RTN : SRC 언어를 정의하는 언어 (Meta language)

:= naming operator
ex) op<4...0> := IR<31..27> : IR의 <31..27>을 op<4..0>이라는 새로운 이름으로 부르겠다는 의미

(op = 12) -> R[ra] <- R[rb] + R[rc] : 조건문
Run : 1이면 동작상태, 0이면 정지상태

Strt : start signal

R[0..31]<31..0> : general purpose registers
Mem[0..2^32-1]<7..0> : 2^32 addressable bytes of memory
M[x]<31..0> :=Mem[x]#Mem[x+1]#Mem[x+2]#Mem[x+3] : 32bit 메모리를 정의
-#: 연결 연산자

instruction formats
-op<4..0> := IR<31..27> : operation code field
-ra<4..0> := IR<26..22> : target register field
-rb<4..0> := IR<21..17> : operand, address index or branch target register
-rc<4..0> := IR<16..12> : second operand, shift counter register
-c1<21..0 := IR<21..0> : long displacement field
-c2<16..0> := IR<16..0> : short displacement or immediate field
-c2<11..0> := IR<11..0> : count or modifier field

Effective address calculations
-disp<31..0> := ((rb = 0) -> c2<16..0> {sign extend}:
(rb != 0) -> R[rb] + c2<16..0> {sign extend}):
-rel<31..0> := PC<31..0> + c1<21..0> { sign extend } :
(rel => relative)

instruction interpretation : RTN Description of Fetch / Execute

- instruction_interpretation:= (

ㄱRunㅅStrt -> Run <- 1:

Run -> (IR <- M[PC]: PC <- PC + 4; instruction_execution));

=>전원이 들어오면 Run을 1 상태로 만들고, Run이 1 이면 IR에다가 M[PC]를 집어넣어라. PC는 다음 명령어를 가리키는 주소이다.

-: (콜론)으로 끊어져 있는 문장들은 동일한 clock 내에서 수행된다.

-;(세미콜론)으로 끊어져 있는 문장들은 successive clock 내에서 수행된다.

individual instructions

- instruction_interpretation 뒤에 instruction_execution이 나온다.
- instruction_execution := (

ld (:= op = 1) -> R[ra] <- M[disp]: : load register

ldr (:= op = 2) -> R[ra] <- M[rel]: : load register relative

st (:= op = 3) -> M[disp] <- R[ra]: : store register

str (:= op = 4) -> M[rel] <- R[ra]: : store register relative

la (:= op = 5) -> R[ra] <- disp : load displacement address

lar(:= op = 6) -> R[ra] <- rel : load relative address

):

SRC RTN – The Main Loop

-ii := instruction_interpretation

-ie := instruction_execution

-**ii** := (ㄱRunㅅStrt -> Run <- 1:

Run -> (IR <- M[PC]: PC <- PC + 4; **ie**));

-**ie** := (

ld (:= op = 1) -> R[ra] <- M[disp]: : load register

ldr (:= op = 2) -> R[ra] <- M[rel]: : load register relative

st (:= op = 3) -> M[disp] <- R[ra]: : store register

stop (:= op = 31) -> Run <0 ;

...

) ; **ii** =>**ii and ie invoke each other**

RTN Descriptions of SRC Branch Instructions

-Branch condition determined by 3 lsbs of inst.

-cond := (c3<2..0> = 0 -> 0 : **never**
c3<2..0> = 1 -> 1 : **always**
c3<2..0> = 2 -> R[rc] = 0 : **if register is zero**
c3<2..0> = 3 -> R[rc] != 0 : **if register is nonzero**
c3<2..0> = 4 -> R[rc]<31> = 0 : **if positive or zero**
c3<2..0> = 5 -> R[rc]<31> = 1); **if negative**

-br(:= op = 8) -> (cond -> PC <- R[rb]); **conditional branch**

-brl (:= op = 9) -> (R[ra] <- PC: **branch and link**
cond -> (PC <- R[rb]));

RTN for Arithmetic and Logic

-shr ra, rb, rc : shift rb right into ra by 5 lsbs of rc (rc의 오른쪽 5bit만큼 shift 한다)

-shr ra, rb, count : shift rb right into ra by 5 lsbs of inst

(shift count가 명령어 자체가 들어가 있다)

shift 에서의 count는 명령어의 오른쪽 5bit 혹은 레지스터의 오른쪽 5bit 일 수 있다.

-@ : 복제 연산자, # : 연결 연산자

-n : =((c3<4..0> = 0) -> R[rc]<4..0>:
(c3<4..0> != 0) -> c3<4..0>) :

-shr (:= op = 26) -> R[ra]<31..0> <- (n@0) # R[rb]<31..n>:

-shra (:= op = 27) -> R[ra]<31..0> <- (n@R[rb]<31>) # R[rb]<31..n>: // sign bit 복제

-shl (:= op = 28) -> R[ra]<31..0> <- R[rb]<31-n .. 0> # (n @ 0) :

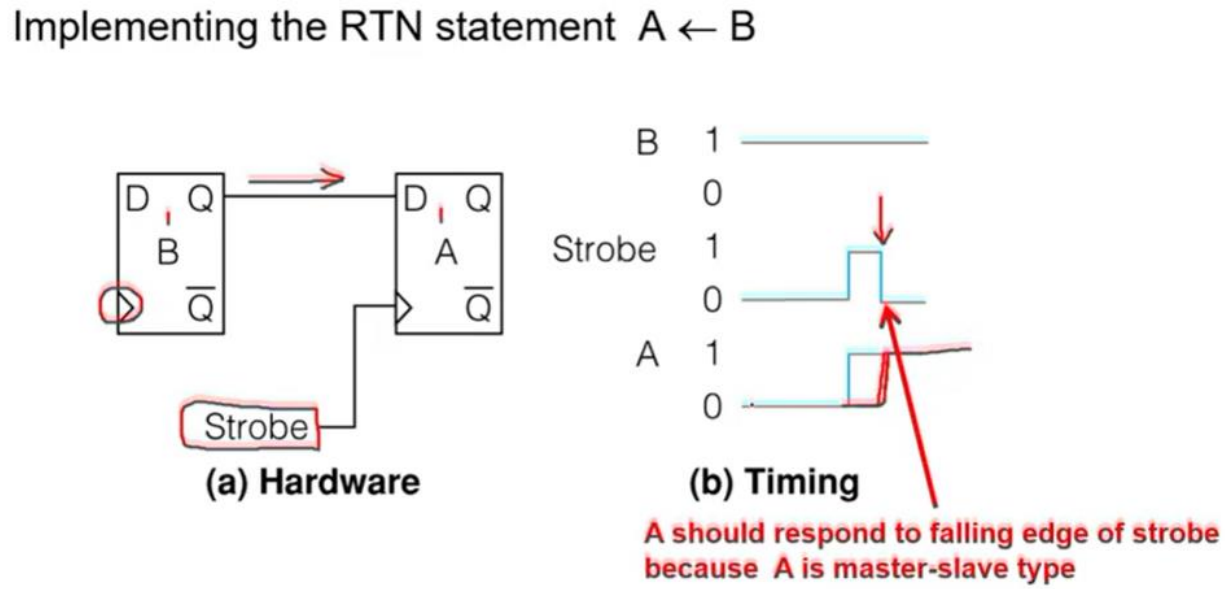
-shc (:= op = 29) -> R[ra]<31..0> <- R[rb]<31-n .. 0> # R[rb]<31.. 32-n>:

(shc는 shift left를 하는데 원래는 버려야 할 bit를 오른쪽에 채운다)

RTN vs SRC

-RTN을 작성한 것을 RTN Compiler를 이용하여 컴파일을 하면 processor를 뱉어낸다. 이것이 바로 SRC 언어를 돌릴 때 사용했던 SRC simulator가 된다. 따라서, 이 SRC simulator을 활용하여 src 언어를 작성해서 assemble language를 볼 수 있다.

