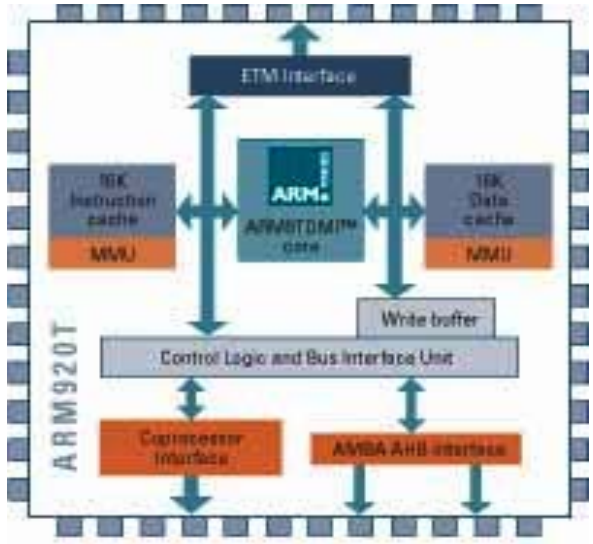
Microprocessor



ARM Core 개요

- ARM : Advanced RISC Machines의 약자
- ARM사는 architecture, processor core, system core를 라이선스 해주는 IP 회사
- Core는 architecture의 구현(implementation)
- ARM의 역사
  - ▷ 1985 : Acorn Computer Group이 세계 최초 상용 RISC 프로세서 개발 (Acorn RISC Machine)
  - ▷ 1991 : ARM사 최초의 내장 가능한 RISC 코어(ARM6) 개발
  - ▷ 1995 : ARM's Thumb architecture extension 제공, ARM8 확장
  - ▷ 1998 : 차세대 ARM10 Thumb 계열 프로세서 제공
- ARM Core Family : 3단계 또는 5단계 파이프라인 사용
  - ▷ ARM7 Family : ARM720T, ARM7EJ-S, ARM7TDMI, ARM7TDMI-S

    Personal audio (MP3, WMA players), 무선 handsets, 휴대용 무선 호출기

▷ ARM9 Family : ARM920T, ARM922T, ARM940T

차세대 휴대용 제품 (비디오폰, PDAs), 디지털 가전 제품 (셋톱박스, 홈 게이트웨이, MP3 audio, MPEG4 video), 영상 제품 (still picture cameras, digital video cameras), 자동차 (Telematic and infotainment systems)

▷ ARM9E Family : ARM926EJ-S , ARM946E-S, ARM966E-S, ARM968E-S

차세대 휴대용 제품 (비디오폰, PDAs, 인터넷 응용 제품), 저장기기 (HDD, DVD), Networking (VoIP, Wireless LAN, xDSL)

▷ ARM10E Family : ARM1020E, ARM1022E, ARM1026EJ-S

차세대 휴대용 제품 (비디오폰, 휴대용 통신기), 디지털 가전 제품 (셋톱박스, 홈 게이트웨이), 영상 제품 (still picture cameras, digital video cameras)

▷ ARM11 Family : ARM1136J(F)-S, ARM1156T2(F)-S and ARM1176JZ(F)-S

자동차 (DVD, 네비게이션), 데이터 저장용 PDA, 디지털 TV, 디지털 카메라, 셋탑박스, router, 스마트폰

- ARM Core를 사용한 임베디드 프로세서

  ▷ StrongARM Core : SA-110, SA-1100, SA-1110, SA-1111 (Intel)

  ARM Core + 각종 주변장치 (USB, PCMCIA, CF, PS/2, DMA) 지원

  ▷ XScale : PXA210, PXA250, PXA255 (Intel)

  무선 통신환경의 휴대 단말기를 목표로 설계된 프로세서 코어

- 삼성전자 : 2001년에 ARM사로부터 ARM946E , ARM926EJ-S , ARM1020E 등의 최신 코어를 도입, ARM의 CPU Core 기술을 바탕으로 스마트폰, PDA, 셋톱 박스 등에 사용될 SOC 칩을 본격적으로 개발 계획

- 주교재 : ARM system-on-chip architecture, steve furber, Addison Wesley
- 참고서적 : ARM Architecture Reference Manual, David Seal, Addison Wesley
- 관련 사이트 : www. arm. com

1. 프로세서 설계 개론

– 전체 요약

The design of a general-purpose processor, in common with most engineering endeavours, requires the careful consideration of many trade-offs and compromises. In this chapter we will look at the basic principles of processor instruction set and logic design and the techniques available to the designer to help achieve the design objectives.

Abstraction is fundamental to understanding complex computers. This chapter introduces the abstractions which are employed by computer hardware designers, of which the most important is the logic gate. The design of a simple processor is presented, from the instruction set, through a register transfer level description, down to logic gates.

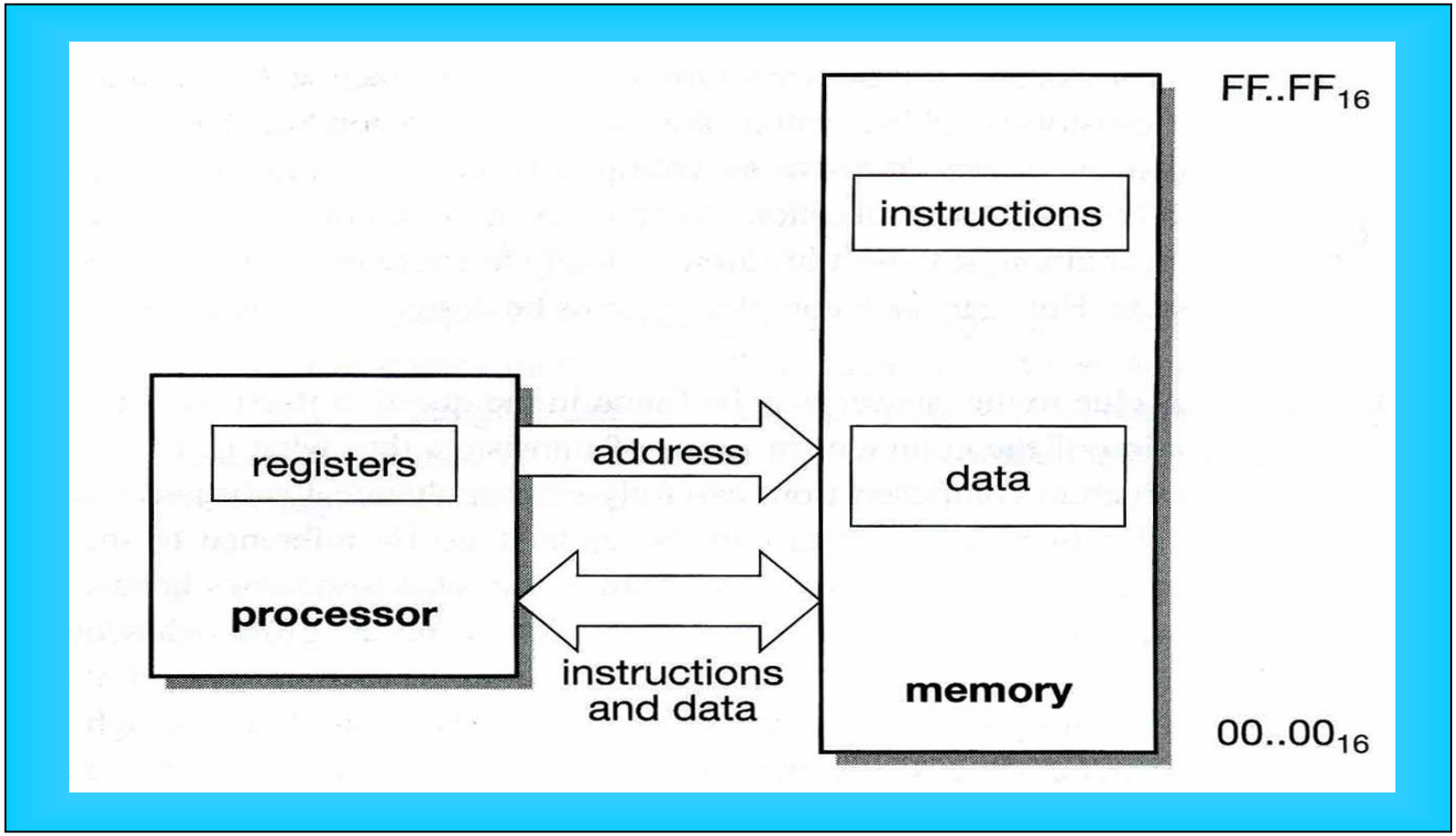
The ideas behind the *Reduced Instruction Set Computer* (RISC) originated in processor research programmes at Stanford and Berkeley universities around 1980, though some of the central ideas can be traced back to earlier machines. In this chapter we look at the thinking that led to the RISC movement and consequently influenced the design of the ARM processor which is the subject of the following chapters.

