4. ARM 조직과 구현

– 전체 요약

The organization of the ARM integer processor core changed very little from the first 3 micron devices developed at Acorn Computers between 1983 and 1985 to the ARM6 and ARM7 developed by ARM Limited between 1990 and 1995. The 3-stage pipeline used by these processors was steadily tightened up, and CMOS process technology reduced in feature size by almost an order of magnitude over this period, so the performance of the cores improved dramatically, but the basic principles of operation remained largely the same.

Since 1995 several new ARM cores have been introduced which deliver significantly higher performance through the use of 5-stage pipelines and separate instruction and data memories (usually in the form of separate caches which are connected to a shared instruction and data main memory system).

This chapter includes descriptions of the internal structures of these two basic styles of processor core and covers the general principles of operation of the 3-stage and 5-stage pipelines and a number of implementation details. Details on particular cores are presented in Chapter 9.

4.1 3단계 파이프라인 ARM 조직

레지스터 뱅크 : 레지스터 접근을 위한 2개의 입력 포트, 1개의 출력 포트, 프로그램 카운터를 위한 1개의 입출력 포트

배럴 쉬프터 : ALU 입력 전 자리 이동과 회전 수행

ALU : 산술 연산, 논리 연산 실행

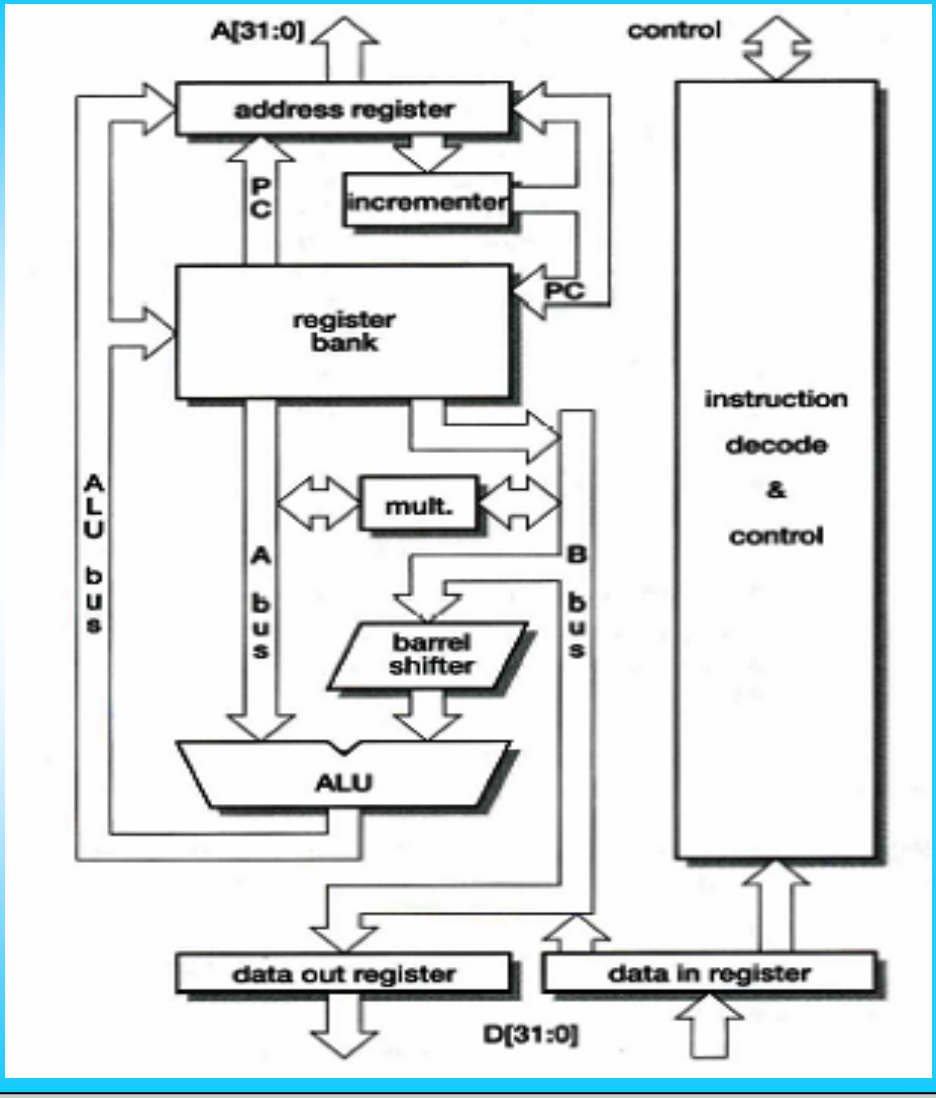
어드레스 레지스터 : 메모리 주소 저장

어드레스 증가기 : 연속적인 메모리 주소 생성

데이터 레지스터 : 메모리로부터 읽은 데이터나 저장할 데이터를 저장

명령어 디코더 : 명령어 해석 및 분석

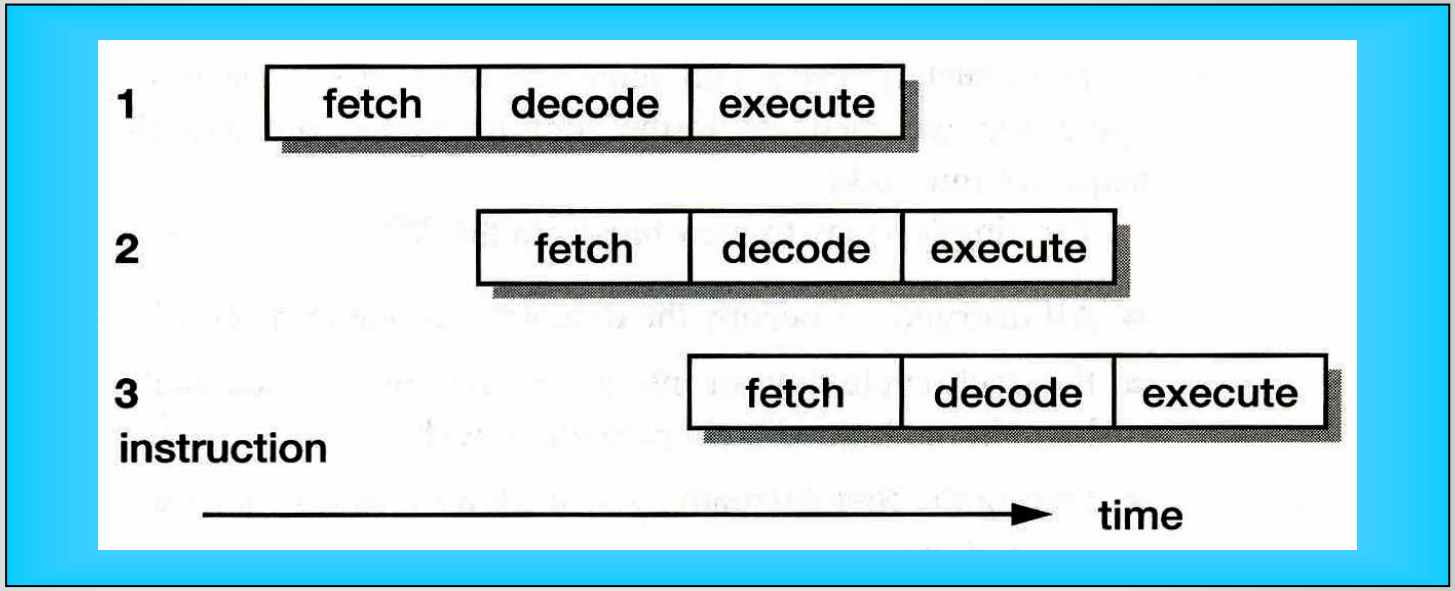
제어 로직 : 명령어 실행을 위해 필요한 제어 신호 생성



- 3단계 파이프라인 동작

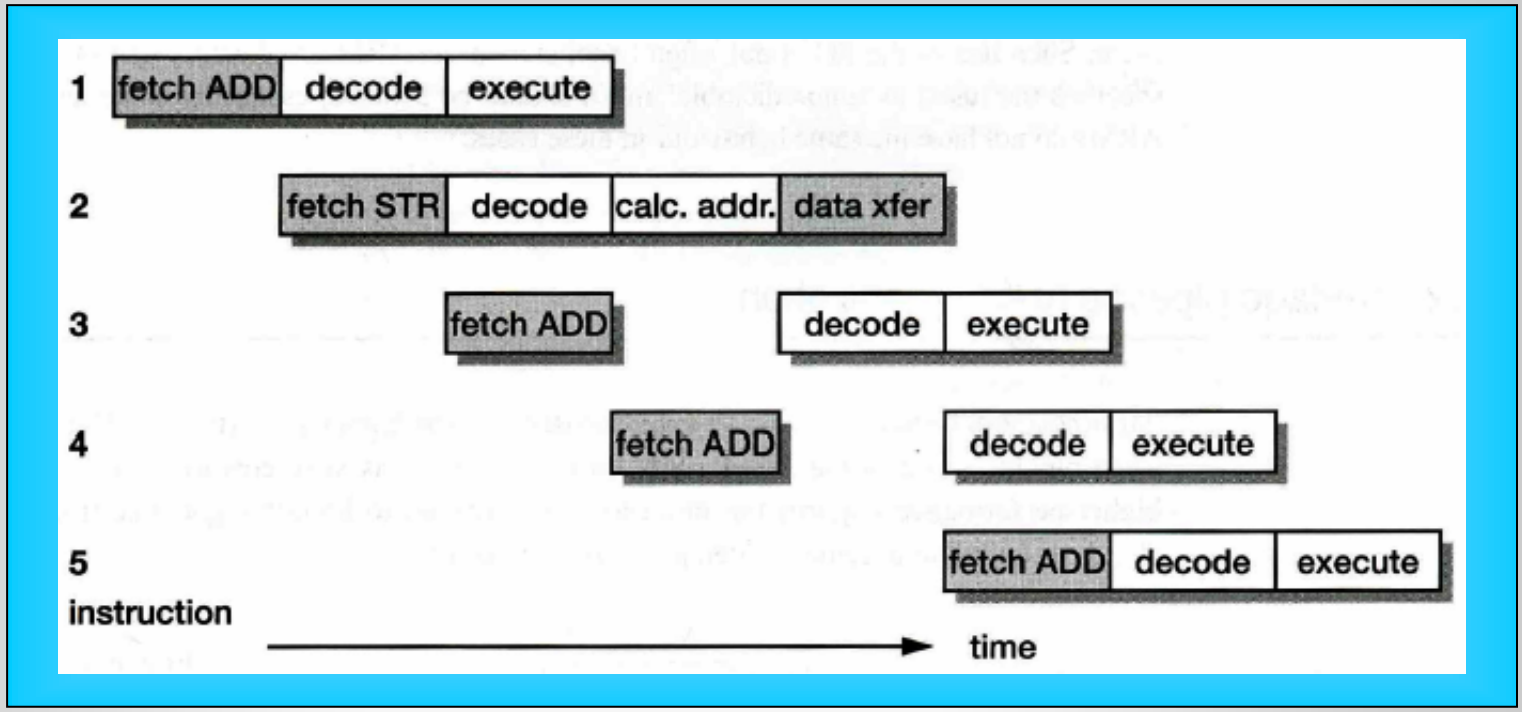