Register transfers can be mapped to Digital Logic Circuit



-Master-slave 타입이므로 Strobe 신호의 falling edge 에서 A <- B 가 실행된다.

-Gates 신호는 출력을 제어하고, Strobe 신호는 입력을 제어한다.

-

Abstract and Concrete RTN for SRC

1) add

Abstract RTN : (IR <- M[PC]: PC <- PC + 4; instruction_execution);

instruction_execution := (

add (:= op = 12) -> R[ra] <- R[rb] + R[rc]:

Concrete RTN :

MA <- PC: C <- PC + 4;

MD <- M[MA]: PC <- C;

IR <- MD;

A <- R[rb];

C <- A + R[rc];

R[ra] <- C

2) addi

Abstract RTN : R[ra] <- R[rb] + c2<16..0>

Concrete RTN :

A <- R[rb];

C <- A + c2<16..0>;

R[ra] <- C

disp<31..0> := ((rb = 0) -> **c2**<16..0> {sign extend } :
(rb != 0) -> R[rb] + c2<16..0> {sign extend }):

3) ld

Abstract RTN : R[ra] <- M[disp]

Concrete RTN :

instruction fetch 과정
A <- (rb = 0 -> 0: rb!= 0 -> R[rb]);
C <- A + (**16@IR<16>#IR<15..0>);**
MA <- C
MD <- M[MA];
R[ra] <- MD;

4) st

Abstract RTN : M[disp] <- R[ra]

Concrete RTN :

instruction fetch 과정
A <- (rb = 0 -> 0: rb!= 0 -> R[rb]);
C <- A + (16@IR<16>#IR<15..0>);
MA <- C
MD <- R[ra];
M[MA] <- MD;

***같은 사이클에 진행되면 동시에 진행 될 수 있다는 것임.

5) br

Abstract RTN : br –> (cond -> PC <- R[rb]);

Concrete RTN :

instruction fetch 과정
CON <- cond(R[rc]);
CON -> PC <- R[rb];

6) shr

Abstract RTN : R[ra]<31..0> <- (n@0)#R[rb]<31..n> :
n := ((c3<4..0>=0) -> R[rc]<4..0> :
(c3<4..0> != 0) -> c3<4..0>) ;

Concrete RTN :

instruction_fetch 과정

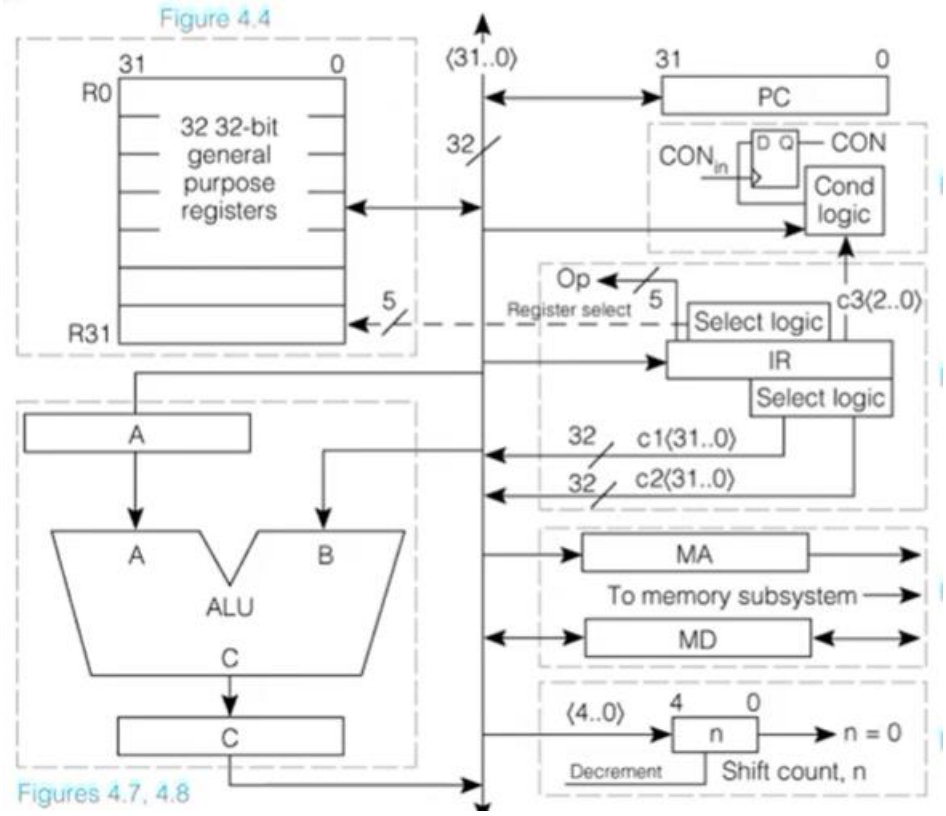
n <- IR<4..0>;

(n = 0) -> (n <- R[rc]<4..0>);

C <- R[rb];

Shr (:= (n!=0) -> (C<31..0> <- 0#C<31..1>: n <- n-1 ; Shr));

R[ra] <- C;



