Tek çevrim (Single-Cycle) tasarımın sorunlarına çözüm

Çözüm Alternatifi: Herbir komut sınıfı için farklı çevrim süreli değişken-periyodlu clock kullanmak.

- İmkansız bir çözüm.Değişken hızlı(periyotlu) bir clock uygulama olarak teknik olarak zor

- Diğer Çözüm:
  - Daha küçük bir clock cycle kullanmak.
  - Farklı komutları, farklı sayıda clock cycleri ile gerçekleştirmek. Yani her komutun değişik fazlarınının herbirini bir cyle'lık çevrimde çözebilmek.

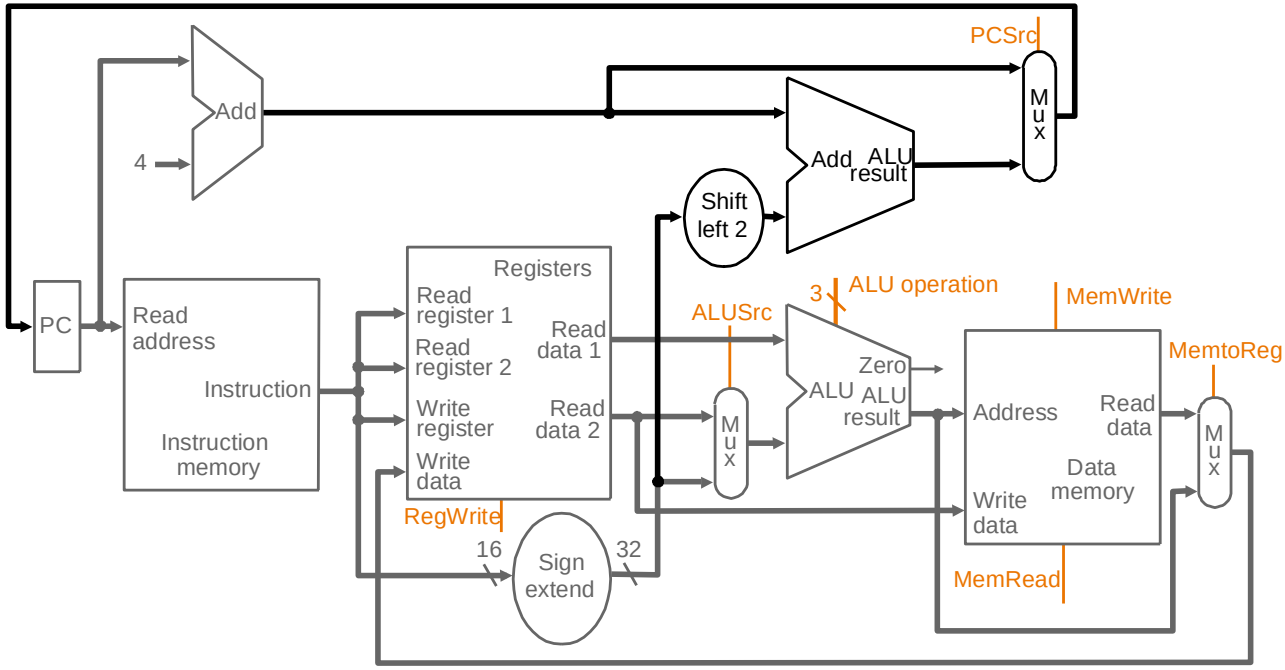
  - *Bu mümkündür : Bunun ismi Multicyle uygulamadır.*

Multicycle Uygulama

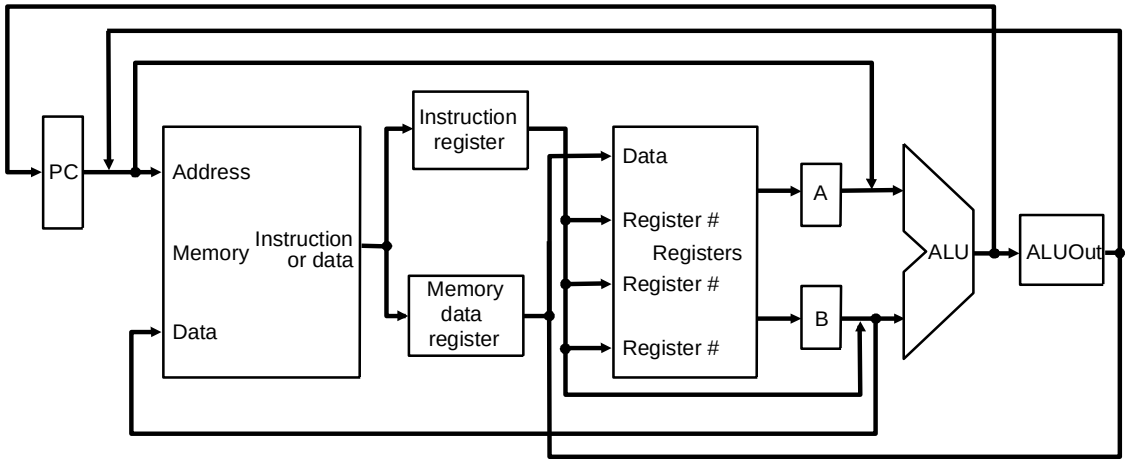
- Komutlar adımlara bölünür.
  - Herbir adım birtek clock cycle'ında yapılır.
  - Komut çevriminde herbir adım/cycle'da yapılan iş miktarı eşitlenerek dengelenir.
  - restrict each cycle to use at most once each major functional unit so that such units do not have to be replicated
  - Bir komutun farklı adımlarında, fonksiyonel birimler paylaşılarak kullanılabilir.
- Adım/Cycle arasındaki ilişki.
  - Bir cycle(çevrim) sonunda store edilen data, aynı komutun daha sonraki cycle'ında kullanılır.
  - Bu amaçla ek dahili (programcı-görünmez) reg tanıtmak gerekir
  - Programcının, store edilmiş programda; Daha sonraki komutlarda kullanılabilmesi için saklanacak data için durum elemanları, reg'ler, Memory kullanılır.

Multicycle Uygulama

- Multicycle ve singlecycle özelliklerini gösteren diyagramlar.
- Tek bir ALU kullanılır. Fazladan toplayıcı gerektirmez.
- Clock cycel'ları arasındaki kullanılacak datayı tutmak için fazladan reg'ler gerekir.