▷ ARM7까지 Fetch, Decode, Execute (Datapath)의 3단계 파이프라인 사용

▷ 싱글 사이클 명령어의 파이프라인 동작 : 각 명령어 실행을 위해 3 사이클이 필요 (3-cycle latency), 처리량은 1 사이클당 1 명령어



▷ 1 사이클에 메모리를 한번 밖에 접근할 수 없기 때문에 STR과 같이 메모리 접근이 필요한 명령어는 멀티 사이클 명령어가 됨

▷ 멀티 사이클 명령어의 3단계 파이프라인 동작



2번째 STR 명령어의 어드레스 계산 사이클 동안에 제어 로직은 데이터 전달을 위한 제어 신호를 생성하여야 하므로 3번째 명령어는 1 사이클 지연, 2번째 명령어가 데이터를 위해 메모리 접근을 하므로 5번째 명령어는 1 사이클 지연 (메모리 접근에 비례하여 소요 사이클이 결정)

▷ ARM 파이프라인 중단이 발생하는 경우

- All instructions occupy the datapath for one or more adjacent cycles.
- For each cycle that an instruction occupies the datapath, it occupies the decode logic in the immediately preceding cycle.
- During the first datapath cycle each instruction issues a fetch for the next instruction but one.
- Branch instructions flush and refill the instruction pipeline.

– PC 동작

▷ r15를 첫번째 이외의 단계에서 사용하게 되면 첫번째 단계에서 r15는 증가하기 때문에 잘못된 결과를 초래할 수 있음

▷ r15 (PC)을 통해 이동할 주소를 계산하는 프로그래머는 파이프라인의 동작을 정확하게 파악하여야 함 (STR의 첫 실행사이클에서 r15는 PC+8이 읽혀지고 두번째 실행 사이클에서는 PC+12이 읽혀짐)

▷ 어셈블러나 컴파일러가 변위나 offset를 계산하도록 하는 것이 좋음

4.2 5단계 파이프라인 ARM 조직

- 3단계 파이프라인보다 높은 성능을 얻기 위해 5단계 파이프라인으로 조직 변경
- 프로그램의 명령어 수가 N_{inst}, 명령어당 평균 사이클 수가 CPI, 프로세서의 클럭 주파수가 f_{clk}일 때 프로그램 실행 시간은 $T_{prg} = (N_{inst} \times CPI) / f_{clk}$ 임
- 프로그램 실행시간을 줄이는 방법
  - ▷ 프로세서의 클럭 주파수 증가 : 파이프라인 단계를 간단하게 하여야 하고 이로 인해 파이프라인 단계가 증가함
  - ▷ CPI 감소 : 3단계 파이프라인에서 멀티 사이클 명령어를 파이프라인 개선으로 싱글 사이클 실행으로 만들거나 파이프라인 지연을 최대한 줄임
- 메모리 bottleneck
  - ▷ 3단계 파이프라인은 모든 사이클에서 메모리를 접근하므로 부분적인 성능 개선이 쉽지 않음 (메모리 접근을 하지 않는 단계를 추가할 필요)
  - ▷ 한번에 전달하는 데이터량을 증가하거나 명령어와 데이터 메모리를 분리
  - ▷ 더 높은 성능을 위해 5단계 파이프라인 도입과 메모리를 분리 (ARM9TDMI)