With the rapid development of markets for portable computer-based products, the power consumption of digital circuits is of increasing importance. At the end of the chapter we will look at the principles of low-power high-performance design.

1.1 프로세서 구조 (architecture)와 조직 (organization)

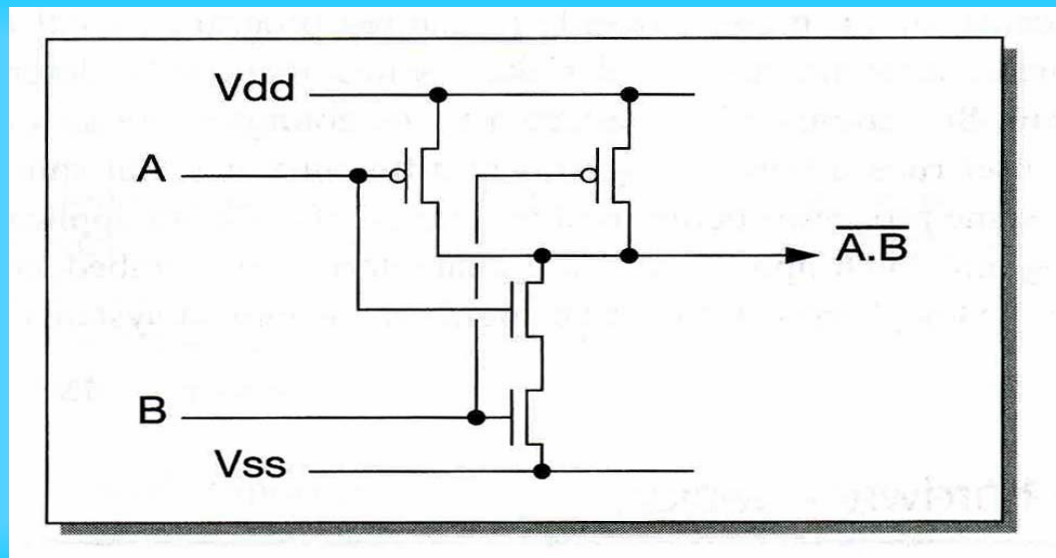
- 저장 프로그램 개념은 1940년도에 개발, 현재 모든 컴퓨터에 적용
- 50년 이상동안 프로세서의 성능은 매우 증가, 비용은 매우 감소
- 컴퓨터 산업의 발달 요인
  - ▷ 컴퓨터의 동작 원리는 거의 변하지 않았음
  - ▷ 전자공학 기술의 발달로 비용 절감, 속도 향상, 전력 소모 감소
  - ▷ 컴퓨터 구조와 조직의 개선으로 성능 향상
- 컴퓨터 구조 (architecture) : 사용자가 직접 이용할 수 있는 부분 (명령어 세트, 레지스터, 메모리 관리 테이블 구조, 예외처리 모델 등)
- 컴퓨터 조직 (organization) : 사용자에 보이지 않는 (이용할 수 없는) 구조의 구현 (파이프라인 구조, transparent 캐쉬, table-walking 하드웨어, translation look-aside 버퍼 등)
- 컴퓨터 발달의 이정표 : 가상 메모리, 캐쉬 메모리, 파이프라인의 도입, RISC idea

- 프로세서 정의 : 메모리에 저장된 명령어를 실행하는 유한 상태 자동 기기
- 프로세서의 상태는 메모리에 저장된 값과 레지스터에 저장된 값에 의해 정의
- 명령어는 프로세서의 상태를 변하게 하고 다음 실행해야 하는 명령어를 정의

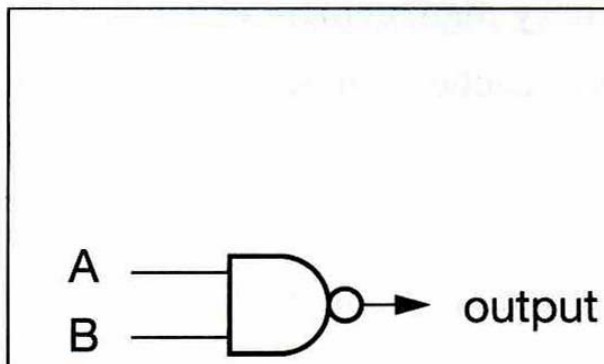


1.2 하드웨어 설계에서의 추상적 개념

- 마이크로프로세서는 1초에 수억 번 스위칭할 수 있는 트랜지스터 수백만개를 사용하여 만들어짐, 하나의 트랜지스터도 정확하게 동작하도록 제어
- 트랜지스터의 물리적인 특성은 매우 복잡하지만 트랜지스터의 동작은 양단 전압과 전류에 대한 수식에 의해 추상화될 수 있음
- 트랜지스터 관계식은 복잡하지만 많은 트랜지스터가 특별한 구조(논리 게이트)에서 사용될 때 그 그룹의 동작은 더욱 간단하게 표현 가능