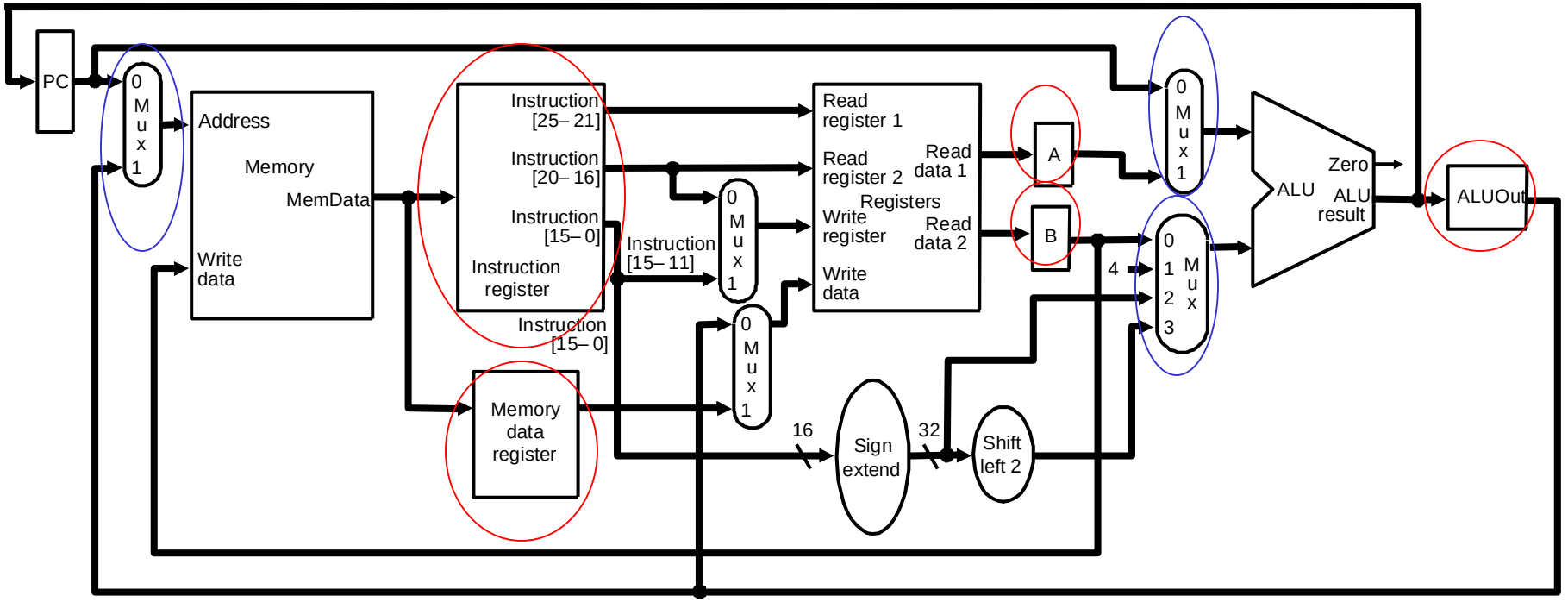


Single-cycle datapath



Multicycle datapath (high-level view)

Multicycle Datapath



Temel multicycle MIPS Datapath yapısı; R-tipi ve yük / store komutları için. kırmızı elips yeni dahili reg, mavi elips yeni mux'lardır.

Komutların adımlar halinde parçalanması

- Biz aşağıdaki avantajlar için komutları, adımlar halinde parçalayacağız.
  - Her bir adım bir clock cycle'da oluşur.
  - Herbir adımda yapılan iş miktarı yaklaşık birirbirine eşittir.
  - Her bir cycle, herbir önemli fonksiyonel birimi en fazla 1 kez kullanır. Bu yüzden böyle birimler çoğaltılamaz.
  - Fonksiyonel üniteler, aynı komutun farklı çevrimlerinde kullanılabilir.
- Bir çevrimin sonunda elde edilen data bir sonraki çevrimde kullanılabileceğinden saklanmalıdır.

Komutların adımlara bölünmesi

- Biz genel olarak bir komutu aşağıdaki yürütme adımlarına parçalayacağız. – Not1: Bütün komutların hepsi aşağıdaki adımların hepsine birden ihtiyaç duymayabilir.–Not2: Her bir adımbir clock cyle'da gerçekleşir.

1. Komutların getirilmesi ve PC'nin arttırılması (IF)
   (Instruction fetch and PC increment (**IF**))
2. Komutların çözülmesi ve register getirilmesi (ID)
   (Instruction decode and register fetch (**ID**))
3. Yürütme, hafıza adres hesabı, veya dallanmayı tamamlama (EX)
   (Execution, memory address computation, or branch completion (**EX**))
4. Hafızaya erişim veya R tipi komutun tamamlanması (MEM).
   (Memory access or R-type instruction completion (**MEM**))
5. Hafızaya okumanın tamamlanması (WB)
   (Memory read completion (**WB**))

- Herbir MIPS komutu 3-5 Cycle'da (adım) oluşur.

Adım 1: Instruction Fetch & PC Increment (**IF**) (Komutun getirilmesi & PC'nin arttırılması)

- Komutu alıp Instruction Reg (IR)'e getirmek için PC'yi kullan. Yeni komut için PC+4 yap bunu PC'ye yaz.

- RTL kullanılarak kısaca tarif edilebilir. *RTL* (*Register-Transfer Language*):

```
IR = Memory[PC];Komut Reg (IR)'e PC'de yazılı adresteki komutu getir.

PC = PC + 4; PC'ye 4 ekle
```

Adım 2: Instruction Decode and Register Fetch (**ID**) (Komutun decode edilmesi ve Registerlerin getirilmesi)

- Onlara İhtiyaç duyulduğunda rs ve rt reg'lerini oku.
  Komut bir dallanma komutu ise dallanma adresini hesap et.