– 5단계 파이프라인

Fetch : 명령어 메모리에서 명령어를 fetch함

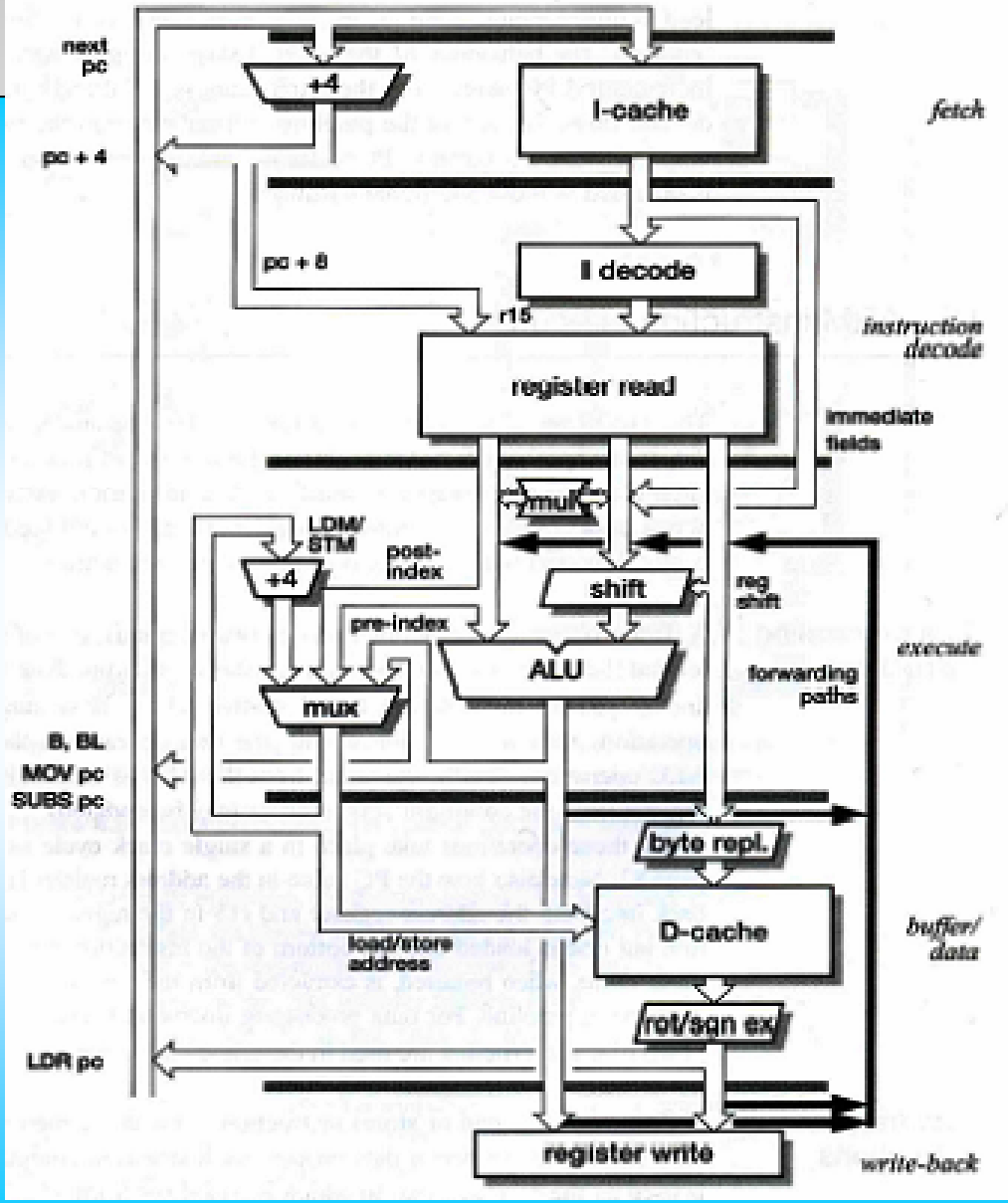
Decode : 명령어를 해석하고 레지스터 오퍼랜드를 읽음

Execute : 오퍼랜드가 shift되고 ALU 연산이 실행, 명령어가 load, store이면 ALU에서 메모리 주소를 계산함

다중 데이터 이동 명령어를 위한 주소 증가 하드웨어 추가

Buffer/data : 데이터 메모리 접근하고 ALU 결과를 임시로 buffer에 저장

Write-back : ALU 결과와 메모리 데이터를 레지스터에 저장



- 데이터 전방전달 (forwarding)

  ▷ 파이프라인이 5단계로 늘어나면 명령어 실행이 1사이클에서 3사이클로 증가하므로 데이터 해저드가 발생

  ▷ 데이터 해저드에 의한 파이프라인 지연을 없애기 위해 전방전달을 위한 하드웨어 추가

  ▷ 결과가 레지스터에 저장되기 이전에 다음 명령어의 소스 오퍼랜드를 위해 명령어 실행 흐름의 반대 방향으로 데이터를 전달

  ▷ 앞 명령어가 적재 명령어이고 다음 명령어에서 데이터 해저드가 발생하면 1 사이클 지연이 필요 (메모리 데이터는 buffer/data 단계에서 이용할 수 있으므로 ADD 명령어는 1 사이클 지연 필요)

```
LDR     rN, [..]          ; load rN from somewhere
ADD     r2, r1, rN        ; and use it immediately
```

▷ 파이프라인 지연을 없애기 위해 위의 형태의 코드를 작성하지 말아야 하고 컴파일러도 위의 형태의 코드를 생성하지 않아야 함

- PC 생성

▷ 1번째 단계에서 PC는 PC+4로 증가하고 decode 단계의 r15에 저장

▷ Decode 단계에서 레지스터 r15를 읽으면 다음 명령어의 PC+4, 즉 현재 명령어의 PC+8이 읽혀짐, 하드웨어 추가 없이 3단계 파이프라인과 동일

4.3 ARM 명령어 실행

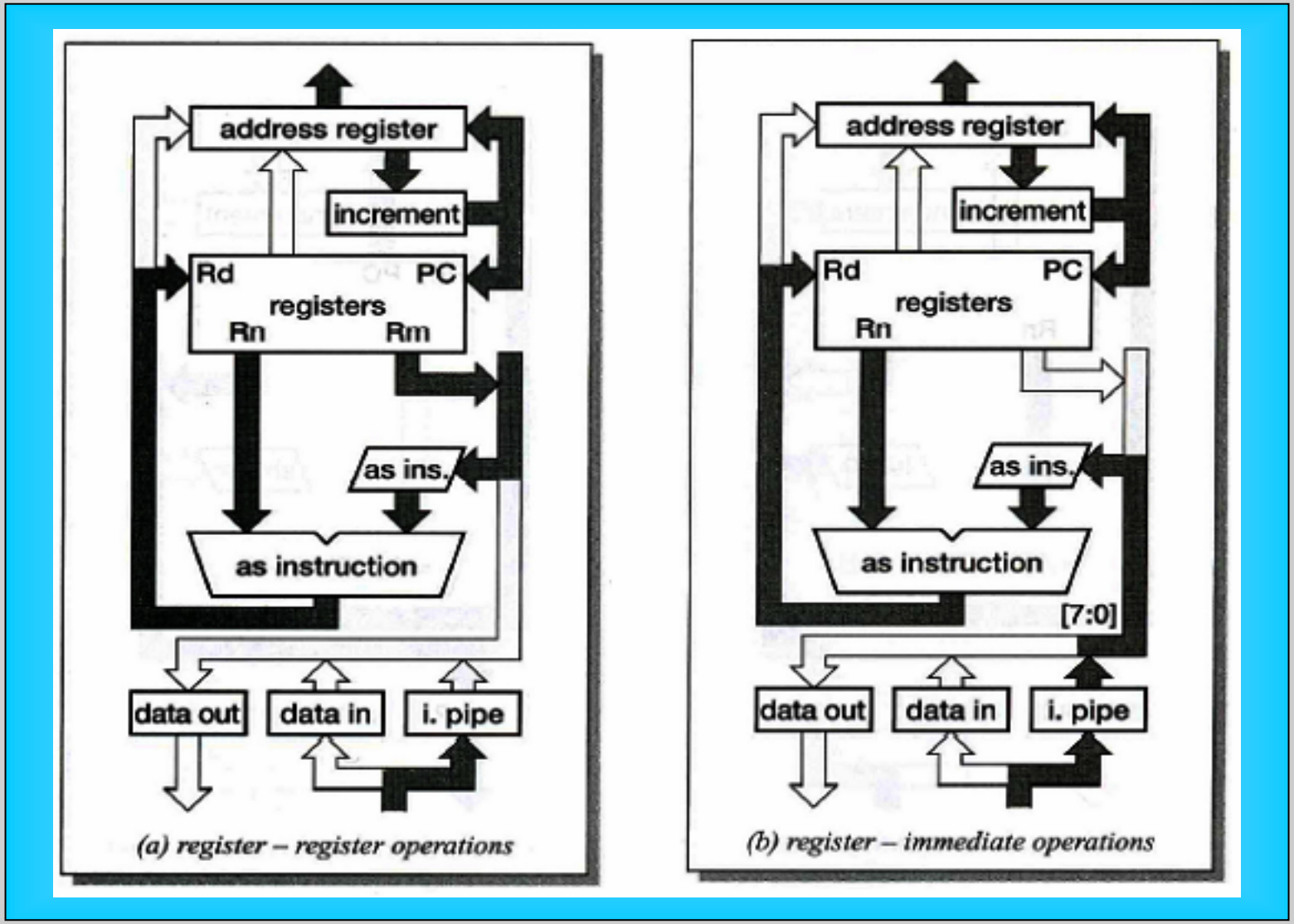
- 각 명령어가 데이터패스에서 어떻게 실행되는지를 파악하는 것이 필요

▷ 데이터 처리 명령어는 하나의 실행 사이클이 필요하지만 저장 명령어 (STR)