- Gate 추상화 : 많은 트랜지스터로 논리 게이트 회로를 설계하는 과정을 간단하게 할 뿐만 아니라 게이트가 트랜지스터로 만들어진다는 것을 알 필요가 없게 됨
- 트랜지스터 레벨 회로 설계자는 논리 회로 설계자가 트랜지스터 식을 이해할 필요가 없도록 게이트 추상화를 철저히 지원하여야 함



Logic symbol

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Truth table

- 추상화의 Level : 게이트 추상화 과정과 같이 이전 단계의 몇 개의 성분이 모여 하나의 특성을 이루고 다음 단계의 추상화 됨. 복잡한 하드웨어는 여러 단계의 추상화 과정을 반복

[하드웨어 레벨의 추상화 계층]

1. transistors;
2. logic gates, memory cells, special circuits;
3. single-bit adders, multiplexers, decoders, flip-flops;
4. word-wide adders, multiplexers, decoders, registers, buses;
5. ALUs (Arithmetic-Logic Units), barrel shifters, register banks, memory blocks;
6. processor, cache and memory management organizations;
7. processors, peripheral cells, cache memories, memory management units;
8. integrated system chips;
9. printed circuit boards;
10. mobile telephones, PCs, engine controllers.

1.3 MU0 – simple processor (Manchester University)

– 간단한 형태의 프로세서의 구성 요소

- a **program counter** (PC) register that is used to hold the address of the current instruction;
- a single register called an **accumulator** (ACC) that holds a data value while it is worked upon;
- an **arithmetic-logic unit** (ALU) that can perform a number of operations on binary operands, such as add, subtract, increment, and so on;
- an **instruction register** (IR) that holds the current instruction while it is executed;
- instruction decode and control logic that employs the above components to achieve the desired results from each instruction.

▷ MU0 : 16 비트 프로세서, 12 비트 어드레스 주소 (최대 메모리 8K바이트)

▷ 명령어는 16 비트 길이, 4 비트는 opcode, 12 비트는 어드레스 필드 (S)

▷ ACC := ACC + $mem_{16}[S]$: S에 저장된 메모리 주소의 내용을 읽어 ACC에 더하고 합을 ACC에 저장

- MU0 명령어 세트 (명령어는 8개)

Instruction	Opcode	Effect
LDA S	0000	ACC := mem_{16}[S]
STO S	0001	mem_{16}[S] := ACC
ADD S	0010	ACC := ACC + mem_{16}[S]
SUB S	0011	ACC := ACC – mem_{16}[S]
JMP S	0100	PC := S
JGE S	0101	if ACC ≥ 0 PC := S
JNE S	0110	if ACC ≠ 0 PC := S
STP	0111	stop

- MU0 논리 설계

▷ Datapath : ACC, PC, ALU, IR를 서로 연결하여 명령어 처리 패스를 만듬

▷ Control logic : 데이터패스를 구성하는 모든 요소들의 동작을 제어하는 신호를 생성, 유한 상태 머신 (FSM)에 의해 설계

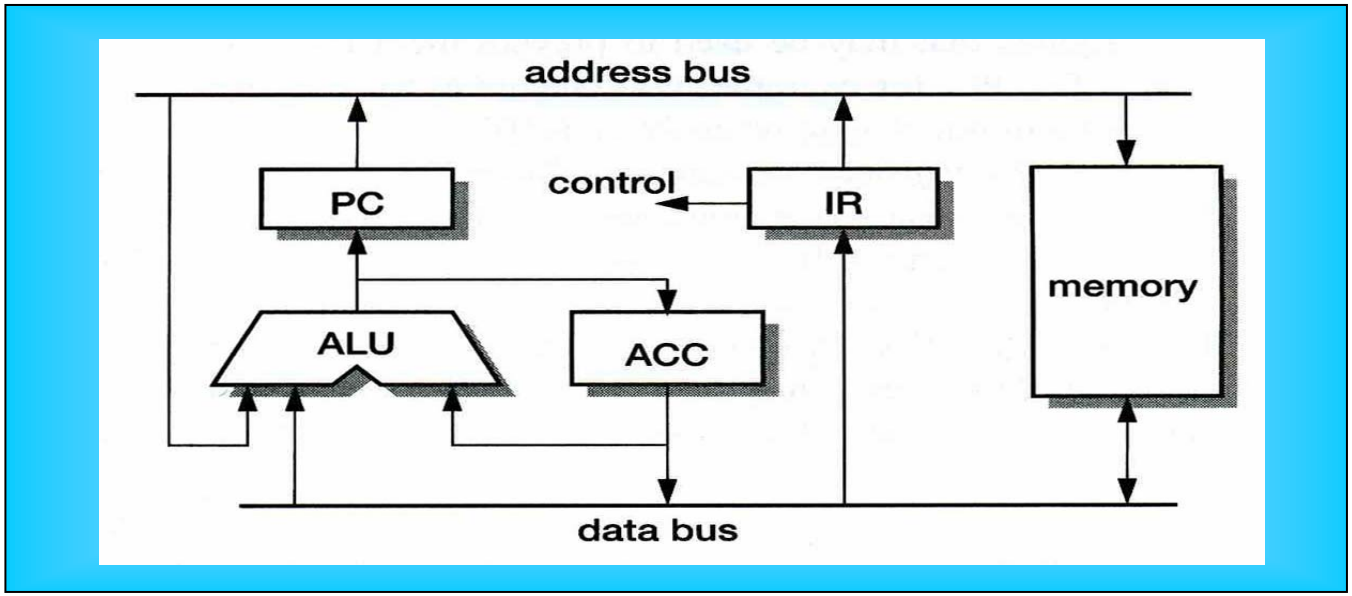
- Datapath 설계

▷ 명령어 세트를 구현하기 위한 많은 방법이 존재, 여러 선택 중에 하나를 선택하기 위해 올바른 선택을 위한 원칙이 필요

▷ MU0 원칙 : 메모리 접근은 무조건 1사이클이 소요

명령어 실행 시간은 메모리 접근 수 만큼의 사이클이 소요됨 (명령어 fetch에 1사이클, 오퍼랜드 fetch, 저장에 1사이클)

▷ MU0 datapath는 1사이클이나 2사이클에 명령어 처리를 완성하도록 설계



▷ 별도의 PC 증가기(incrementer)가 불필요, PC를 새로운 어드레스로 변화시키지 않는 명령어는 2사이클이 소요되므로 ALU를 사용하여 PC를 증가시키는 것이 가능 (LDA, STO, ADD, SUB)

- Datapath의 동작

▷ 명령어가 명령어 레지스터에 저장되어야 명령어가 실행

▷ 명령어 실행 2단계

1. **Access the memory operand and perform the desired operation.**

   The address in the instruction register is issued and either an operand is read from memory, combined with the accumulator in the ALU and written back into the accumulator, or the accumulator is stored out to memory.

2. **Fetch the next instruction to be executed.**

   Either the PC or the address in the instruction register is issued to fetch the next instruction, and in either case the address is incremented in the ALU and the incremented value saved into the PC.