- RTL:

```
A = Reg[IR[25-21]];

B = Reg[IR[20-16]];

ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

Adım 3: Execution, Address Computation or Branch Completion (**EX**) (Yürütme, Adress hesabı veya dallanmanın tamamlanması)

- ALU komut tipine bağımlı olarak aşağıdaki 4 fonksiyondan birisini başarır (yürütür).
  - memory reference (Hafıza referansını hazırlar):
    ```
    ALUOut = A + sign-extend(IR[15-0]);
    ```
  - R-type:
    ```
    ALUOut = A op B;
    ```
  - branch (komutu tamamlıyor):
    ```
    if (A==B) PC = ALUOut;
    ```
  - jump (komutu tamamalıyor):
    ```
    PC = PC[31-28] || (IR(25-0) << 2)
    ```

Adım 4: Memory access or R-type Instruction Completion (**MEM**)
(Hafızaya erişim veya R-tipi komutun tamamalanması)

- Komut tipine bağımlı olarak yeniden:
- Load ve store için hafızaya erişim sağlar.
  - load

    ```
    MDR = Memory[ALUOut];
    ```

  - store (instruction *completes*)

    ```
    Memory[ALUOut] = B;
    ```

- R-type (instructions *completes*)

```
Reg[IR[15-11]] = ALUOut;
```

Adım 5: Memory Read Completion (**WB**) (Hafıza Okumanın tamamlanması)

- Komut tipine bağımlı olarak yeniden;
- Yüklenen değeri geri yazar.(Komut tamamlama)

```
Reg[IR[20-16]]= MDR;
```

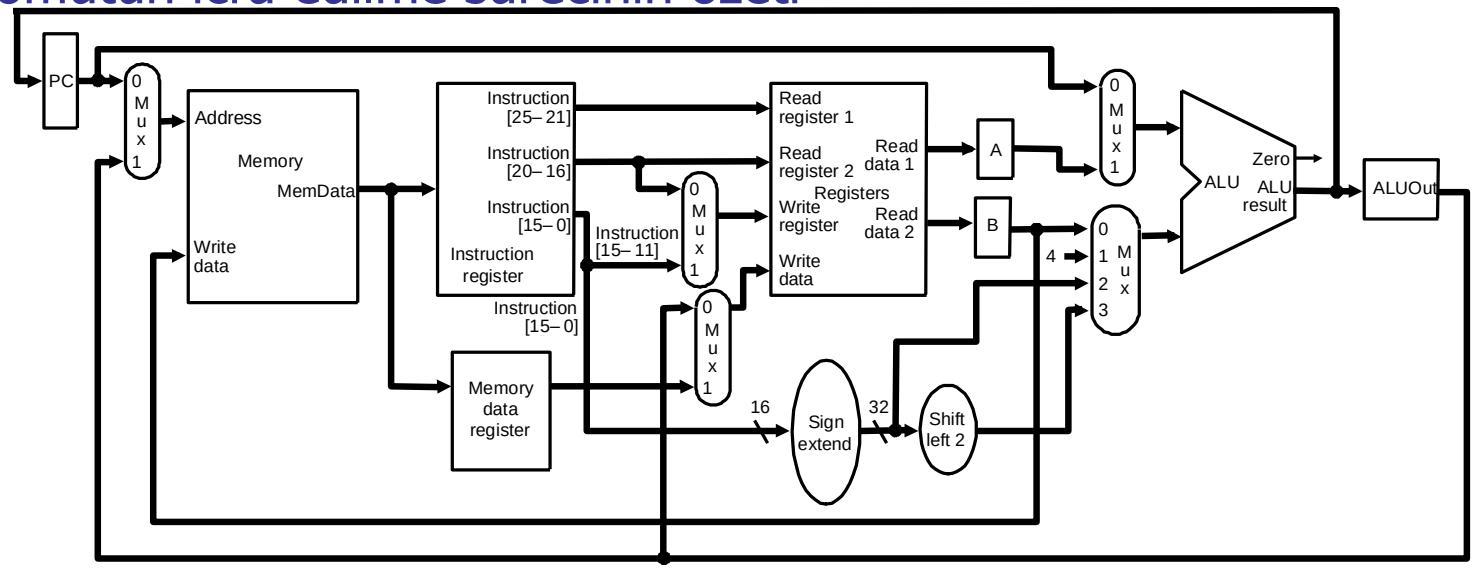
Önemli: There is no reason from a datapath (or control) point of view that Step 5 cannot be eliminated by performing

