E단계에서는 RS내에 EP가 가르키는 위치의 값이 유효 비트가 '1'인지를 검사한다. 만약 '1'이면 D단계에서 선리턴했던 PC값이 유효함을 의미하므로 프로그램을 계속 진행시킨다. 그러나 '1'이 아닐 경우 외부 메모리에 저장되어 있는 리턴 어드레스 값을 PC에 저장하여 다음 사이클에 패취하도록 한다. 그리고 나서 EP값을 DP에 복사한다. 그림 7은 흐름제어 명령어의 파이프라인 동작 예를 보여주고 있다.

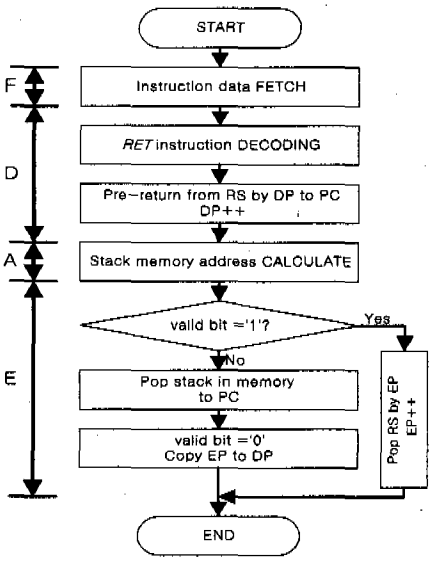


그림 6. RET 명령어 수행 흐름도.

```
10:     CALLNZ 30
11:     ADD
12:     MUL
13:     AND
.
.
.
30:     JUMPNZ 40
31:     CMP
32:     SUB
33:     XOR
.
.
.
40:     RET
41:     AND
```

PC	F	D	A	E
10	CALLNZ	-	-	-
11	ADD	CALLNZ	-	-
11	(nop)	(nop)	CALLNZ	-
30	JUMPNZ	(nop)	(nop)	CALLNZ
31	CMP	JUMPNZ	(nop)	(nop)
31	(nop)	(nop)	JUMPNZ	(nop)
40	RET	(nop)	(nop)	JUMPNZ
41	AND	RET	(nop)	(nop)
11	ADD	(nop)	RET	(nop)
12	MUL	ADD	(nop)	RET

그림 7. 흐름 제어 명령어 파이프라인 동작 예.

그림 7은 분기 명령어 모두가 조건이 일치하여 분기하는 것으로 전제하였다. 분기 명령어임을 D 단계에서 디코딩하고 나면 새로운 PC값이 입력될 때까지 명령어를 수행하지 않는다. 물론 분기 조건에 맞지 않아 분기가 일어나지 않을 경우는 모든 분기 명령어의 E 단계가 끝나고, 다음 사이클에 리턴 어드레스에 저장되어 있는 명령어를 수행한다.

Ⅳ. VHDL 시뮬레이션 및 분석

본 논문의 프로그램 흐름 제어 명령어 처리 구조를 검증하기 위하여 각 모듈 블록을 VHDL을 이용하여 구조적 수준 및 행위적 수준에서 기술한 후 각 모듈을 Model Technology Inc.의 V-System PC 버전 4.4를 이용하여 VHDL 시뮬레이션하였다. 사용한 라이브러리는 V-System에서 기본으로 제공하는 IEEE 라이브러리만을 사용하였다[9]. 시뮬레이션을 위한 VHDL 코드는 5개의 컴포넌트(RS, RSCU, AGU, PCU, RTOS)로 구성되어 있다. 시뮬레이션을 위한 입력 값들은 그림 7의 CALLNZ, JUMPNZ, RET 명령어를 사용한 파이프라인 동작 예를 검증하기 위해 발생할 수 있는 경우를 고려하여 처리하였다. 그림 8은 시뮬레이션 결과를 보여주고 있다.

in1 신호는 DDB0값이며, in2는 직접 모드 값이다. io는 DDB값이며, out1 신호는 DAB 값이다. fc_o, fc_u 신호는 각각 RS의 오버플로우와 언더플로우를 나타내는 신호이다. 클럭 신호인 clk는 100ns를 주기로 발생한다. /u4/u2/pc_out 신호는 PC 결과 값이다.