- MU0 프로세서의 초기화

  ▷ 항상 동일한 상태에서 프로세서는 동작해야 함, Reset 입력이 필요

  ▷ ALU를 0로 초기화, PC=0000로 초기화 필요

- 레지스터 전달 레벨 설계

  ▷ datapath가 명령어를 실행하기 위해 요구되는 제어 신호를 결정

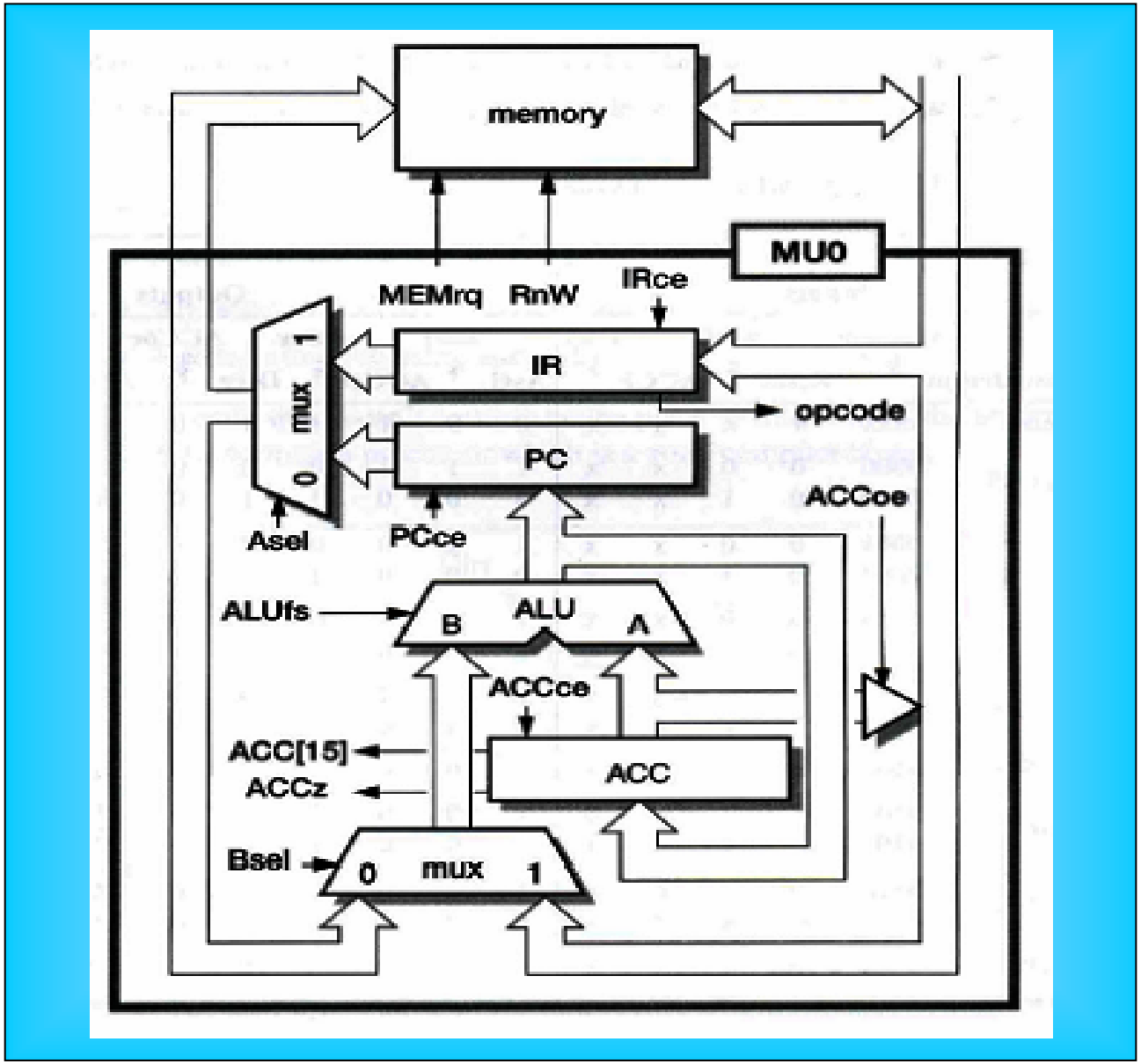
  ▷ 레지스터는 모든 클럭 에지에서 값이 바뀔 수 있지만 특정한 클럭 에지에서만 값이 바뀌도록 하기 위해 제어 신호가 필요 (PCce 제어 신호는 PCce=1일 때에만 PC값이 변화)

  ▷ 설정되는 제어 신호 : 레지스터 제어 신호, ALU 동작 선택 신호, 멀티플렉서 선택 제어 신호, ACC값을 메모리에 보내는 것을 제어하는 신호, 메모리 동작 제어 신호, 메모리 읽기/쓰기 제어 신호

- Control logic : 현재의 명령어를 분석하고 datapath의 제어 신호를 결정

  ▷ FSM(finite state machine)이나 마이크로 명령어로 구현

  ▷ MU0는 두개의 상태(fetch, execute)이므로 (Ex/ft) 상태 비트 추가


memory
MU0
MEMrq
RnW
IRce
IR
opcode
PC
mux
1
0
Asel
PCce
ACCoe
ALUfs
B
ALU
A
ACCce
ACC[15]
ACCz
ACC
Bsel
0
mux
1

[MU0 control logic] – PLA나 조합회로로 구현 가능

Inputs						Outputs									
Instruction	Opcode	Reset	Ex/ft	ACC15 ACCz		Asel	Bsel ACCce	PCce	IRce	ACCoe	ALUfs	MEMrq	RnW	Ex/ft
Reset	xxxx	1	x	x	x	0	0	1	1	0	=0	1	1	0
LDA S	0000	0	0	x	x	1	1	1	0	0	=B	1	1	1
	0000	0	1	x	x	0	0	0	1	0	B+1	1	1	0
STO S	0001	0	0	x	x	1	x	0	0	1	x	1	0	1
	0001	0	1	x	x	0	0	0	1	0	B+1	1	1	0
ADD S	0010	0	0	x	x	1	1	1	0	0	A+B	1	1	1
	0010	0	1	x	x	0	0	0	1	0	B+1	1	1	0
SUB S	0011	0	0	x	x	1	1	1	0	0	A–B	1	1	1
	0011	0	1	x	x	0	0	0	1	0	B+1	1	1	0
JMP S	0100	0	x	x	x	1	0	0	1	0	B+1	1	1	0
JGE S	0101	0	x	x	0	1	0	0	1	0	B+1	1	1	0
	0101	0	x	x	1	0	0	0	1	0	B+1	1	1	0
JNE S	0110	0	x	0	x	1	0	0	1	0	B+1	1	1	0
	0110	0	x	1	x	0	0	0	1	0	B+1	1	1	0
STP	0111	0	x	x	x	1	x	0	0	0	x	0	1	0