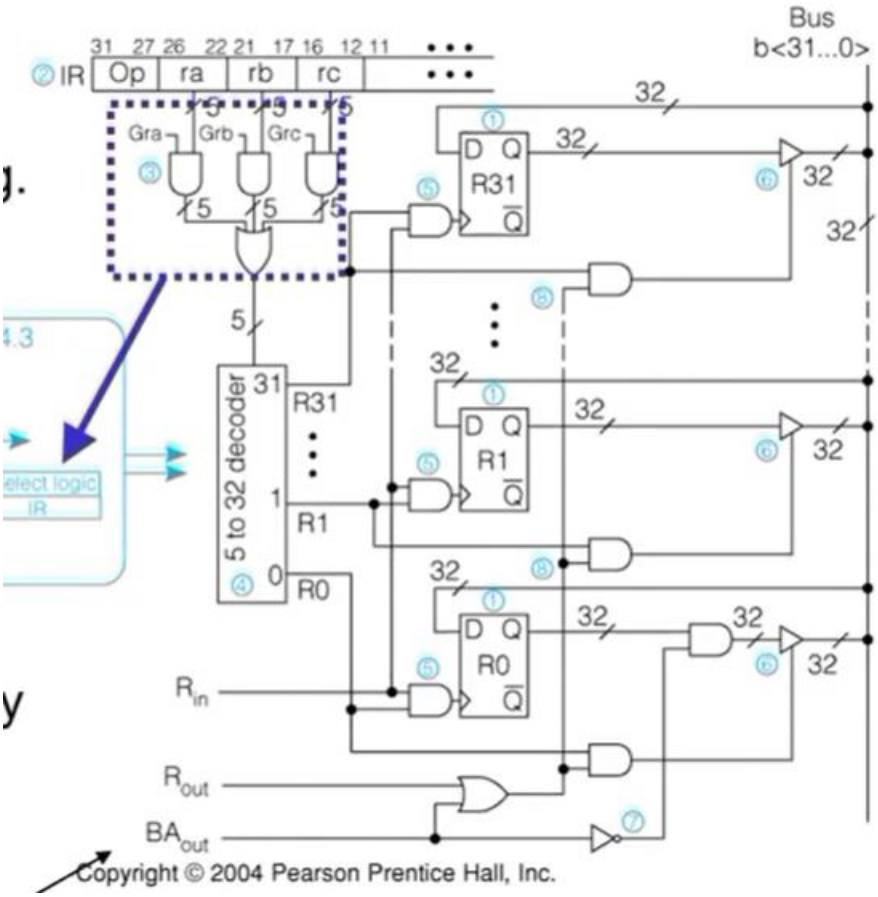
-IR밑에 Select Logic 은 c1, c2를 sign extend 해준다.

-OP의 5비트는 Control Unit 으로 간다.

-c3<2..0> 은 branch condition 따지는데 사용되는데, Cond logic 으로 간다.

-shift count인 n은 Decrement 라는 n의 값을 줄이는 신호를 가지고 있다.

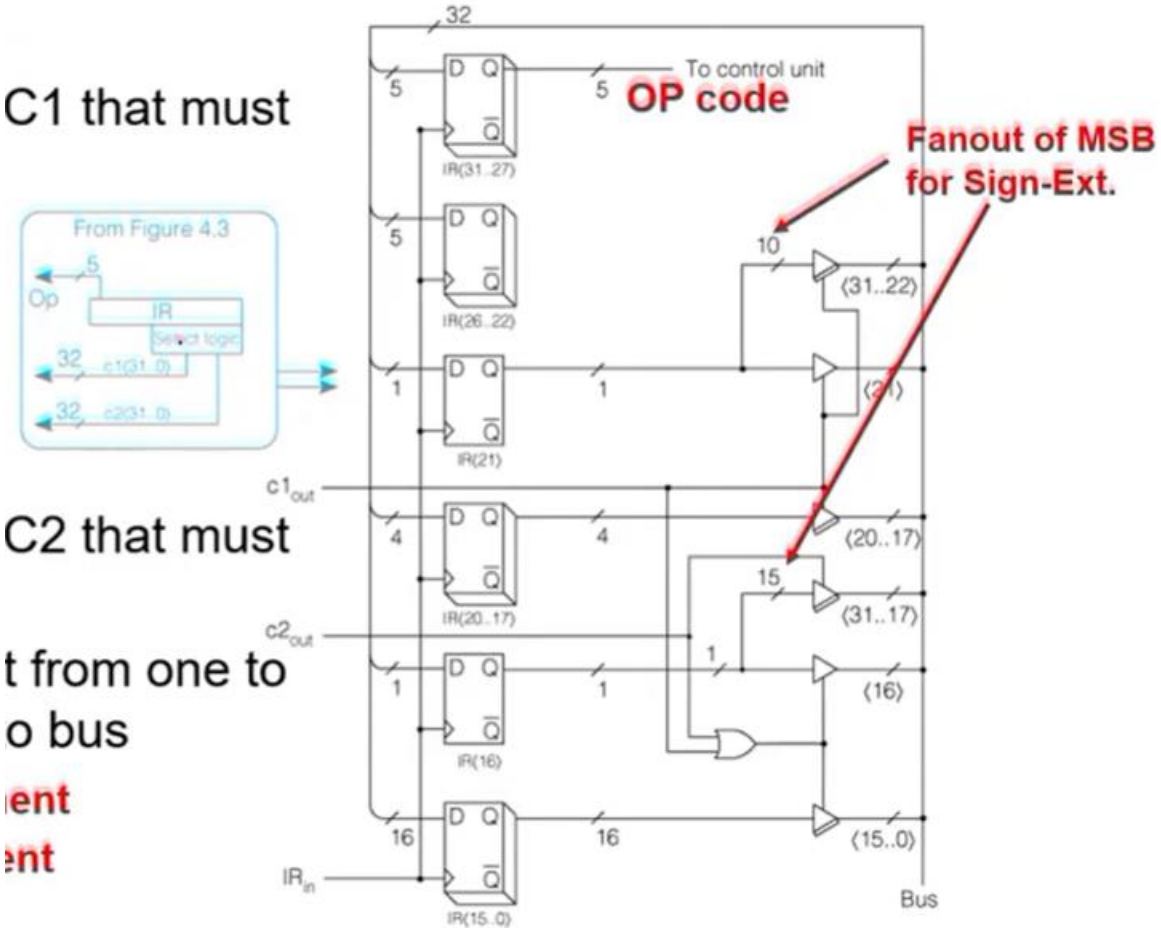
flip flop : Edge triggering (master – slave)
latch : simpler hardware



BAout : R[0] 레지스터 즉, **0번 레지스터가 선택되었을 때 0 을 내보내야 해서** 별도의 신호를 씀
=>Disp 계산시 사용한다
-0번 레지스터말고 따른거는 tri-state-buffer를 통해서 바로 값을 내보낸다
-0번 레지스터는 앞서 말한거와 같이 0을 출력해야 되서 별도의 회로가 첨가됨

Rout : R[0] 말고 다른 레지스터에 대한 출력 신호
(회로도 사진 첨가)

Extracting c1, c2, and op from the instruction register
-select logic 에 따라서 c1, c2 신호가 나와서 버스에 물리게 된다
-c1, c2는 각각 22bit, 17bit 짜리인데 이것을 32bit로 sign extension을 한 상태에서 나와야 한다
-따라서, **select logic은 c1, c2를 sign extension 해주는 회로로 구성이 된다**