```
Reg[IR[20-16]]= Memory[ALUOut];
```

for loads in Step 4. This would eliminate the MDR as well.

The reason this is not done is that, to keep steps balanced in length, the design restriction is to allow each step to contain *at most* one ALU operation, or one register access, or one memory access.

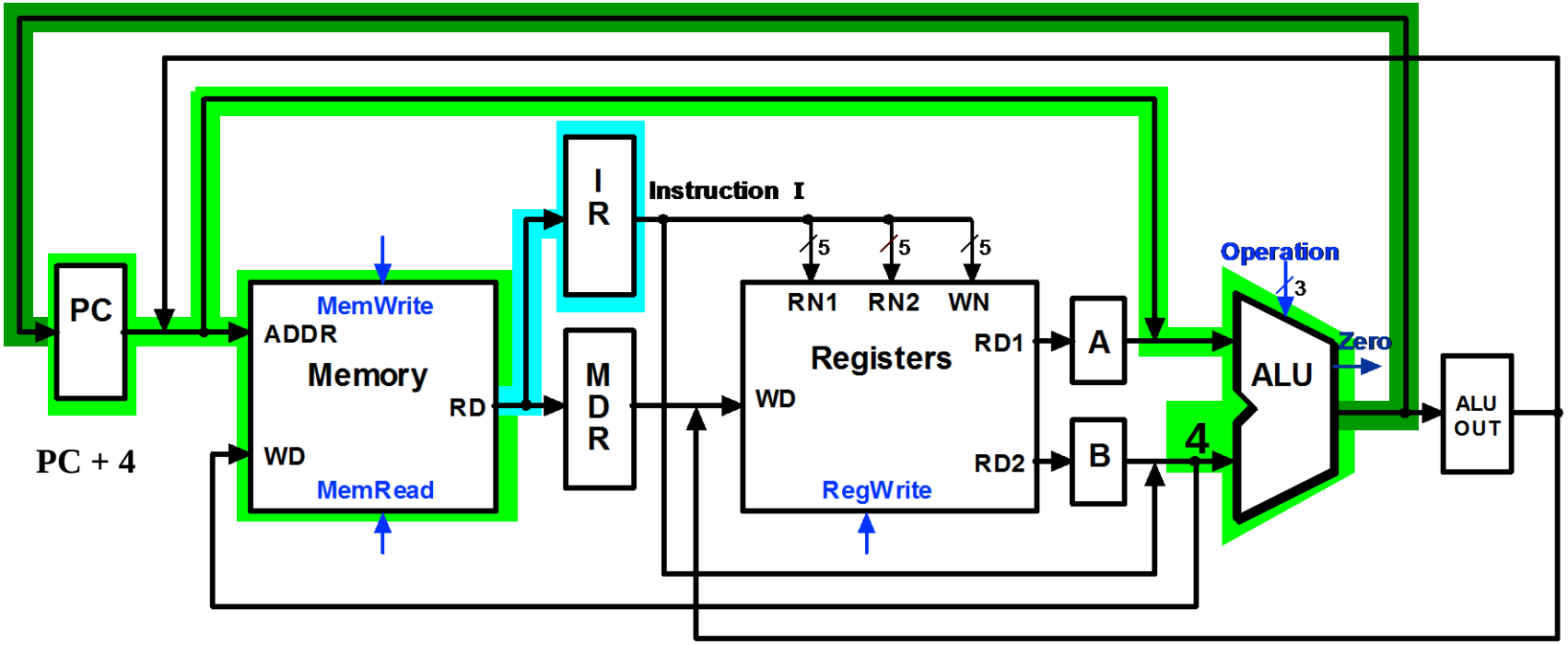
Bir komutun icra edilme sürecinin özeti



Step	Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
1: IF	Instruction fetch	IR = Memory[PC]			
		PC = PC + 4			
2: ID	Instruction decode/register fetch	A = Reg [IR[25-21]]			
		B = Reg [IR[20-16]]			
		ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
3: EX	Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
4: MEM	Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
5: WB	Memory read completion		Load: Reg[IR[20-16]] = MDR		

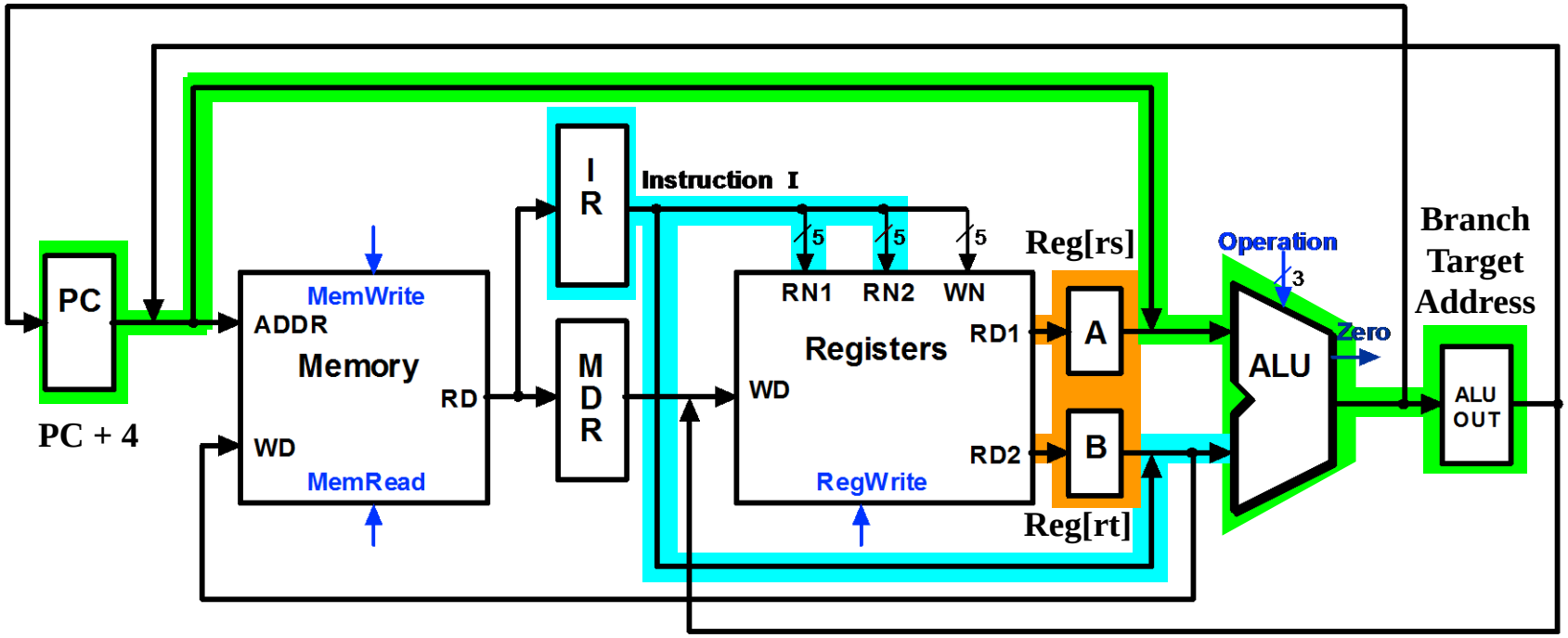
Multicycle Yürütme Adım (1): Instruction Fetch (Komutun getirilmesi)