- ALU Design

▷ 5개의 동작 수행 (A+B, A-B, B, B+1, 0)

▷ Reset 신호가 ALU를 0로 초기화하는 것을 제어할 수 있으므로 4개의 동작 구분을 위한 2 비트의 동작 선택 신호 필요 (ALUfs는 2비트)

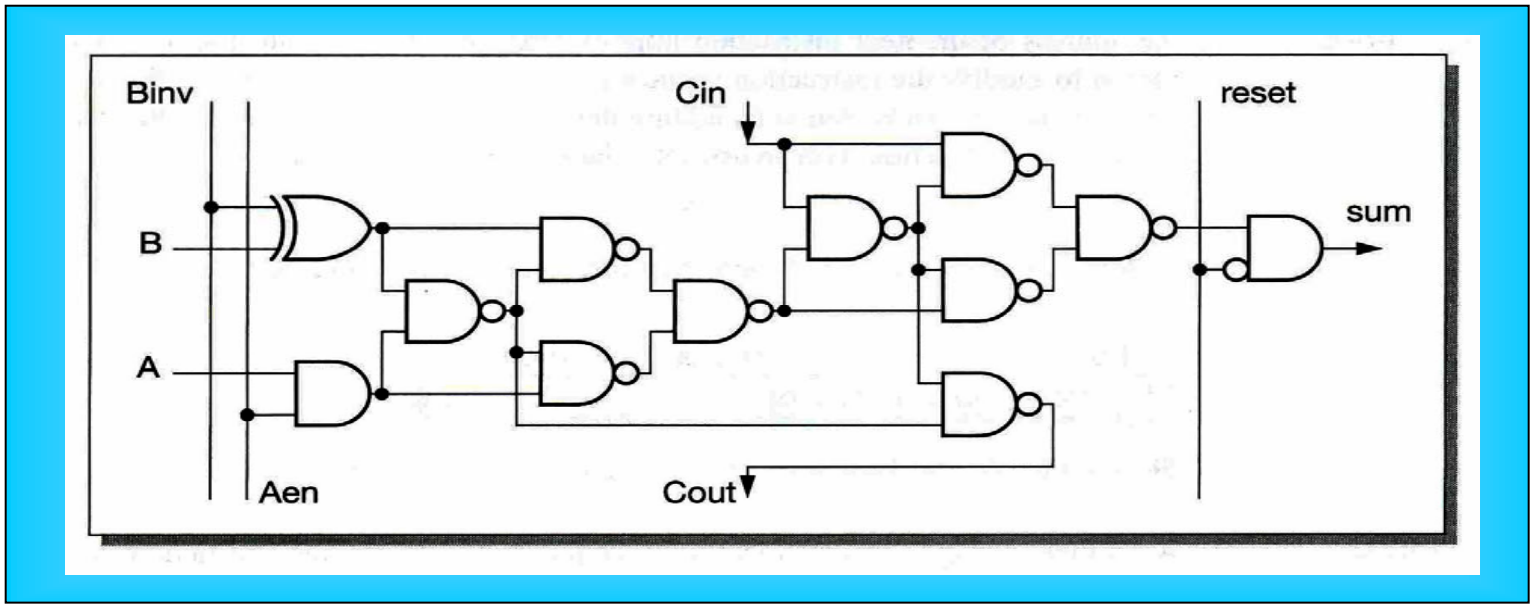
▷ ALU는 binary adder를 확장하여 설계 가능

- $A + B$ is the normal adder output (assuming that the carry-in is zero).
- $A - B$ may be implemented as $A + \overline{B} + 1$, requiring the B inputs to be inverted and the carry-in to be forced to a one.
- B is implemented by forcing the A inputs and the carry-in to zero.
- $B + 1$ is implemented by forcing A to zero and the carry-in to one.

▷ ALU의 제어 신호 : Aen (오퍼랜드 A enable 신호), Binv (invert 제어 신호)

ALUfs 신호에 따라 Aen, Binv, Cin (LSB Carry-in)을 제공하는 조합 회로 필요

[1 비트 MU0 ALU logic]



- MU0가 좋은 프로세서가 되기 위한 과제

- **Extending the address space.**
- **Adding more addressing modes.**
- **Allowing the PC to be saved in order to support a subroutine mechanism.**
- **Adding more registers, supporting interrupts, and so on...**

1.4 명령어 세트 설계

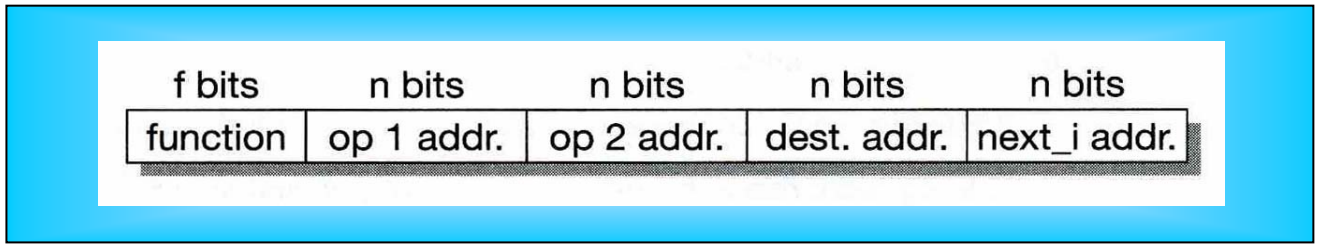
- 두 수를 더하여 결과를 저장하는 ADD 명령어를 위한 명령어 형식을 어떻게 설계할 것인가?

- 4주소 명령어

  ▷ 명령어 종류를 나타내기 위한 비트, 두 source 주소와 목적지를 나타내기 위한 비트, 다음 명령어의 주소를 나타내기 위한 비트가 필요

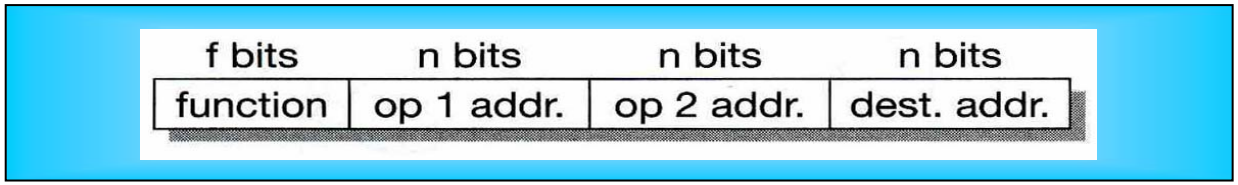
  ▷ 어셈블리 명령어 형식 : ADD d, s1, s2, next_i ; d=s1+s2

  ▷ 각 오퍼랜드 주소가 n 비트, opcode가 f 비트이면 명령어당 4n+f 비트 필요



- 3주소 명령어 : Branch 명령어를 제외한 모든 명령어의 다음 실행될 명령어가 PC에 명령어 크기를 더하여 구할 수 있다면 하나의 주소 공간을 줄일 수 있음