=>**c1와 c2는 각각 원래 22bit, 17bit 짜리 값인데 나올때 각각 32bit로 맞춰서 내보내야함**

-2차원 형태는 1bit, 3차원 형태는 여러 bit를 의미한다

-**c1out, c2out는 각각을 32bit로 sign extension 해주는 신호**이다

-**IR(21)은 c1의 최상위 비트(sign bit)를 <31...21> 만큼 복제**해서 내보내는 기능

-**IR(16)은 c2에 대해서 sign extension 기능을 수행**

-IRin은 strobe 신호이다

-최종적으로 Bus로 내보낸다

CPU to Memory interface : MA and MD Registers

<입력쪽>

-MA의 strobe신호인 MAin이 활성화 되고(falling edge), MA의 입력은 버스를 통해서 32비트로 들어온다

-MA는 메모리의 주소를 가지고 있기 때문에 출력은 항상 메모리로만 향한다

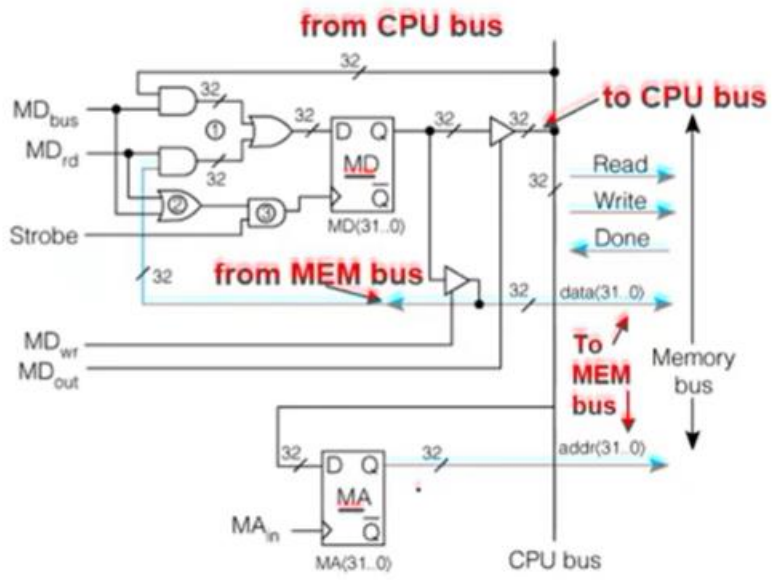
-MD는 써놓을 값(Write)을 집어넣는 역할과 읽어온 것(Read)을 보관하는 역할을 한다

-MD는 Memory Read, Memory Write 기능이 있다

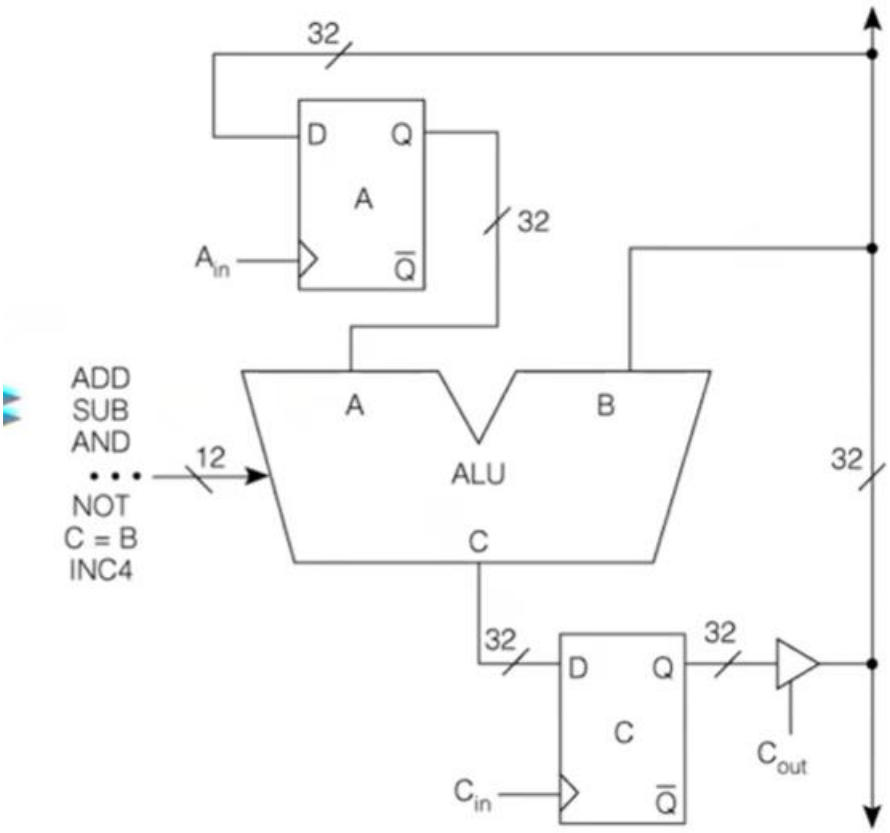
-버스를 통해 어떤 값이 오고 MDbus 신호가 활성화가 되면 MD에 값이 들어간다. (Write)
-메모리로 부터 온 값이 MDrd 신호가 활성화가 되면 MD에 값이 들어간다. (Read)
-하지만, MD에 값이 들어가기 위해서는 MDbus 혹은 MDrd 둘중 하나가 활성화가 되고, Strobe 신호가 활성화가 돼야 한다.

<출력쪽>
-Read에 해당 하는 MD에 저장된 값은 출력쪽의 맨 위의 tri state buffer를 지나 버스를 타고 간다. (MDout 신호가 활성화 돼야함)

-Write에 해당 하는 MD에 저장된 값은 출력쪽의 맨 아래의 tri state buffer를 지나 메모리로 향한다. (MDwf 신호가 활성화 돼야함)



The ALU and its Associated Registers



-**bus로 부터 첫번째 피연산자가 들어와서** Ain 신호가 활성화가 되면 **A로 들어간다**

-**두번째 피연산자는 bus로 부터 바로 들어와서 A와 연산** 후, 순식간에 C를 통해서 결과 레지스터에 들어간다 (Cin 신호가 활성화 되어야 함)

-그러나, 결과값인 C는 Cout 신호가 활성화 되어야지 bus를 통해서 나간다

-bus의 독점 사용 방지를 위해 Cout 이라는 제어 신호가 필요하다.

(Cout 게이트가 없다면 A + B -> 2A + B -> 3A + B .... 와 같이 원하지 않는 결과가 나올 수 있다)

From Concrete RTN to Control Signals : The Control Sequence

-Control Signal (제어 신호) 이 순차적으로 모여있는 것이 **Control Sequence** 이다

<instruction fetch>

Concrete RTN	**Control Sequence**
MA<- PC : C<- PC + 4;	PCout, MAin, Inc4, Cin
MD<-M[MA]: PC <- C;	Read, Cout, PCin, Wait
IR<-MD	MDout, IRin
instruction execution	

=>**이런 Control signal을 때에 맞게 발생시켜 주는 것이 Control Unit의 역할**이다.

1) add(:= op = 12) -> R[ra] <- R[rb] + R[rc]

Concrete RTN	**Control Sequence**
MA<- PC: C <- PC + 4;	PCout, MAin, Inc4, Cin, Read
MD <- M[MA]: PC<- C;	Cout, PCin, Wait
IR <- MD;	MDout, IRin
A <- R[rb]	Grb, Rout, Ain
C <- A + R[rc]	Grc, Rout, ADD, Cin
R[ra] <- C	Cout, Gra, Rin, End

=>위의 Read와 다른 점은 위에는 MAin이 master-slave 라서 falling edge에서 값이 들어가 시간을 낭비하게 된다. 하지만, 이 경우는 MAin를 Latch type으로 사용하는데 level sensitive 이므로 Read가 기존보다 일찍 들어가게 된다.