시뮬레이션 결과 값이 그림 7.의 경우와 같음을 확인할 수 있다. 또한 CALLNZ 명령어 수행시 리턴 어드레스 였던 '11'을 RTOS 결과 값인 t_rtos에 우선 저장하고(A), E 단계에 RS에 저장한다(B). 물론 이때 valid 신호(u0/u35/se_out)는 '1'이 된다. RET 명령어를 패취하면 RS에 저장하고 있던 리턴 어드레스를 RET 명령어의 D단계가 끝나면 다음 패취 어드레스로 사용한다(C). JUMPNZ 명령어는 리턴 어드레스가 필요없기 때문에 RS에 리턴 어드레

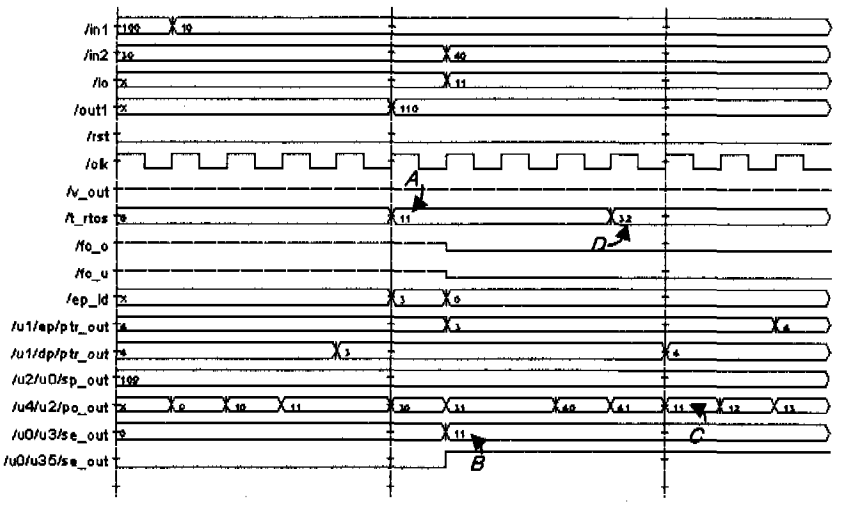


그림 8. 흐름 제어 명령어 처리 구조의 VHDL 시뮬레이션 결과.

스를 저장할 필요 없다. 따라서 두 번째 리턴 어드레스 '32'(D)는 RTOS에만 저장하고 RS에는 저장하지 않는다. 파이프라인의 모든 구간을 지나쳐서 수행되는 전체동작을 태스크(task) 라고 정의하면, 클록 사이클 시간이 tp인 K개의 파이프라인에서 n개의 태스크를 수행할 경우 첫 번째 태스크는 동작을 완료하기 위해 K×tp만큼의 시간이 필요하고, 나머지 (n-1) 태스크중 호출 명령어를 수행하는 태스크가 L개라고 할 경우 총 파이프라인 처리 시간, S_0는 (2)와 같다.

$$S_0 = K \times tp + L \times K \times tp + (n-1-L) \times tp \quad (2)$$

태스크 수가 증가하면 n은 K에 비해 값이 크므로 K+n-1은 n에 근사할 수 있다. K가 4이면, S_0은 (3×L + n)×tp이다. 그러나 본 논문에서는 리턴 명령어에 대해 2단의 파이프라인 단계를 절약할 수 있으므로 (2)의 두 번째 항에서 L은 분기 태스크의 개수로 리턴 동작은 이의 50%이므로, 리턴 태스크 항인 3항은 K대신에 K-2가 되어 총 처리 시간, S_1은 (3)과 같다.

$$S_1 = K \times tp + \frac{L}{2} \times K \times tp + \frac{L}{2} \times (K-2) \times tp + (n-1-L) \times tp \quad (3)$$

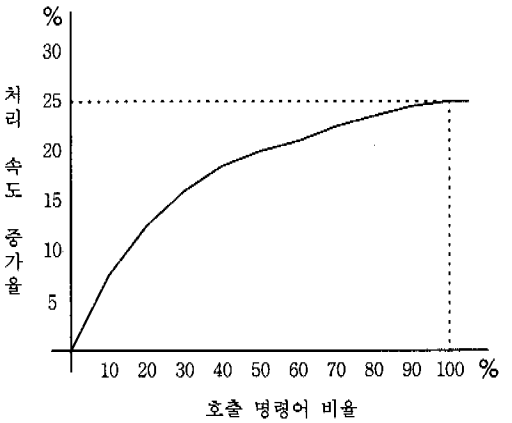


그림 9. 호출 명령어 비율에 따른 태스크 처리 속도 증가율.