▷ 어셈블리 명령어 형식 : ADD d, s1, s2 ; d=s1+s2



- 2주소 명령어 : 목적지 레지스터를 소스 레지스터 중의 하나와 일치시킴

▷ 어셈블리 명령어 형식 : ADD d, s1 ; d=d+s1

- 1주소 명령어 : 목적지 레지스터가 ALU이면 명령어는 하나의 오퍼랜드는 가짐

- 0주소 명령어 : evaluation 스택을 사용하면 오퍼랜드를 0로 할 수 있음

▷ ADD ; 스택 맨위 = 스택 맨위 + 스택 맨위 다음

- n주소 사용의 예

  ▷ 4주소 명령어는 거의 사용되지 않음

  ▷ 0주소 evaluation 스택 구조 : Inmos transputer, 1주소 구조 : MU0

  ▷ 2주소 구조 : ARM의 Thumb 명령어 세트, 3주소 구조 : ARM 명령어 세트

- 주소 : 3주소 명령어의 주소는 모두 레지스터, 1주소 명령어의 주소는 메모리. 2주소 공간의 주소는 모두 레지스터이거나 하나는 메모리, 하나는 레지스터

- 명령어 종류

- Data processing instructions such as add, subtract and multiply.
- Data movement instructions that copy data from one place in memory to another, or from memory to the processor's registers, and so on.
- Control flow instructions that switch execution from one part of the program to another, possibly depending on data values.
- Special instructions to control the processor's execution state, for instance to switch into a privileged mode to carry out an operating system function.

▷ 명령어들은 동시에 여러 기능을 할 수 있도록 설계

'decrement and branch if non-zero' : (데이터 처리 + 제어 흐름) 명령어

- Orthogonal 명령어 : 명령어를 만들 때의 선택 기준들이 서로 독립적인 명령어

▷ 유사한 명령어는 유사한 구조를 가짐 (add 명령어가 레지스터 주소의 3주소 구조이면 subtract 명령어도 동일한 구조를 가짐)

▷ 컴파일러 작성이 용이하고 하드웨어 구현이 효율적임

- 어드레싱 모드 : 데이터 처리, 데이터 이동 명령어의 오퍼랜드를 억세스하는 방법

1. Immediate addressing: the desired value is presented as a binary value in the instruction.
2. Absolute addressing: the instruction contains the full binary address of the desired value in memory.
3. Indirect addressing: the instruction contains the binary address of a memory location that contains the binary address of the desired value.
4. Register addressing: the desired value is in a register, and the instruction contains the register number.

5. Register indirect addressing: the instruction contains the number of a register which contains the address of the value in memory.
6. Base plus offset addressing: the instruction specifies a register (the **base**) and a binary offset to be added to the base to form the memory address.
7. Base plus index addressing: the instruction specifies a base register and another register (the **index**) which is added to the base to form the memory address.
8. Base plus scaled index addressing: as above, but the index is multiplied by a constant (usually the size of the data item, and usually a power of two) before being added to the base.
9. Stack addressing: an implicit or specified register (the **stack pointer**) points to an area of memory (the **stack**) where data items are written (**pushed**) or read (**popped**) on a last-in-first-out basis.

- 제어 흐름 명령어 : 정상적인 프로그램 흐름을 벗어나기 위한 명령어, PC를 수정

▷ 대부분의 branch는 짧은 범위에서 일어남 ('PC-relative' branch)

▷ 어셈블러가 특정 위치로 이동하기 위한 상대적인 변위를 계산

▷ 이동하는 최대 변위는 변위에 할당된 비트 수에 의해 결정

- 조건 branch 명령어 : 특정 명령어 시퀀스를 반복하거나 데이터 값에 따라 실행되는 프로그램이 달라야 할 때 사용되는 명령어

  ▷ 특정 레지스터값이 0이거나 두 레지스터 값이 같으면 branch 등

  ▷ 이를 위해 데이터 처리 결과를 저장하는 condition code 레지스터가 필요

- Subroutine calls

  ▷ 요청된 곳으로 다시 돌아오기 위해 복귀 주소를 저장하는 것이 필요

  ▷ 복귀 주소는 프로세서에 따라 메모리, 스택, 레지스터에 저장

- System calls : operating system routine

  ▷ 입출력 주변장치 접근 등의 프로세서의 특별한 기능은 사용자 코드로부터 보호될 필요가 있으므로 이런 특별한 기능을 사용할 때는 system call 사용

  ▷ 악의적인 사용자가 시스템 코드를 변화시키지 못하도록 프로세서는 보호 모드를 반드시 지원

- 예외 (Exceptions) : 프로그램 실행에서 예상하지 못한 변화 (하드웨어 인터럽트, 소프트웨어 인터럽트, 메모리 시스템에서의 오류)

1.5 프로세서 설계 원칙 (trade-offs)

- 명령어 세트는 프로그래머에게 유용하고 구현하기 효율적인 기능들을 지원하여야 하고 미래지향적이고 복잡한 구현이 가능하여야 함
- 컴파일러 작성이 용이하고 고급 언어를 효율적으로 지원하여야 함
- CISC (Complex Instruction Set Computer)
  - ▷ 1980 이전에는 컴파일러를 간단하게 하기 위해 명령어를 복잡하게 설계
  - ▷ 복잡한 많은 수의 명령어를 해독하는데 많은 실리콘이 필요, 크기가 커짐
  - ▷ 인텔 8086 계열, 모토롤라 68000 계열 마이크로 프로세서
- RISC revolution : 고급 언어와 명령어 세트의 문법적인 갭을 줄이기 위해 명령어를 복잡하게 만드는 것은 효율적인 컴퓨터를 만들지 못한다는 생각에서 출발
  - ▷ 작은 명령어 세트로 인해 남는 실리콘을 다른 유용한 목적을 위해 사용
  - ▷ ARM, MIPS 프로세서는 RISC 프로세서
  - ▷ 프로세서는 무엇을 하고 어떻게 속도를 증가시킬 수 있는지 파악 필요

- 프로세서 실행 분석

▷ 실행되는 명령어 빈도를 측정하는 것이 필요

Instruction type	Dynamic usage
Data movement	43%
Control flow	23%
Arithmetic operations	15%
Comparisons	13%
Logical operations	5%
Other	1%