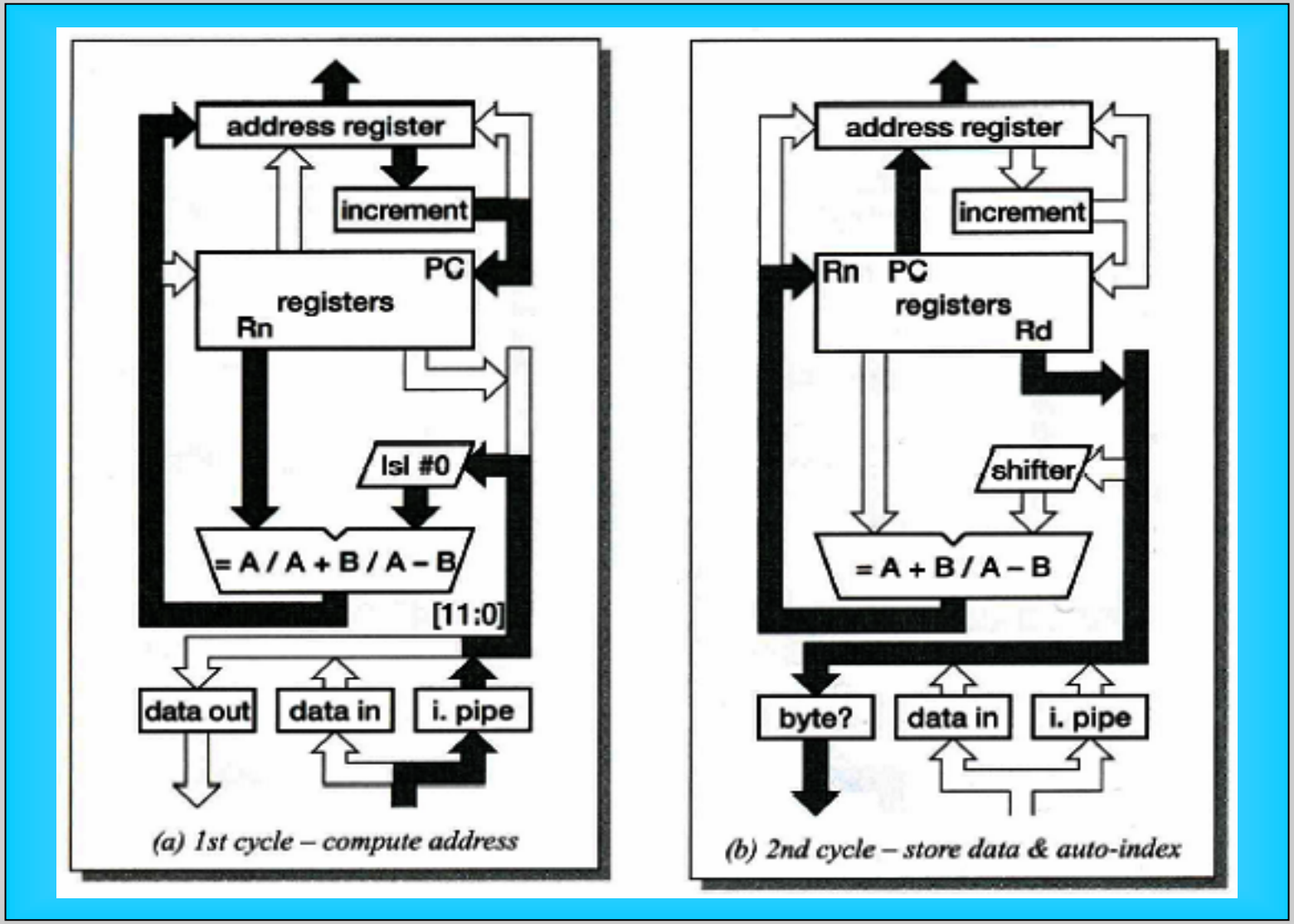
은 2 사이클 실행이 필요하고 적재 명령어 (LDR)은 3 사이클 실행 필요

▷ Branch 명령어는 3 사이클 실행이 필요한데 첫번째 사이클에서 branch 주소를 계산하고 두번째 사이클에서 필요하면 복귀 주소를 r14에 저장하고 마지막 사이클에서 r14를 PC+8에서 PC+4로 조정하면서 refilling을 완료

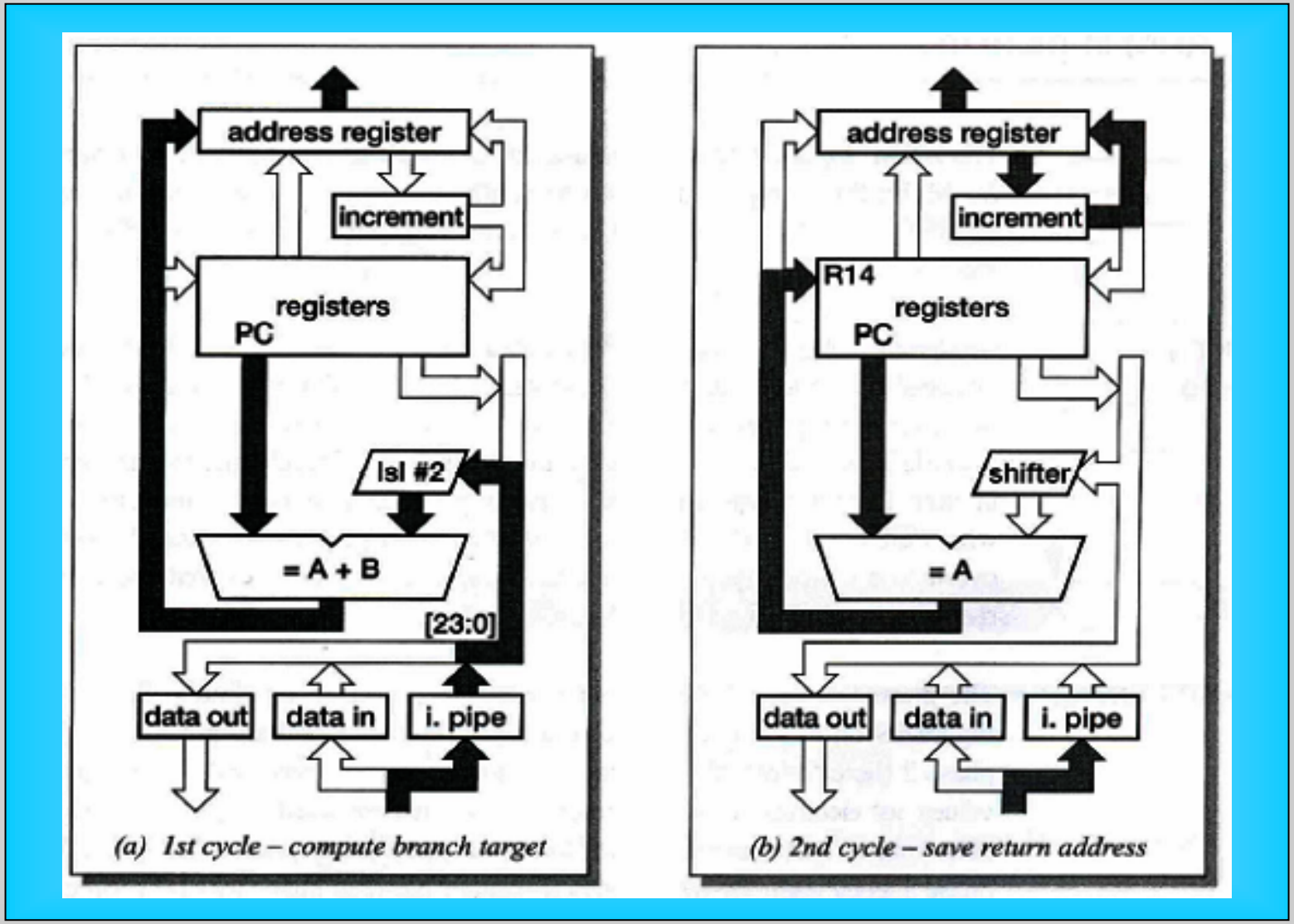
- 데이터 처리 명령어 실행 (3단계 파이프라인)