```
IR = Memory[PC];
PC = PC + 4;
```



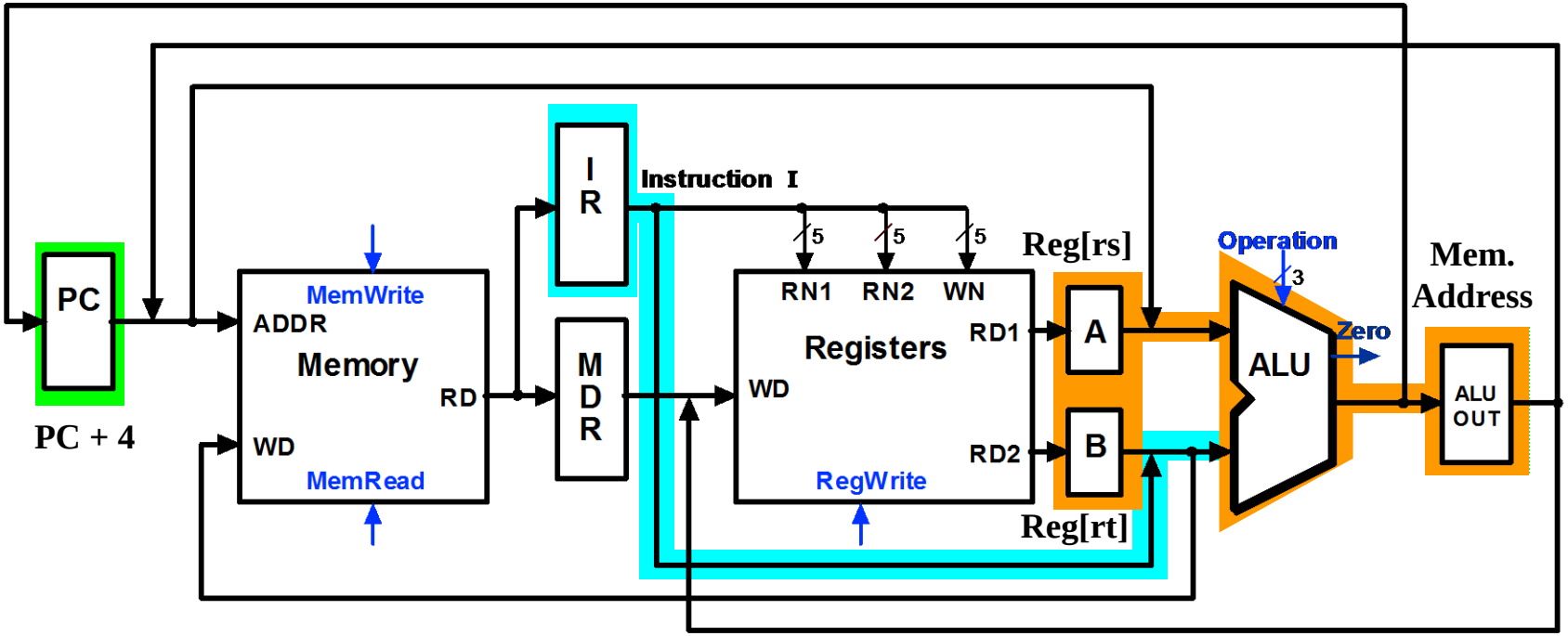
Multicycle Execution Step (2): Instruction Decode & Register Fetch

```
A = Reg[IR[25-21]];                    (A = Reg[rs])
B = Reg[IR[20-15]];                    (B = Reg[rt])
ALUOut = (PC + sign-extend(IR[15-0]) << 2)
```



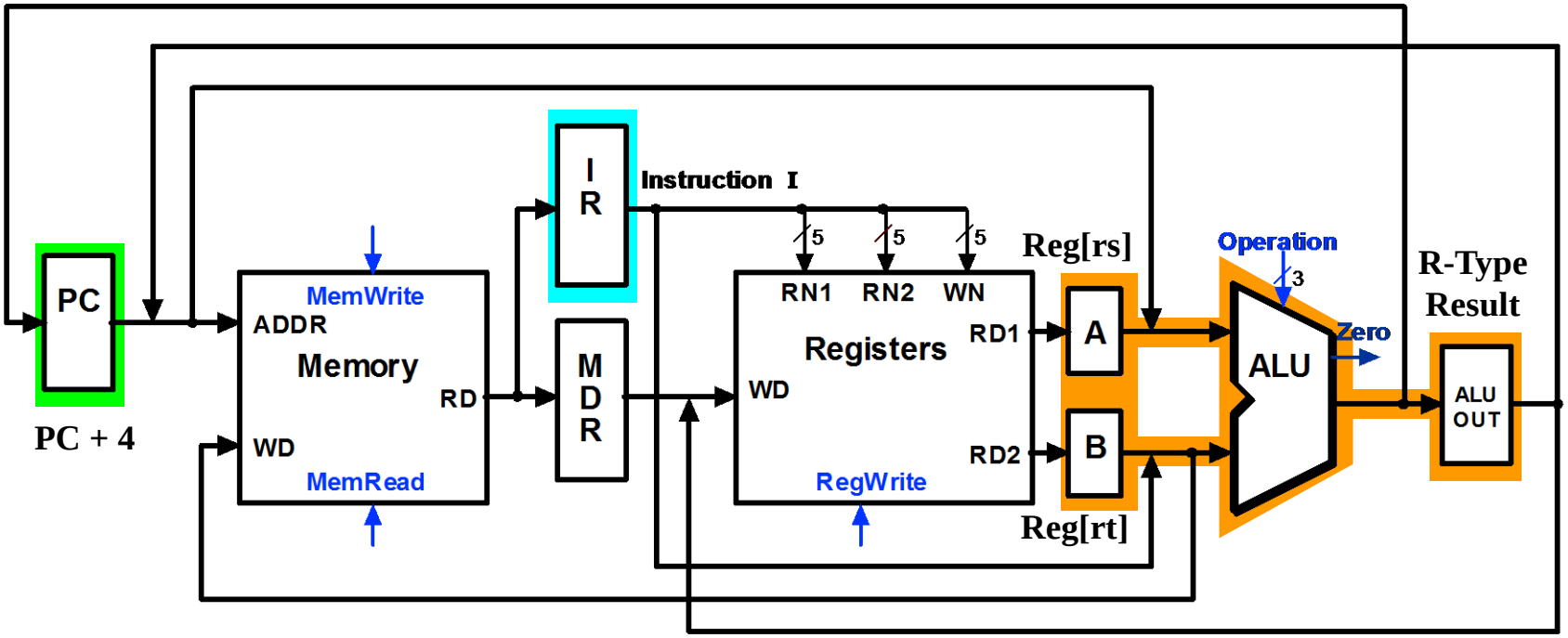
Multicycle Execution Step (3): Memory Reference Instructions