▷ 데이터 이동 명령어가 실행하는데 프로세서는 시간을 가장 많이 소비

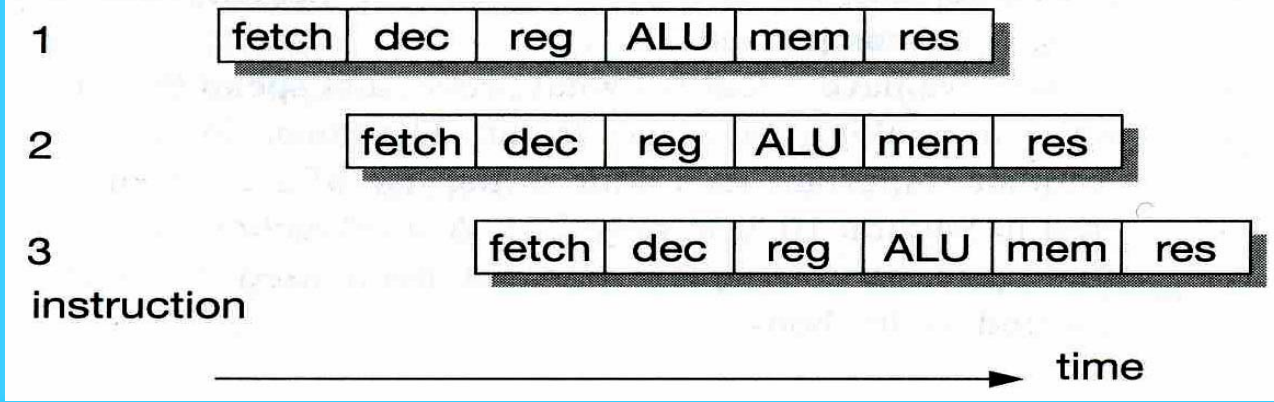
▷ 프로세서의 속도를 증가시키는 방법 : 파이프라인, 캐쉬 메모리, 슈퍼 스칼라 명령어 실행

– 파이프라인 (6단계 명령어 실행인 경우)

1. Fetch the instruction from memory (fetch).
2. Decode it to see what sort of instruction it is (dec).
3. Access any operands that may be required from the register bank (reg).
4. Combine the operands to form the result or a memory address (ALU).
5. Access memory for a data operand, if necessary (mem).
6. Write the result back to the register bank (res).

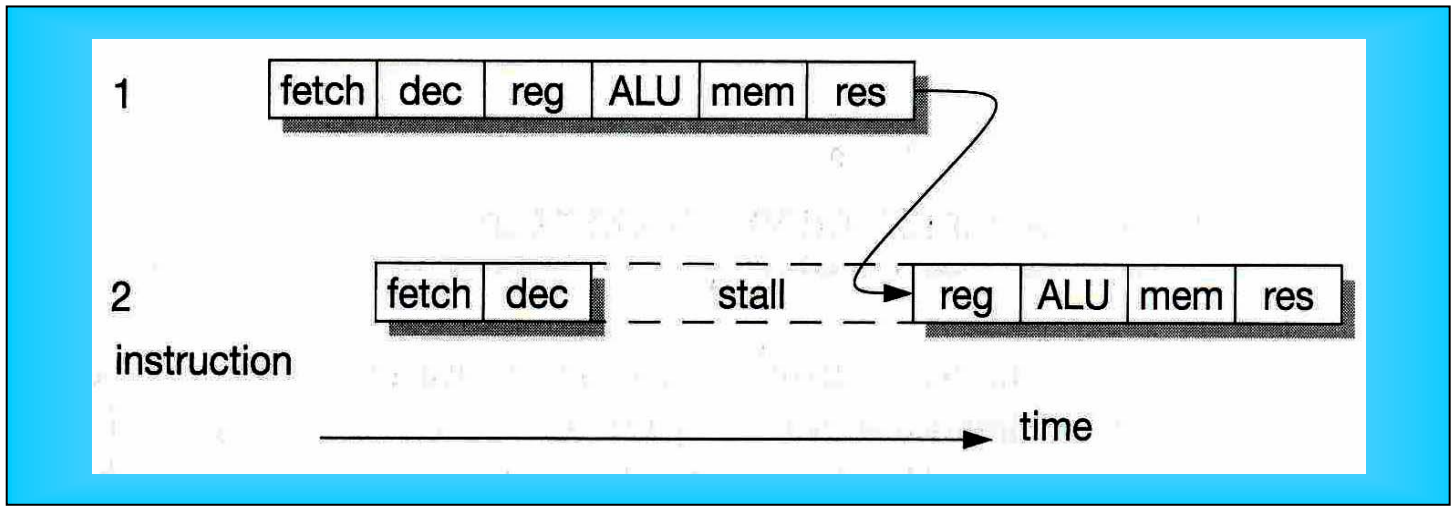
▷ 모든 명령어가 6단계를 필요로 하는 것은 아니지만 6단계를 반드시 거침

▷ 하드웨어 자원을 효율적으로 사용하기 위해 병렬로 명령어 처리(속도 6배 ↑)