- 데이터 이동 명령어 STR 실행 (3단계 파이프라인)



- 분기 명령어 실행 (3단계 파이프라인)



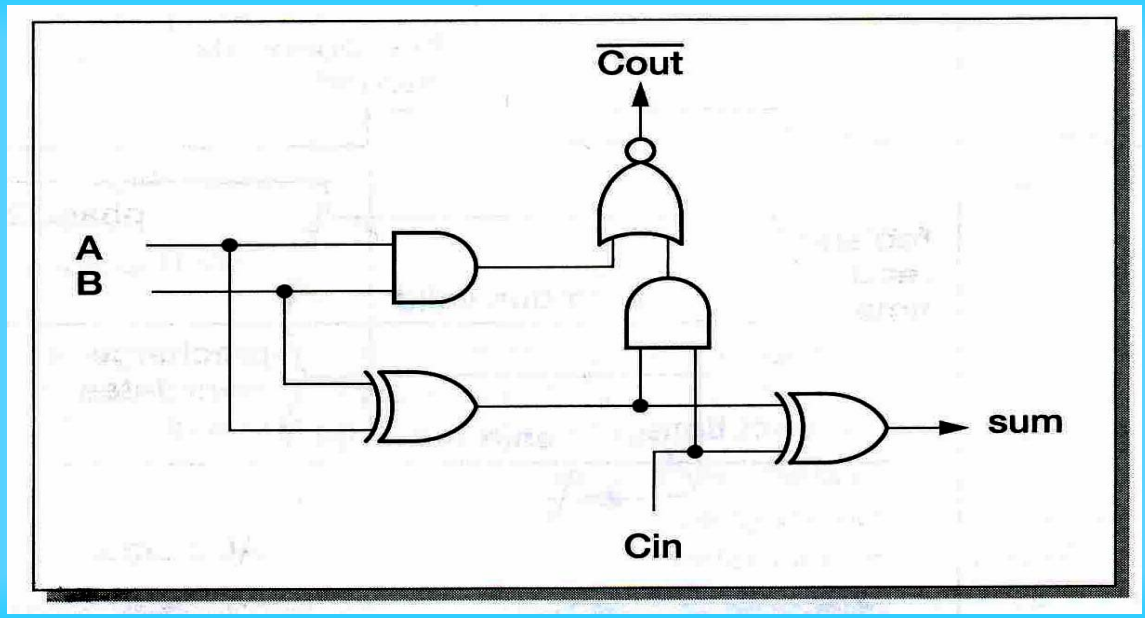
4.4 ARM 구현

- Adder 설계

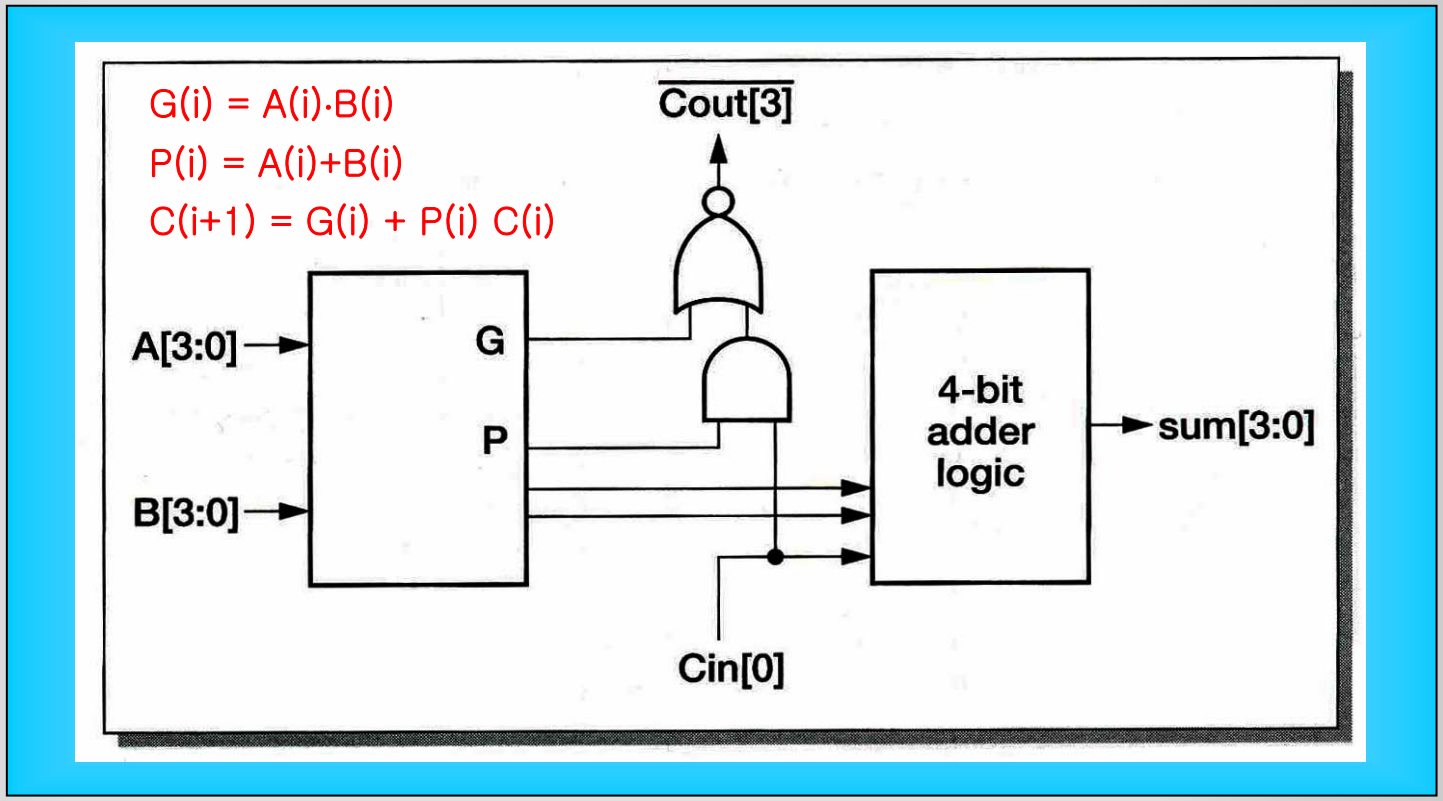
▷ 32 비트 덧셈 시간은 데이터패스의 사이클 시간, 최대 클럭 주파수, 프로세서 성능에 많은 영향을 줌