=>**End 신호를 만나면 다시 T0 로 돌아가서 그 다음 instruction을 실행**한다.

2) addi (: = op = 13) -> R[ra] <- R[rb] + c2<16..0> { 2's comp., sign ext. }:

Concrete RTN	**Control Sequence**
MA <- PC: C <- PC + 4	PCout, MAin, Inc4, Cin, Read
MD <- M[MA]; PC <- C;	Cout, PCin, Wait
IR <- MD;	MDout, IRin
A <- R[rb];	Grb, Rout,, Ain
C <-A + c2<16..0>	c2out, ADD, Cin
R[ra] <- C	Cout, Gra, Rin, End

=>c2out을 해야지 32bit로 sign extension이 된다.

3) st (: = op = 3) -> M[disp] <- R[ra]:

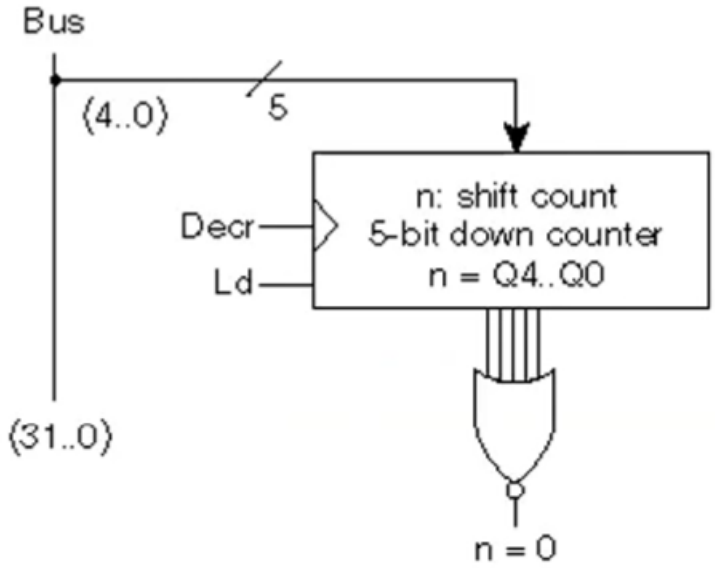
disp<31..0> := ((rb=0) -> c2<16..0> {sign ext.}:
(rb != 0) -> R[rb] + c2<16..0> {sign extend, 2's comp.] }:

Concrete RTN	**Control Sequence**
instruction fetch	
A<- (rb = 0) -> 0 : rb != 0 -> R[rb];	Grb, BAout, Ain
C<- A + c2<16..0> {sign ext. }	c2out, ADD, Cin
MA<- C;	Cout, MAin
MD <- R[ra];	Gra, Rout, MDin, Write

M[MA] <- MD; Wait, End

=>disp 계산할 때 BAout 신호를 사용한다.

The Shift Counter



-**Ld** : **bus로 부터 들어온 오른쪽 5bit를 넣는 입력 신호 역할**

-Shift right (shr) 에는 두가지 포멧이 존재한다

1) shr ra, rb, rc <- rc에 0이 들어간다

2) shr ra, rb, 10

=>**rc가 0인지 아닌지를 통해서 어떤 포멧인지 확인**할 수 있다.

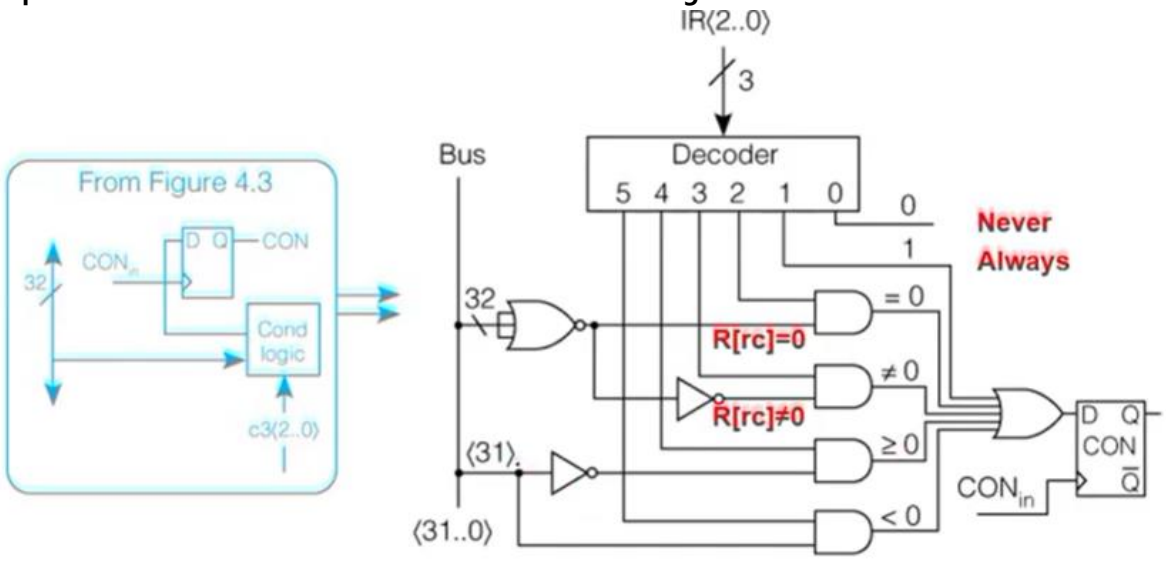
Concrete RTN	Control Sequence
instruction fetch	
n <- IR<4..0>	c1out, Ld
(n = 0) -> (n <- R[rc]<4..0>);	n=0 -> (Grc, Rout, Ld)
C <- R[rb];	Grb, Rout, **C=B**, Cin
shr (: = (n != 0) ->	n!=0 -> (Cout, **SHR**, Cin,
<C<31..0> <- 0#C<31..1>:	Decr, Goto6)
n<- n – 1; Shr));	
R[ra] <- C;	Cout, Gra, Rin, End

-C<31..0> <- 0#C<31..1>

=>C의 31 부터 1 까지를 오른쪽으로 한칸 밀고 **맨왼쪽을 0으로 채운다**.