```
ALUOut = A + sign-extend(IR[15-0]);
```



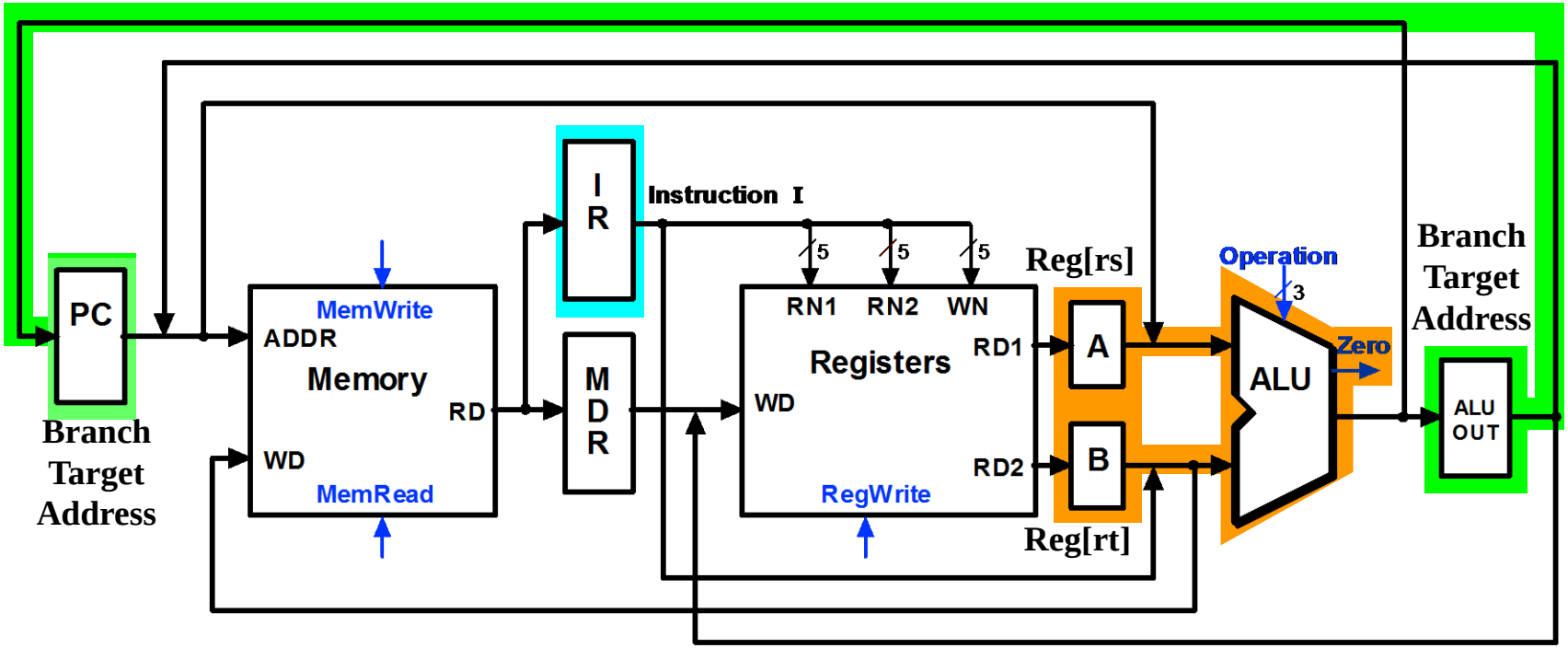
Multicycle Execution Step (3): ALU Instruction (R-Type)

```
ALUOut = A op B
```



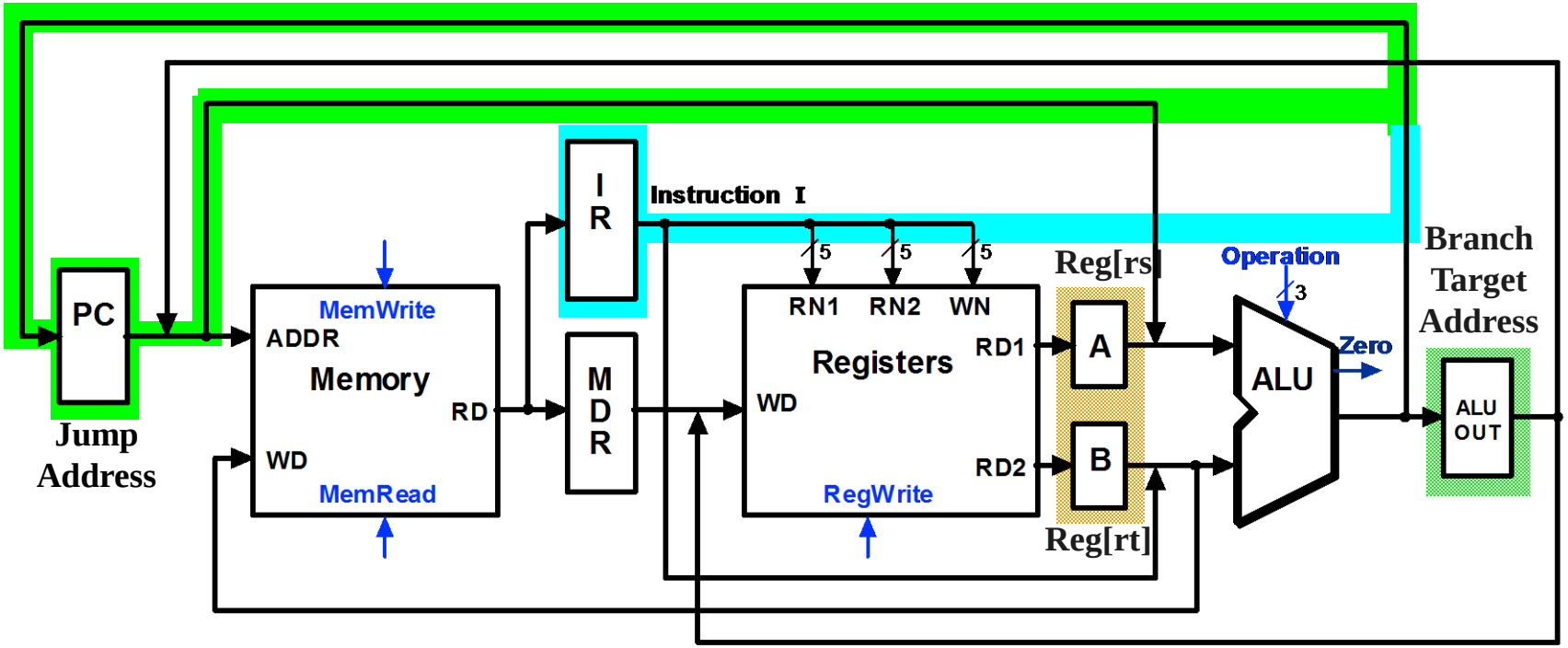
Multicycle Execution Step (3): Branch Instructions