▷ ARM은 연속되는 버전에서 adder의 성능을 연속적으로 개선

▷ ARM1 ripple-carry adder (RCA) : Carry 전달을 위해 32 게이트 지연 필요

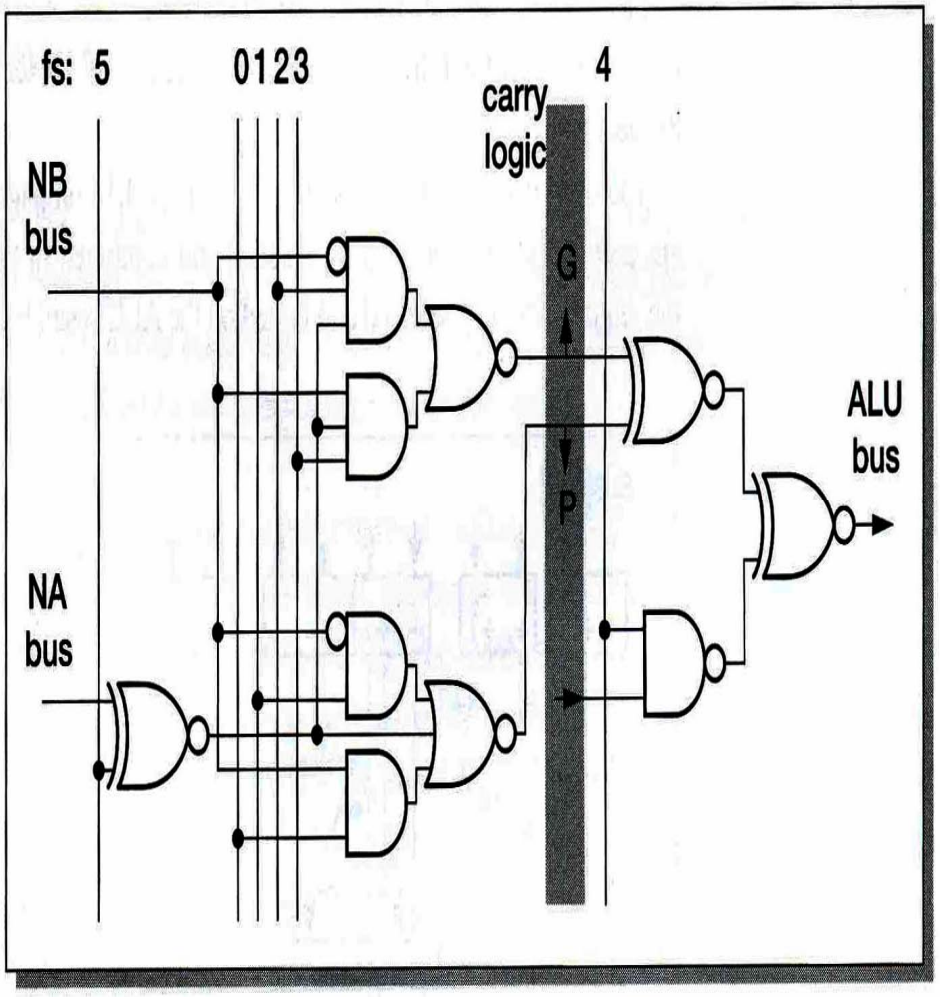


▷ 클럭 주파수를 증가시키기 위해 ARM2는 4 비트 carry look-ahead (CLA) 사용 (Carry 전달을 위해 8 게이트 지연 필요, ARM1보다 성능 개선)



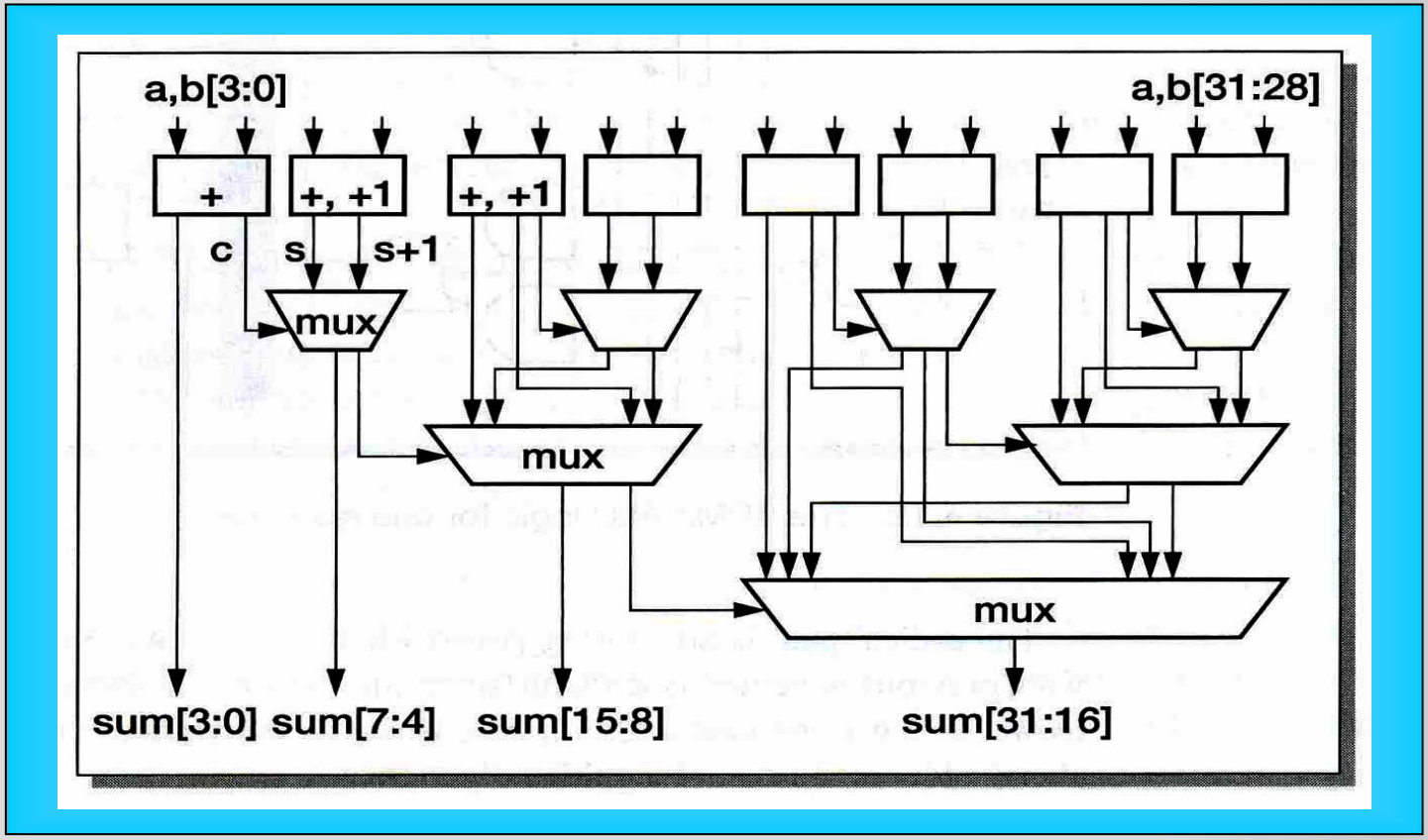
▷ ALU는 덧셈 뿐만 아니라 논리 연산, 메모리 주소 계산, branch 여부 계산 등을 수행하여야 함