K가 4라면 S_1는 (L+n)×tp와 근사하다. 따라서 호출 명령어가 차지하는 비율에 따른 S_0 대비 S_1의 처리 속도 증가율은 그림 9와 같다. 이론적으로는 최대 속도 증가율은 25%이다. 그러나 실제 객체 지향 프로그램에서 분기 동작이 차지하는 비율은 30% 정도 이므로, 최대 증가율은 15%이상이다.

V. 결 론

본 논문은 흐름 제어 명령어 수행의 오버헤드를 제거하기 위해 파이프라인 초기 단계에서 그 흐름을 예측해서 예측된 명령어를 다음 단계에 바로 수행하게 함으로써 명령어 수행 사이클 수를 줄일 수 있는 흐름 제어 명령어 처리 구조를 제안하고, Model Technology Inc.의 V-System을 이용하여 시뮬레이션을 하였다.

호출 명령어의 분기 타겟 어드레스를 예측하기 위한 알고리즘은 많이 연구되고 있으나 리턴 명령어 어드레스는 서브루틴 호출 명령어의 어드레스에 종속되어 있기 때문에 그 흐름을 예측하기가 어렵다. 즉 서브루틴 어드레스의 정보가 없다면 예측할 수 없음을 의미한다. 따라서 본 논문에서는 서브루틴 리턴 명령어를 처리할 때 초기 파이프라인 단계에서 리턴 어드레스 처리가 가능한 구조를 제안하고, 검증하였다. 특히 객체 지향 프로그램을 처리하는 프로세서에서 다량의 근거리 분기 동작에 따른 리턴 어드레스 예측 구조에 적용이 될 수 있으며, 흐름 제어 명령어 처리 블록을 별도로 가지고 있지 않는 일반적인 파이프라인 구조에서 보다 최고 30% 정도의 분기 관련 명령어 수행 사이클 수를 줄일 수 있다. 앞으로 리턴 명령어 이외에 서브루틴 호출 명령어 및 분기 명령어를 우선 예측 하기 위한 구조를 추가하여 객체 지향형 프로세서에 응용할 계획이다.

참 고 문 헌

[1] P. J. Koopman, Jr., "Stack Computers: the new wave,"http://www.cs.cmu.edu/~koopman/stack_computers/, 1989.

[2] J. K. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE COMPUTER* Vol. 17, No. 1, pp. 6-22, Jan. 1984.

[3] 박주현, 김영민, "네트워크 기반 객체 지향형 영상 처리를 위한 MPEG 디코더 코어 설계," *한국통신 학회논문지* 제 23권 제 8호, pp. 2120-2128, 1998.

[4] A. Forrest and D. R. McGregor, "Video in Java: Virtual Statistics of a Time-Critical Application,"

http://www.sct.cs.strath.ac.uk/papers/java-implement-ation.html.

[5] *TMS320C4x User's Guide*, DSP Products, 1993.

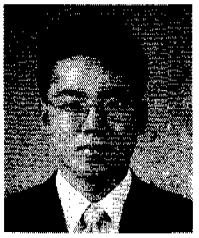
[6] *The ARM RISC Chip: A Programmer's Guide*, 1993.

[7] IBM, "AIX Version 4.3 Assembler Language Reference," http://www.rs6000.ibm.com/doc_link/en_US/a_doc_lib/aixassem/alangref/branch.htm#A206P134, Oct. 1997.

[8] IBM Corp., Armon, N.Y., "Instruction address stack in the data memory of an instruction-pipelined processor," United States Patent, Appl. no.: 280,417, 1983.

[9] *V-System/VHDL PC User's Manual Version 4.4*, Model Technology, 1996.

박 주 현(Ju-Hyun Park) 정회원



1993년 2월 : 전남대학교 전자공학과 졸업

1995년 2월 : 전남대학교 전자공학과 석사

1999년 2월 : 전남대학교 전자공학과 박사

1999년 2월~현재 : 한국전자통신연구원 집적회로설계 연구부 선임연구원

<주관심 분야> MPEG-4 시스템설계, RISC, DSP설계

김 영 민(Young-Min Kim) 정회원

1976년 2월 : 서울대학교 전자공학과 졸업

1978년 2월 : 한국과학원 전기 및 전자공학과 석사

1986년 : 미국 오하이오 주립대학교 전기공학과 박사

1991년 9월~현재 : 전남대학교 전자공학과 교수

<주관심 분야> 영상 시스템 설계, ADSL 모뎀 설계