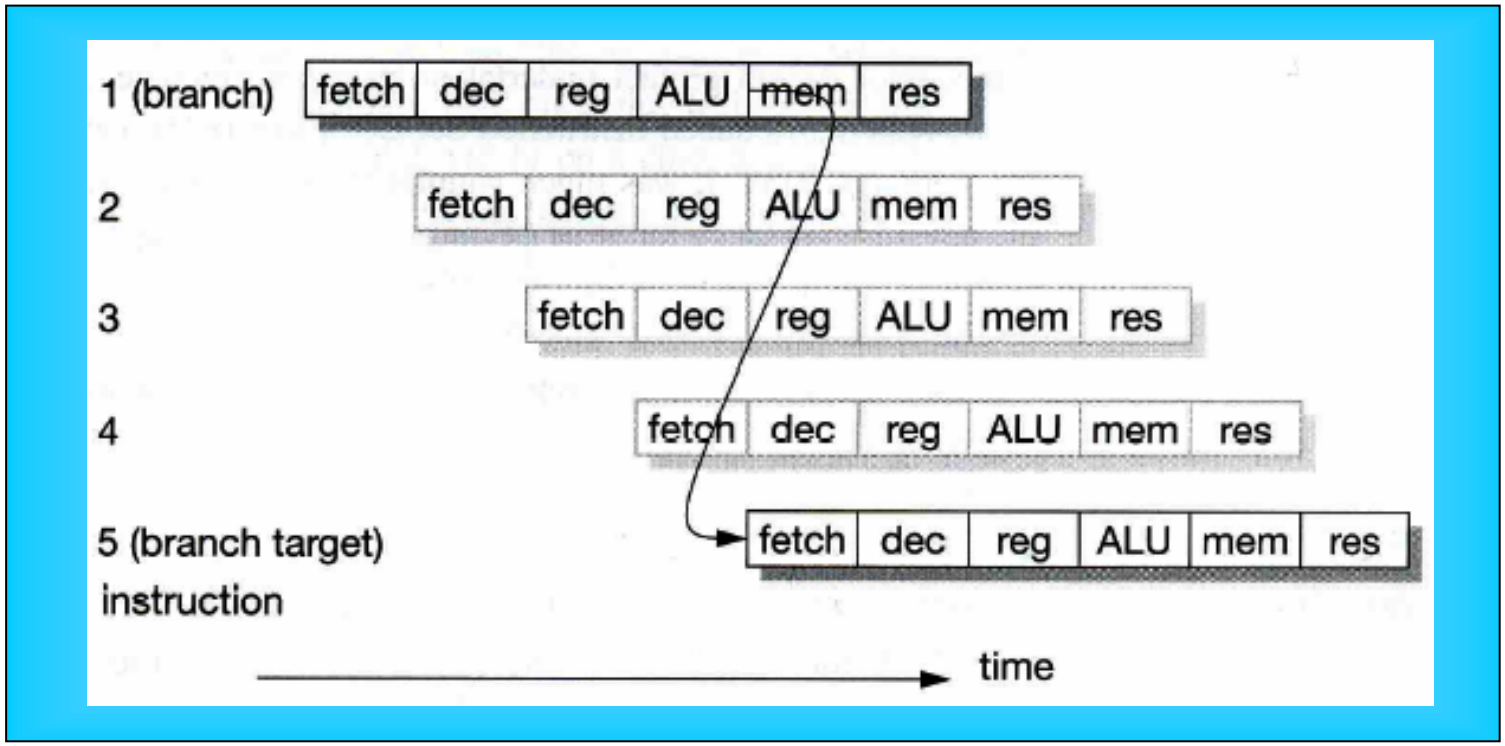
- 파이프라인 해저드

▷ 데이터 해저드 (read-after-write hazard) : 앞선 명령어의 결과가 다음 명령어의 소스 오퍼랜드로 사용되는 경우 발생



데이터 해저드 때문에 3 클럭이 지연되어야 하지만 내부 소스 (ALU, 메모리)에서의 전방전달을 통해 지연 클럭을 줄일 수 있음

▷ 제어 해저드 (branch hazard) : branch 명령어인 경우 이동해야 하는 주소를 계산하기 전까지 의미 없는 다음 명령어들이 실행



3 클럭 뒤에 branch 목표 위치의 명령어가 실행 (3 클럭 지연)

ALU에서 주소를 계산하지 않고 dec 단계에서 주소 계산 필요

조건부 branch 명령어의 경우 ALU 결과에 따라 branch 여부 결정

이전 branch 상태에 따라 branch를 결정하는 동적 예측 알고리즘 사용

branch 다음의 명령어를 무조건 실행하는 delayed branch 명령어 사용

- 파이프라인 효율성

  ▷ 파이프라인 단계가 많아질수록 프로세서의 성능이 좋아지지만 파이프라인 해저드가 많이 발생하여 구현하기가 어려움

  ▷ 1980년도의 CISC 마이크로 프로세서는 제한된 공간, 복잡한 명령어 때문에 파이프라인으로 구현되지 않았음

1.6 RISC (Reduced Instruction Set Computer)

- 초기 RISC 프로세서인 Berkeley RISC I은 CISC 프로세서보다 더 간단하고 규모가 작지만 비슷한 성능을 유지

- RISC 구조 (architecture)

  ▷ 몇 개의 형식을 가진 고정된 명령어 길이, CISC는 명령어마다 명령어 길이와 명령어 형식이 달라 많은 명령어 형식과 길이 존재

  ▷ 데이터 처리 명령어는 레지스터에 있는 데이터만을 가지고 처리하는 load-store 구조 (메모리 접근 명령어와 데이터 처리 명령어를 구분)

▷ 32개의 32비트 레지스터 뱅크 (load-store 구조를 효율적으로 지원하고 범용 목적을 위해 레지스터를 사용, 데이터나 주소 저장 등의 특정 목적을 위해 사용되는 CISC 레지스터보다 레지스터 수가 많음)

- RISC 조직 (organization)

- Hard-wired instruction decode logic; CISC processors used large microcode ROMs to decode their instructions.
- Pipelined execution; CISC processors allowed little, if any, overlap between consecutive instructions (though they do now).
- Single-cycle execution; CISC processors typically took many clock cycles to complete a single instruction.

- RISC 장점

▷ 프로세서가 간단하기 때문에 작은 실리콘 면적이 필요, 남은 실리콘 면적을 캐쉬 메모리, 메모리 관리 기능, floating-point 하드웨어 등의 성능 강화 블록을 위한 공간으로 사용

▷ 낮은 설계 비용과 짧은 개발 기간 필요

▷ 높은 성능 (CISC 프로세서 설계자는 공감하지 않음)

복잡한 명령어는 프로그램 실행 시간이 많이 소요, 간단한 프로세서에서 복잡한 기능을 하나씩 추가하여 복잡한 명령어를 만드는 것이 필요

- RISC 회상

▷ 파이프라인을 통해 프로세서의 속도를 증가시킴

▷ 프로세서가 간단하기 때문에 single 사이클로 실행할 수 있고 높은 클럭에서 동작하도록 설계 가능

▷ 일정한 명령어 길이와 load-store 구조를 가진 CISC 프로세서 (microcode, multi 사이클 실행, no pipeline)를 구현할 수 있으나 장점이 없음

▷ RISC 초기에 hard-wired, single 사이클 실행의 파이프라인 CISC를 구현하는 것은 불가능

▷ 현재는 서로의 장점을 서로 보완하는 EISC 프로세서 출시 (?)

- RISC 단점

  ▷ RISC는 CISC에 비해 코드 밀도가 낮음 (코드 길이가 증가하고 cache 적중률이 떨어져 메모리 access 빈도를 증가시킴)

  ▷ x86 코드를 실행할 수 없음 (PC에서 구동되는 ARM용 emulation 소프트웨어는 ARM 코드를 생성하도록 개발되어야 함)

- ARM 코드 밀도와 Thumb : 코드 밀도를 증가시키기 위해 Thumb 구조를 도입 (Thumb 명령어는 기존의 32비트 명령어보다 작은 16비트 명령어)

1.7 낮은 파워 소모를 위한 설계

- 현재 마이크로프로세서의 설계 요인 중에 파워 소모를 줄이는 것이 점차 대두
- 이동형 기기, 장치에 대한 수요 증가로 파워 소모가 작은 마이크로프로세서에 대한 수요가 점점 증가
- 비용이 작고 성능이 우수하고 전력 소모가 작은 마이크로 프로세서가 설계 목표
- ARM 프로세서는 낮은 파워 소모를 위한 설계를 고려하기 적합한 프로세서

- 저전력 회로 설계

1. **Minimize the power supply voltage, *Vdd*.**

   The quadratic contribution of the supply voltage to the power dissipation makes this an obvious target. This is discussed further below.

2. **Minimize the circuit activity, *A*.**

   Techniques such as clock gating fall under this heading. Whenever a circuit function is not needed, activity should be eliminated.

3. **Minimize the number of gates.**

   Simple circuits use less power than complex ones, all other things being equal, since the sum is over a smaller number of gate contributions.

4. **Minimize the clock frequency, *f*.**

▷ 이전에는 Vdd가 5V인 프로세서가 많았지만 현재 대부분 3.3V이고 나아가 2.5V, 2V 이하인 프로세서가 점차적으로 개발 예정

▷ 클럭 주파수를 줄이면 파워 소모는 작지만 성능이 떨어짐, 적절한 절충이 필요