```
if (A == B) PC = ALUOut;
```



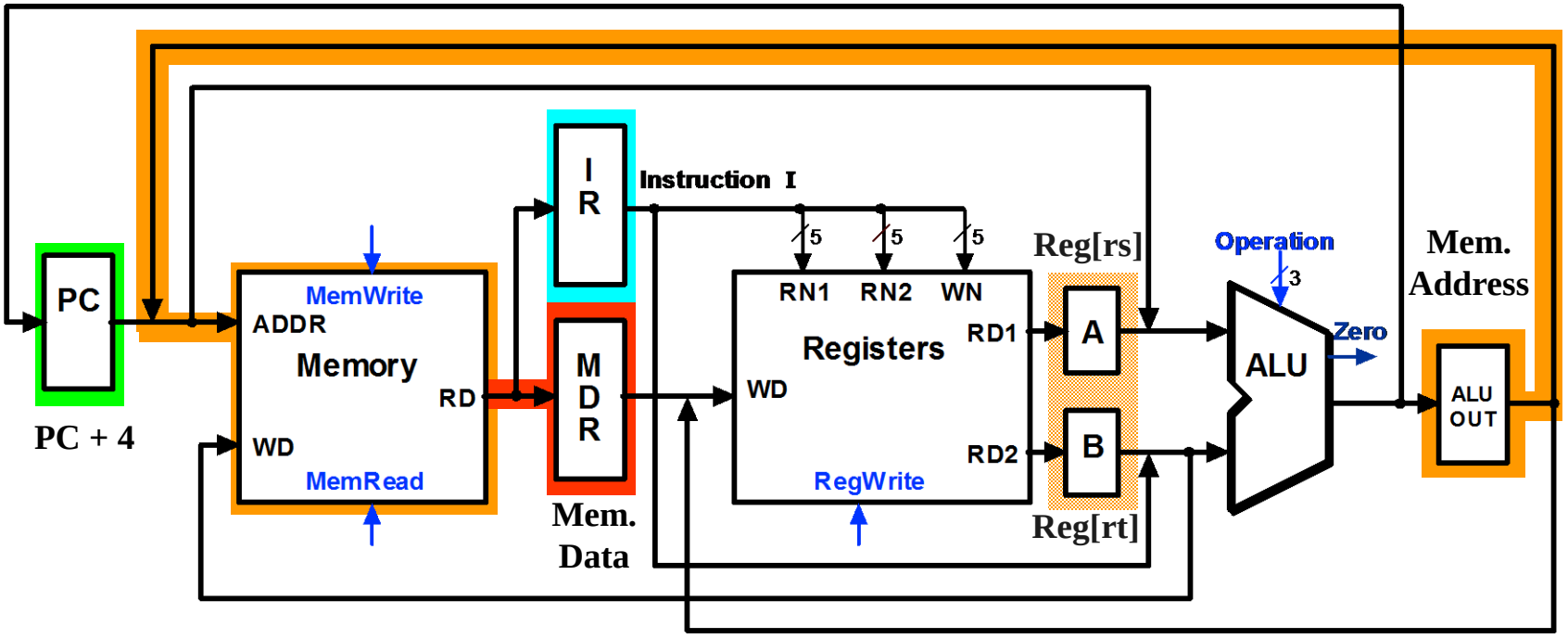
Multicycle Execution Step (3): Jump Instruction

```
PC = PC[31-28] concat (IR[25-0] << 2)
```



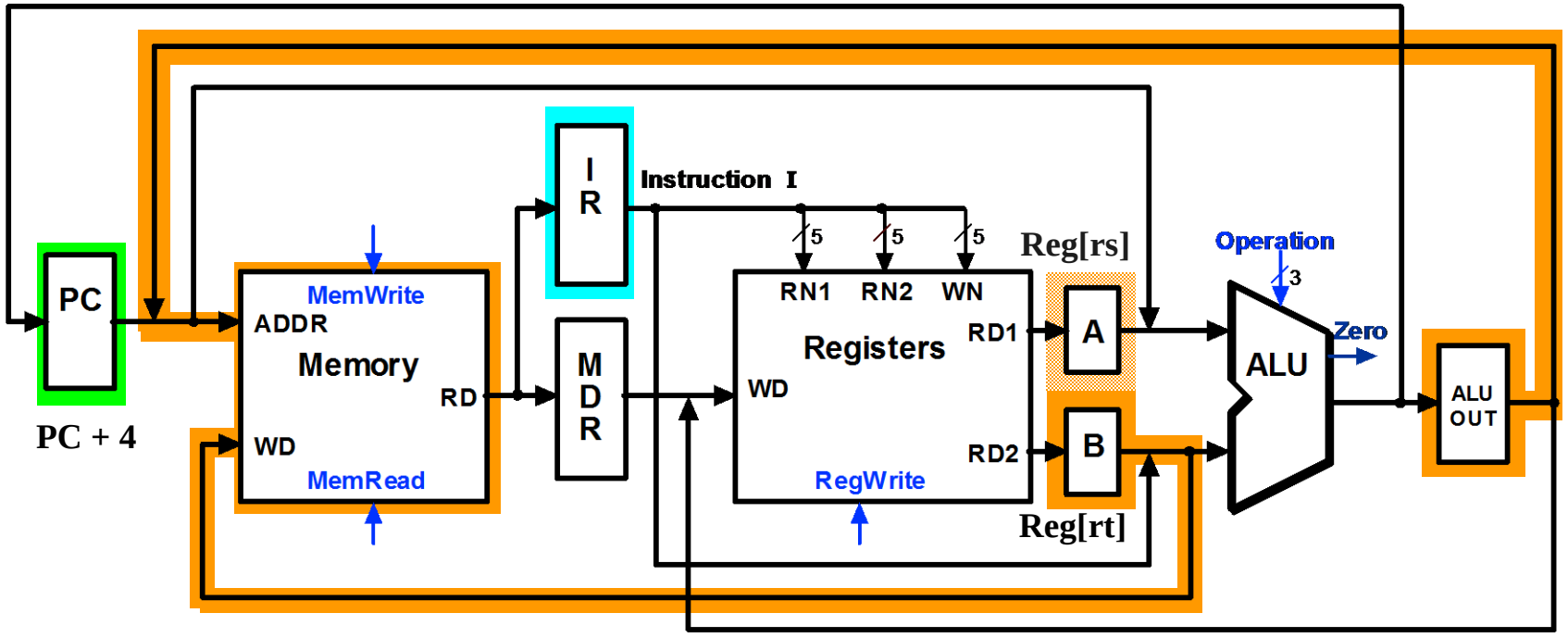
Multicycle Execution Step (4): Memory Access - Read (`lw`)