▷ ARM2 ALU 로직과 기능 코드



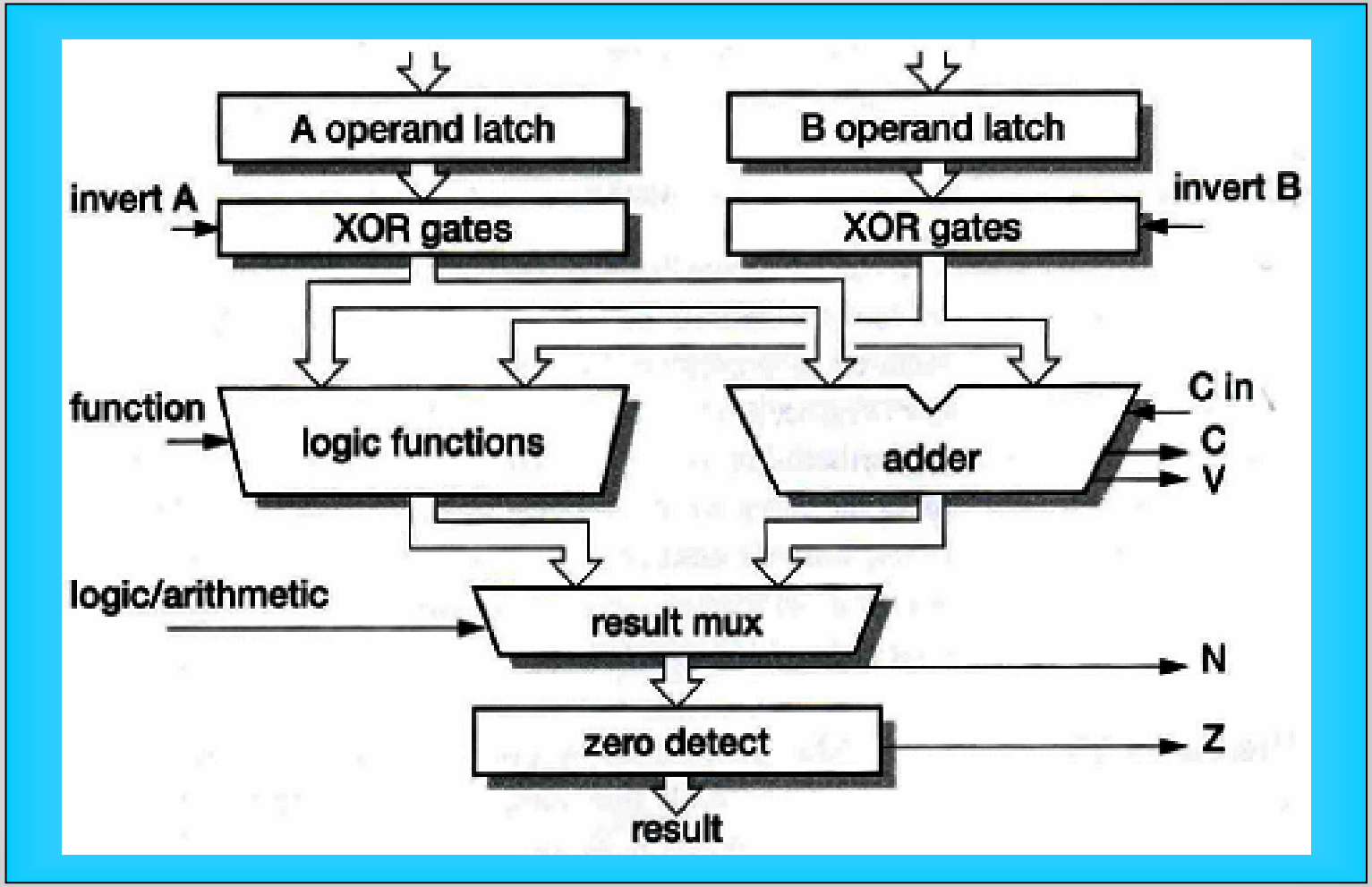
fs5	fs4	fs3	fs2	fs1	fs0	ALU output
0	0	0	1	0	0	A and B
0	0	1	0	0	0	A and not B
0	0	1	0	0	1	A xor B
0	1	1	0	0	1	A plus not B plus carry
0	1	0	1	1	0	A plus B plus carry
1	1	0	1	1	0	not A plus B plus carry
0	0	0	0	0	0	A
0	0	0	0	0	1	A or B
0	0	0	1	0	1	B
0	0	1	0	1	0	not B
0	0	1	1	0	0	zero

▷ 좀 더 나은 성능을 위해 ARM6는 carry-select adder를 사용 (carry 입력이 0이거나 1 모두 sum을 계산한 후 정확한 carry 입력에 의해 결과를 선택)



▷ 덧셈 출력을 위해 O[$\log_2$[word width]) 지연으로 덧셈 시간은 감소하지만 실리콘 면적이 증가함

▷ ARM6 carry-select adder는 산술 기능과 논리 기능을 위해 하나의 형태로 통합할 수 없어 ARM6의 ALU는 adder와 논리 장치가 분리되어 있음



▷ C와 V 플래그는 adder에서 발생, N 플래그는 결과의 부호 비트, Z 플래그는 결과에 32 비트 NOR 게이트 연결

▷ ARM9TDMI는 더 나은 성능을 위해 carry arbitration adder를 사용하는데 매우 빠른 병렬 논리 구조를 사용하여 모든 중간 carry값을 계산

▷ CLA의 propagate-generate 정보를 u, v의 변수로 인코딩하고 합의 모든 비트에서 u, v를 계산

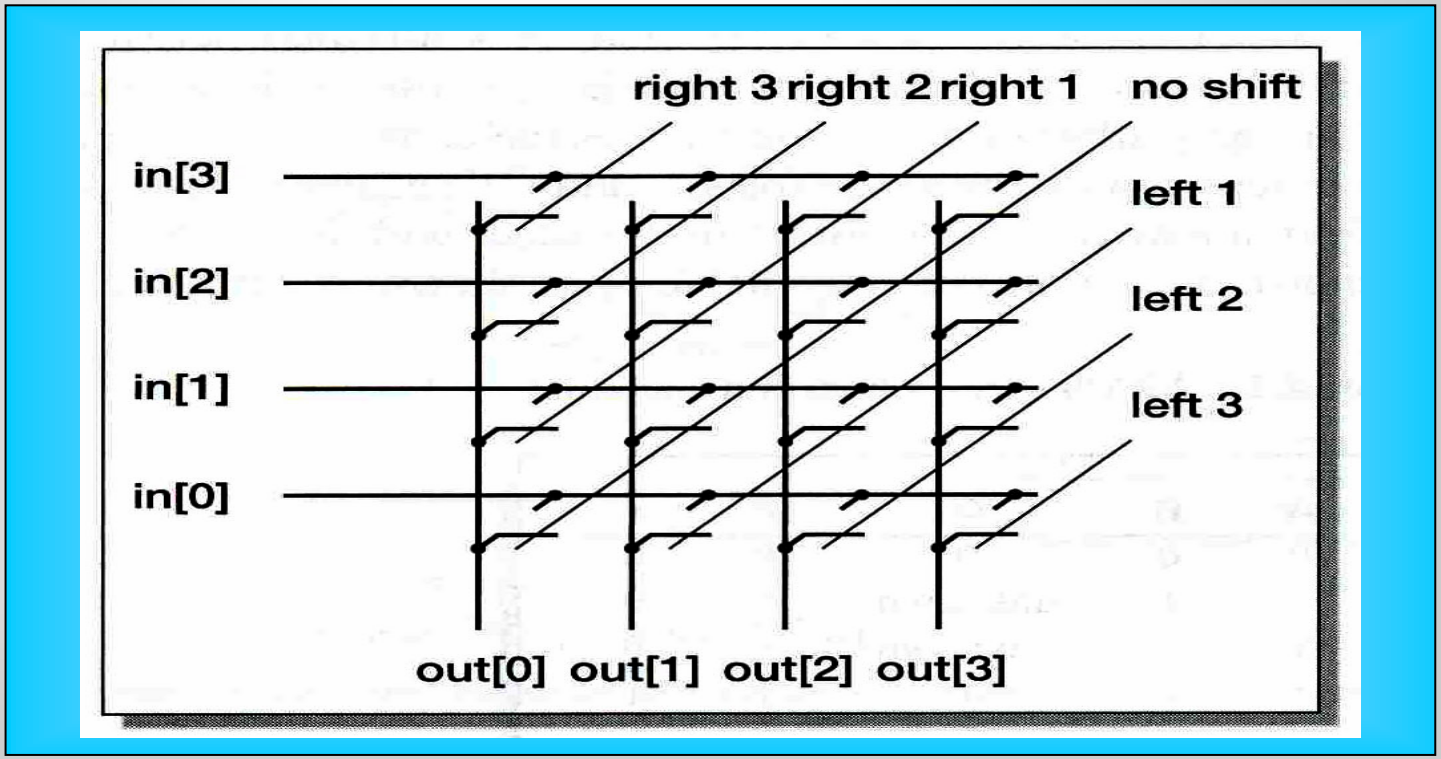
A	B	C	u	v
0	0	0	0	0
0	1	unknown	1	0
1	0	unknown	1	0
1	1	1	1	1