-n에다가 집어넣을라면 Ld 신호가 필요하다

Computation of the Conditional Value CON for Branching

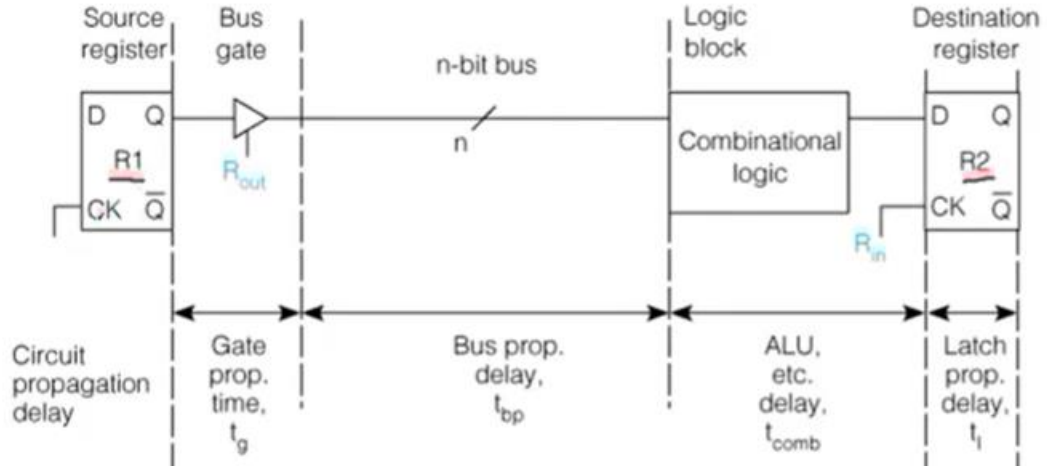


-결과값을 CON 이라는 1bit 의 레지스터에 집어 넣는다

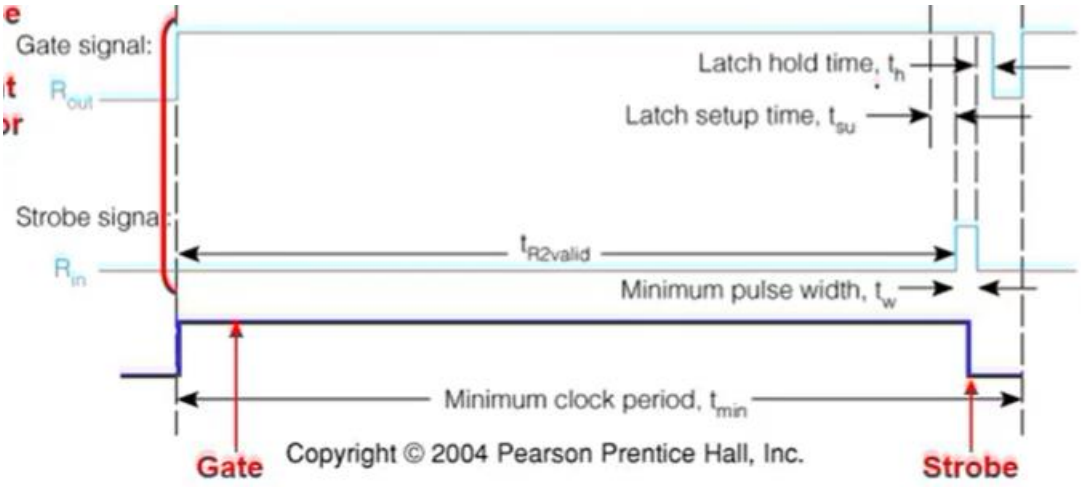
br (:= op = 8) -> (cond ->PC <- R[rb]):

Concrete RTN	Control Sequence
instruction fetch	instruction fetch
CON <- cond(R[rc]);	Grc, Rout, CONin
CON -> PC <- R[rb];	Grb, Rout, CON->PCin, End

Clocking the Data Path : Register Transfer Timing



-Latch prop delay : 값이 들어가서 안정되는데 걸리는 시간



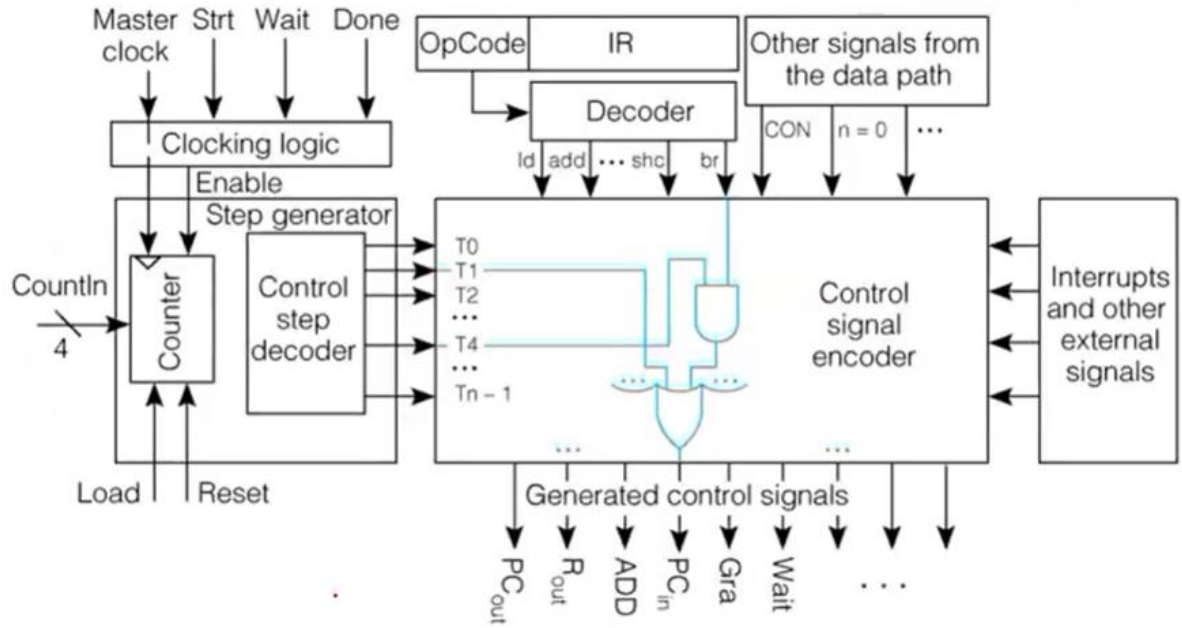
-Edge-triggered 타입은 Gate와 Strobe 신호를 표현하기 위해 1개의 신호만 필요하다 (Rout 은 rising , Rin은 falling edge)

-Latch 타입은 Gate와 Strobe 신호를 표현하기 위해 2개의 신호가 필요하다

The Control Unit

-CPU는 Data Path 부분 (지금까지 한 부분) 과 Control Unit 으로 나누어져 있다.

-적당한 순서에 맞춰서 Control signal 들을 발생시키는 역할



1) Control Signal Encorder 부분

Synthesizing Control Signal Encorder Logic

Step	Control Sequence
T0.	PC_{out}, MA_{in}, Inc4, C_{in}, Read
T1.	C_{out}, PC_{in}, Wait
T2.	MD_{out}, IR_{in}

add

Step	Control Sequence
T3.	Grb, R_{out}, A_{in}
T4.	Grc, R_{out}, ADD, C_{in}
T5.	C_{out}, **Gra**, R_{in}, End

addi

Step	Control Sequence
T3.	Grb, R_{out}, A_{in}
T4.	$c2_{out}$, ADD, C_{in}
T5.	C_{out}, **Gra**, R_{in}, End

st

Step	Control Sequence
T3.	Grb, BA_{out}, A_{in}
T4.	$c2_{out}$, ADD, C_{in}
T5.	C_{out}, MA_{in}
T6.	**Gra**, R_{out}, MD_{in}, Write
T7.	Wait, End

shr

Step	Control Sequence
T3.	$c1_{out}$, Ld
T4.	n=0 → (Grc, R_{out}, Ld)
T5.	Grb, R_{out}, C=B
T6.	n≠0 → (C_{out}, SHR, C_{in}, Decr, Goto7)
T7.	C_{out}, **Gra**, R_{in}, End

ex) (Control Signal) **Gra** = T5 * (add + addi) + T6 * st + T7 * shr + ...

Cout = T1 + T5 * add + ...

(T1 에서는 instruction fetch 부분이므로 무조건 공통적으로 Cout이 발생)

-전부 이런 논리식으로 작성한다음에 회로 식으로 구현한다

Use of Data Path Conditions in Control Signal Logic

add

Step	Control Sequence
T3.	Grb, R_{out}, A_{in}
T4.	**Grc**, R_{out}, ADD, C_{in}
T5.	C_{out}, Gra, R_{in}, End

addi

Step	Control Sequence
T3.	Grb, R_{out}, A_{in}
T4.	$c2_{out}$, ADD, C_{in}
T5.	C_{out}, Gra, R_{in}, End

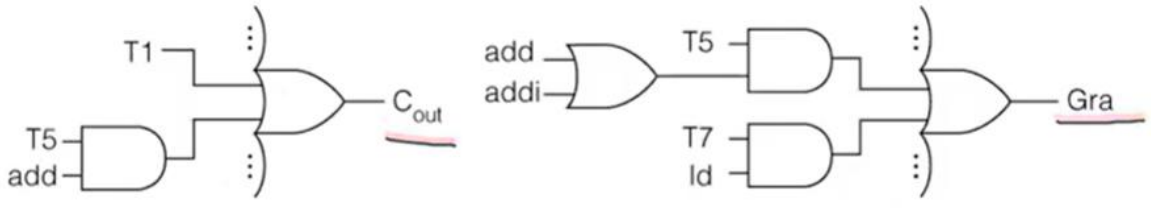
st

Step	Control Sequence
T3.	Grb, BA_{out}, A_{in}
T4.	$c2_{out}$, ADD, C_{in}
T5.	C_{out}, MA_{in}
T6.	Gra, R_{out}, MD_{in}, Write
T7.	Wait, End

shr

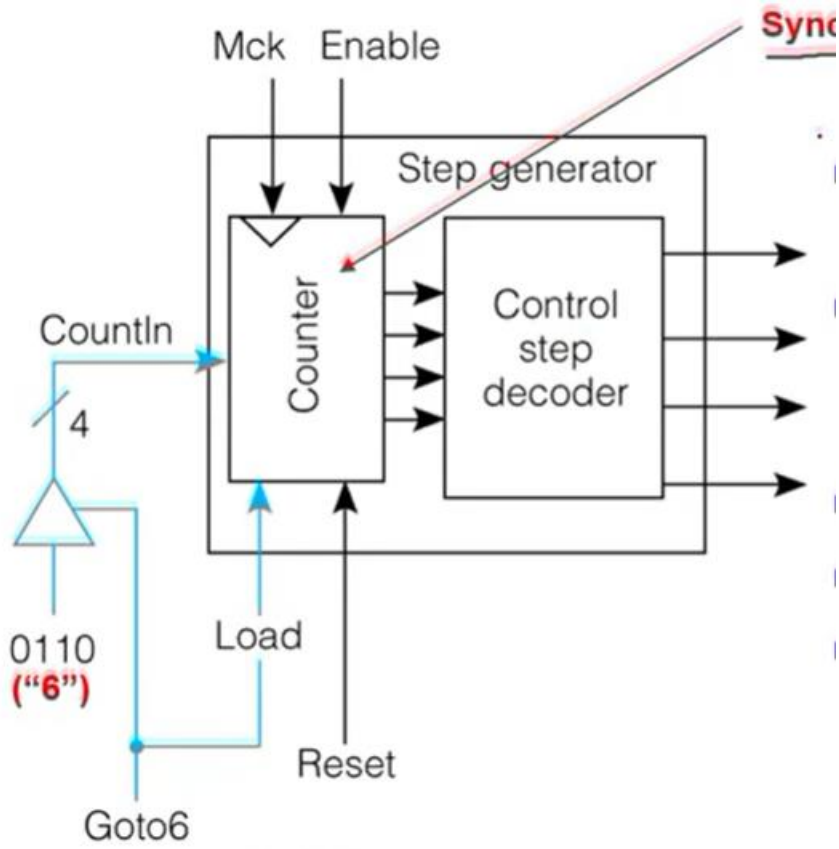
Step	Control Sequence
T3.	$c1_{out}$, Ld
T4.	**n=0 → (Grc, R_{out}, Ld)**
T5.	Grb, R_{out}, C=B
T6.	n≠0 → (C_{out}, SHR, C_{in}, Decr, Goto7)
T7.	C_{out}, Gra, R_{in}, End

ex) **Grc** = T4 * add + T4 * (n = 0) * shr +...



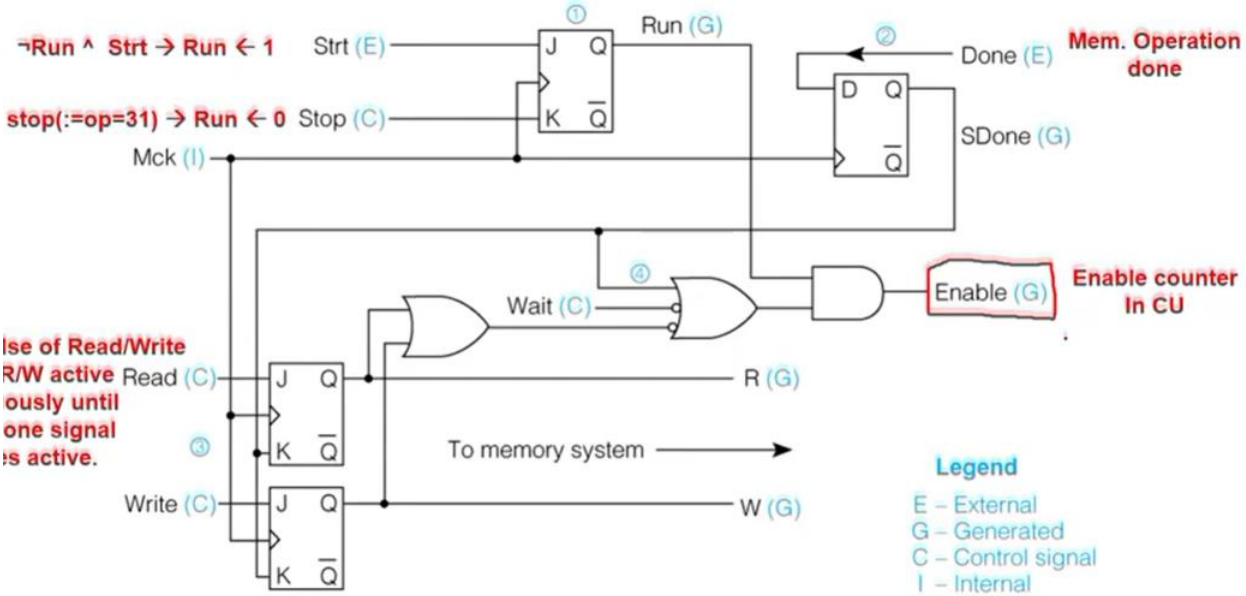
=>결과적으로 Control Signal Encorder 부분에서는 회로를 이렇게 만들면 된다

Branching in the Control Unit



-time step은 T0 에서 T15 까지 간다고 하면, Control step decorder는 4 to 16 decorder가 된다.
-Reset은 Counter를 0으로 초기화 하는 신호이다.
-만약에, T0만 1이면 instruction fetch를 하라는 신호이다.
-Goto6는 shift counter에서 쓰이는 컨트롤 신호인데, 0110(6)이 흘러가서 카운터에 흘러간다.
=>T6만 active 상태가 된다.
-실질적으로 Clocking logic 에서 enable 신호를 활성화 해야지 time step이 활성화 된다.