▷ Carry 입력이 1이면 u가 carry 출력이 되고 carry 입력이 0이면 v가 carry 출력이 됨

- Barrel Shifter

▷ ALU 동작과 직렬로 자리 이동 동작을 수행하므로 barrel shifter의 shifter 시간은 데이터 패스 사이클 시간에 직접적인 영향을 줌

▷ Shifter 지연을 최소화하기 위해 ARM은 32 × 32의 cross-bar 스위치 행렬을 입력에 대한 출력을 조정하기 위해 사용



▷ Left shift와 right shift를 위해 하나의 대각 wire가 On이 됨

▷ Rotate right를 위해 right shift를 위한 대각 wire와 complementary left shift를 위한 대각 wire가 On이 됨 (4 × 4 cross-bar 스위치 행렬에서 1 비트 right shift는 ‘right 1’과 ‘left 3’ (3=4-1) 대각 wire를 On)

▷ Arithmetic shift right는 부호 확장을 위한 독립적인 로직을 사용

- 곱셈기 설계

▷ 정수형 곱셈을 위해 하드웨어를 지원

▷ 초기 ARM Core들은 단지 32 비트 결과를 저장하는 곱셈과 곱셈-적재 명령어를 위한 저비용 곱셈 하드웨어를 제공

▷ 최근 ARM Core들은 64 비트 결과를 저장하는 높은 성능의 하드웨어 제공

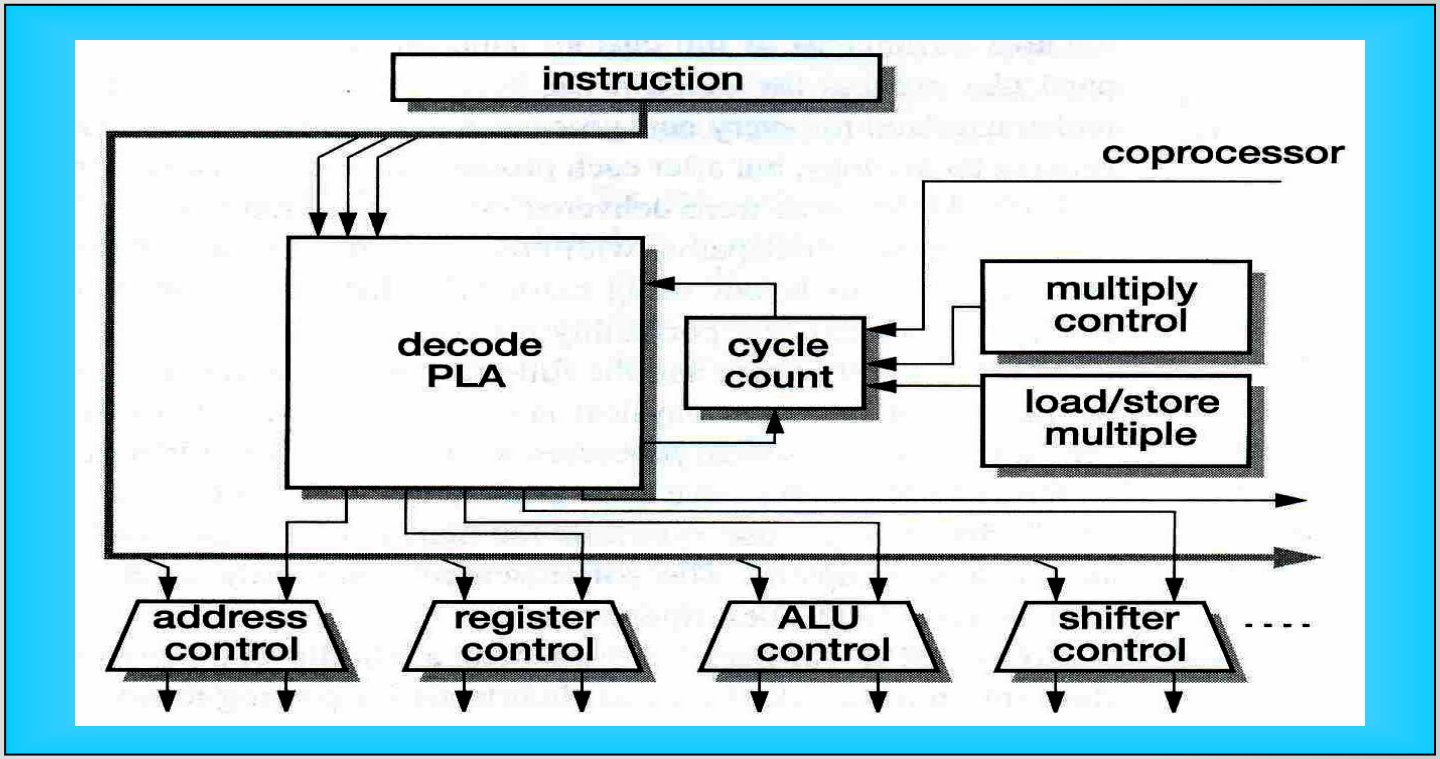
▷ 정수형 곱셈기는 barrel shifter, ALU, 승수 shifter 레지스터, 곱셈 알고리즘 제어 로직 등에 의해 간단하게 구현 가능

▷ 디지털 신호 처리를 위해 곱셈기의 성능은 매우 중요, DSP 코어는 특히 곱셈기의 성능을 매우 강화하여 디지털 연산 시간을 효율적으로 줄임

- 제어 로직 구조

▷ PLA는 명령어의 opcode와 상태에 따라 데이터패스의 각 제어 신호를 출력

▷ PLA는 약 14개 입력, 40개 출력을 가지고 cycle count는 finite state 기기



▷ 최근 ARM Core는 두개의 PLA를 사용 (시간이 중요한 제어 신호는 작고 빠른 PLA를 사용하고 다른 일반 제어 신호는 크고 느린 PLA를 사용)

4.5 ARM coprocessor interface

- 하드웨어 coprocessor의 추가로 인한 명령어의 확장을 지원
- Coprocessor 구조
  - ▷ 16개까지의 coprocessor를 지원하고 각 coprocessor는 합당한 크기의 16개까지의 레지스터를 가질 수 있음
  - ▷ Coprocessor도 load-store 구조를 가짐 (내부 레지스터 사이의 동작 실행, 외부 메모리와 load, store 명령, ARM 레지스터와의 데이터 이동)
  - ▷ Coprocessor는 높은 속도 실행과 높은 성능을 위해 on-chip으로 구현
- ARM7TDMI coprocessor interface
  - ▷ ARM7TDMI coprocessor는 ARM과의 명령어 실행을 원활하게 하기 위해 handshake 신호를 사용
  - ▷ /cpi (ARM → coprocessor) : ARM이 coprocessor 명령어를 확인
  - ▷ cpa (coprocessor → ARM) : 명령어를 실행할 수 있는 coprocessor 존재 유무를 ARM에 알림 (CoProcessor Absent, 없으면 high)

▷ cpb (coprocessor → ARM) : 명령어를 바로 실행할 수 있는지의 여부를 ARM에 알림 ((CoProcessor Busy, 없으면 high)

- Handshake 출력

▷ 조건 실행 coprocessor 명령어에서 조건 코드 실패로 명령어를 실행할 필요가 없는 경우 ARM은 명령어를 실행하지 않게 되므로 /cpi를 생성 안함

▷ ARM이 명령어 실행을 결정 (/cpi=low)했지만 coprocessor 부재인 경우 (cpa= high) ARM은 undefined trap을 실행하여 trapped 명령 emulation함

▷ ARM이 명령어 실행을 결정하고 coprocessor가 명령어를 받았지만 명령어 실행을 바로 할 수 없을 때 (cpa=low, cpb=high) ARM은 cpb이 high 일까지 기다리고 그동안 인터럽트가 발생하면 coprocessor 명령 실행을 연기

▷ ARM이 명령어 실행을 결정하고 coprocessor가 명령어 실행

- 데이터 이동

▷ ARM은 어드레스를 생성하고 coprocessor는 데이터 길이, 이동 종료 등을 결정, 데이터 이동 동안 인터럽트는 허용되지 않고 최대 16 워드 이동 가능