Clocking logic : start, stop and memory synchronization



-Strt : 지금 Run 상태가 아니고 Strt 신호가 들어오면 Run을 활성화 한다.
-Stop : K에 1이 들어가면 stop
-Enable이 1이 되기 위해서는 Run이 1이고 Done 상태이고, Wait가 0일 때
=>여기까지 배운게 One bus 에서의 CPU 설계를 끝낸 것임!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

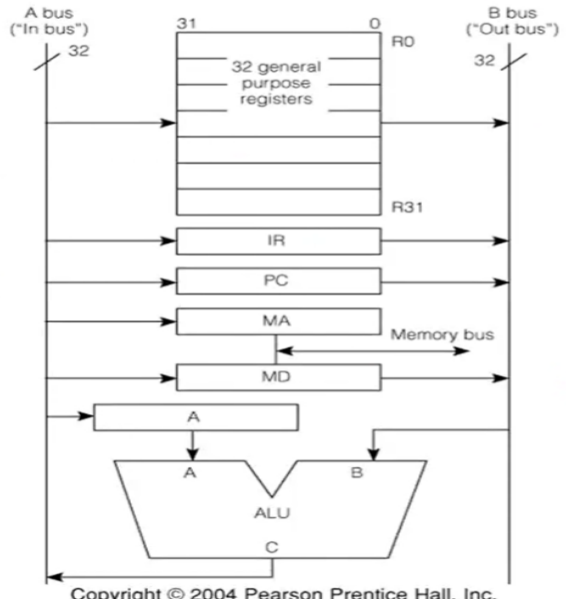
interrupts and other external signals

1) Process reset function
-리셋 신호가 들어오면 PC를 고정된 값으로 바꾼다. (이미 정해진 값, 정해진 주소에 가보면)
-control step counter를 리셋시킨다. =>T0만 1이 되어서 instruction fetch를 시작한다.
-condition code를 초기 상태로 바꾼다.
-프로세스 상태 레지스터들을 초기화 한다.
soft reset : 최소화의 변화만 한다 (PC, T-step-counter) => 동작하고 있을 때
-PC를 0으로 바꾼다.
-instruction fetch를 재시작한다.

hard reset : 더 많은 것을 초기화 한다. => 멈추었을 때 사용
-Strt : PC를 0으로 바꾼다
-32 범용 레지스터를 0으로 바꾼다

*PC를 0으로 바꾼다 => 메모리 0번지에는 target 주소가 Rom bios인 jump address가 있다
-해당 주소로 가서 Rom이 가지고 있는 초기화 프로그램을 실행한다.

2 BUS micro Architecture



-bus A는 레지스터로 들어가는 데이터를 위한 bus 이다.
-bus B는 레지스터로 부터 나오는 데이터를 위한 bus 이다.

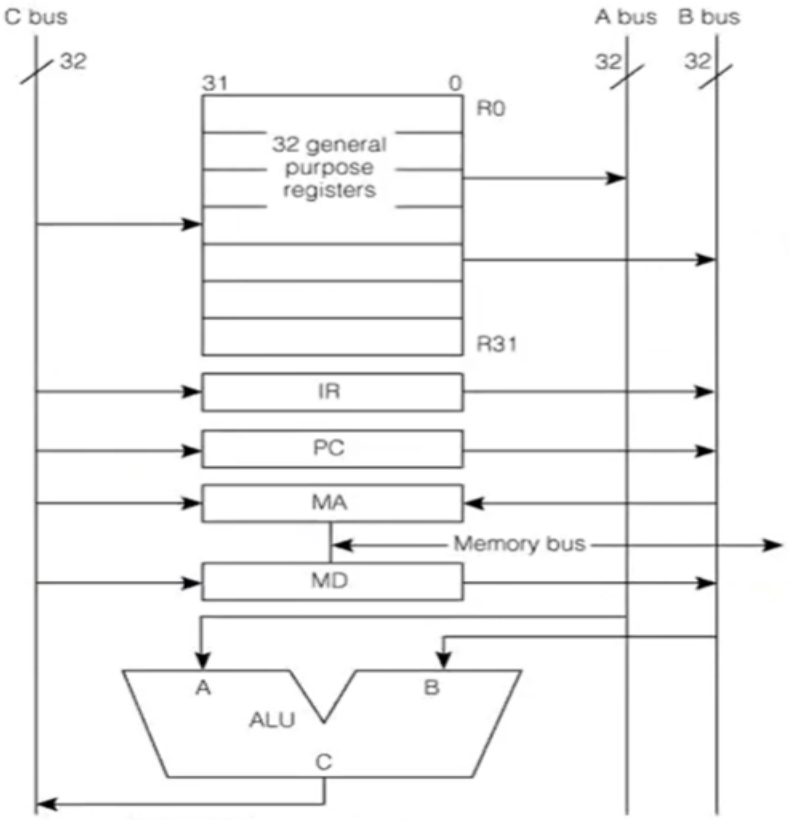
Concreete RTN and Control Sequence for 2-bus SRC add

Concrete RTN	Control Sequence
MA<-PC;	PCout, C=B, MAin, Read
PC<-PC+4: MD<- M[MA];	PCout, INC4, PCin, Wait
IR <- MD;	MDout, C=B, IRin

A <- R[rb];	Grb, Rout, C=B, Ain
R[ra] <- A + R[rc];	Grc, Rout, ADD, Sra, Rin, End

=>여기서 PC = PC + 4 가 가능한 이유는 Master-slave 구조에서 출력은 rising 하자마자 진행하고, 입력은 falling edge에서 하기때문에 하나의 신호에서 가능하기 때문이다.
(falling edge 에서 단 한번 입력을 받아 들이기 때문에, Latch 에서의 문제점은 생기지 않음)
=>MA <- PC 해서 MA에 값을 집어 넣었기 때문에 바로 Read 명령을 할 수 있다.

3 bus



SRC add instruction for the 3-bus MicroArchitecture

Concrete RTN	Control Sequence
MA <- PC: PC <- PC + 4: MD <- M[MA];	PCout, MAin, INC4, PCin, Read, wait
IR <- MD;	MDout, C = B, IRin
R[ra] <- R[rb] + R[rc];	GArc, RAout, GBrb, RBout, ADD, Sra, Rin, End