```
MDR = Memory[ALUOut];
```



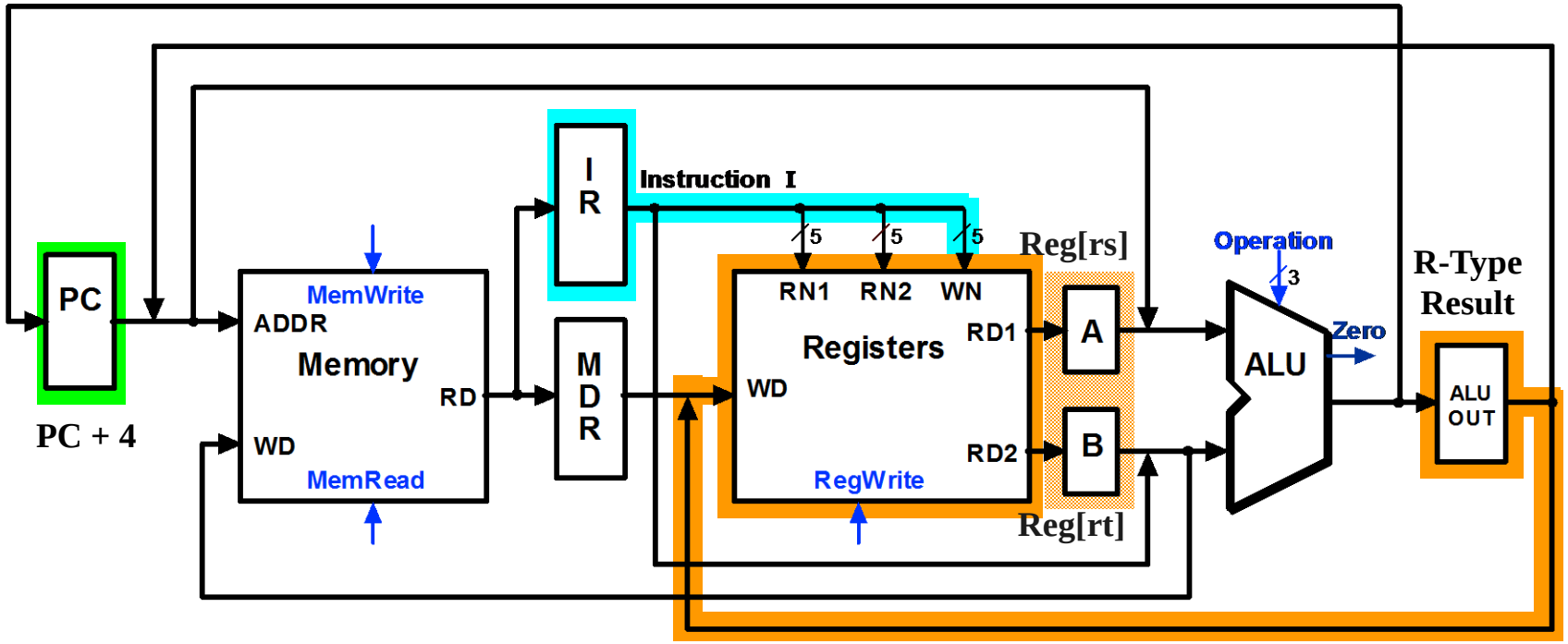
Multicycle Execution Step (4): Memory Access - Write (`sw`)

```
Memory[ALUOut] = B;
```



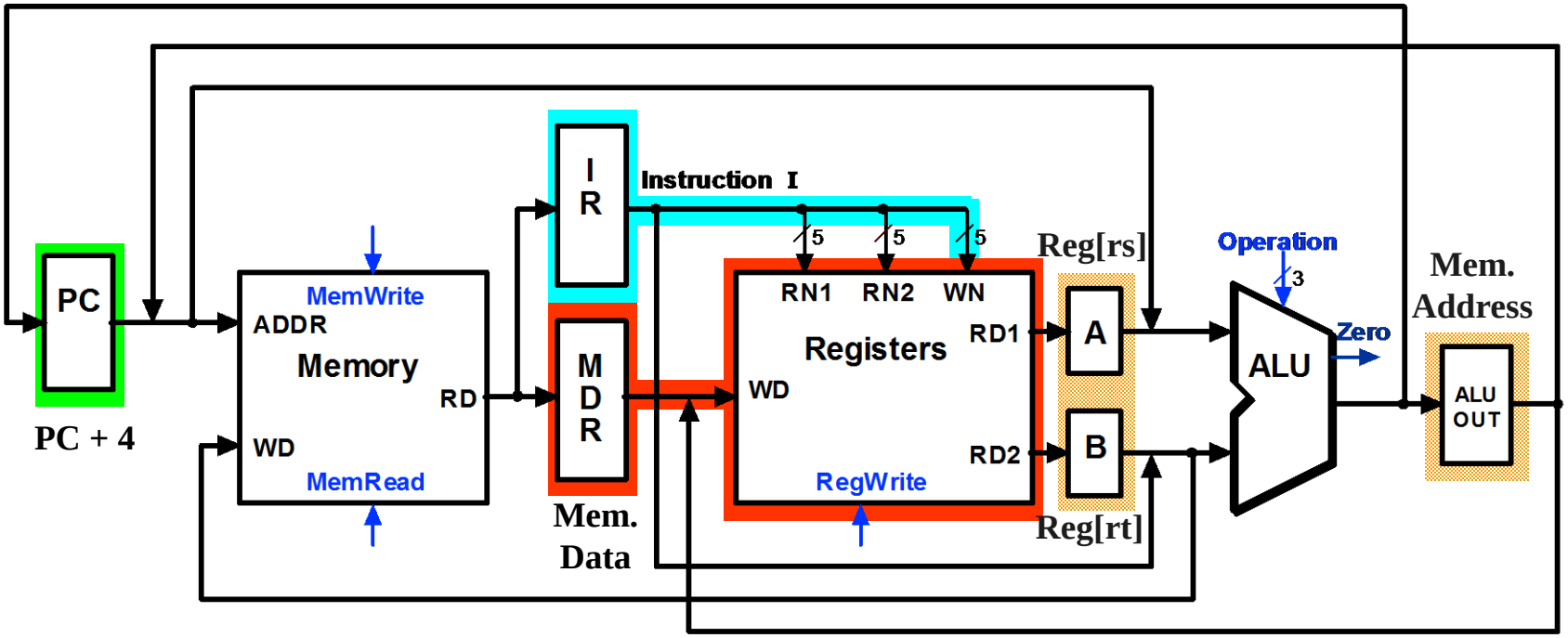
Multicycle Execution Step (4): ALU Instruction (R-Type)

```
Reg[IR[15:11]] = ALUOUT
```

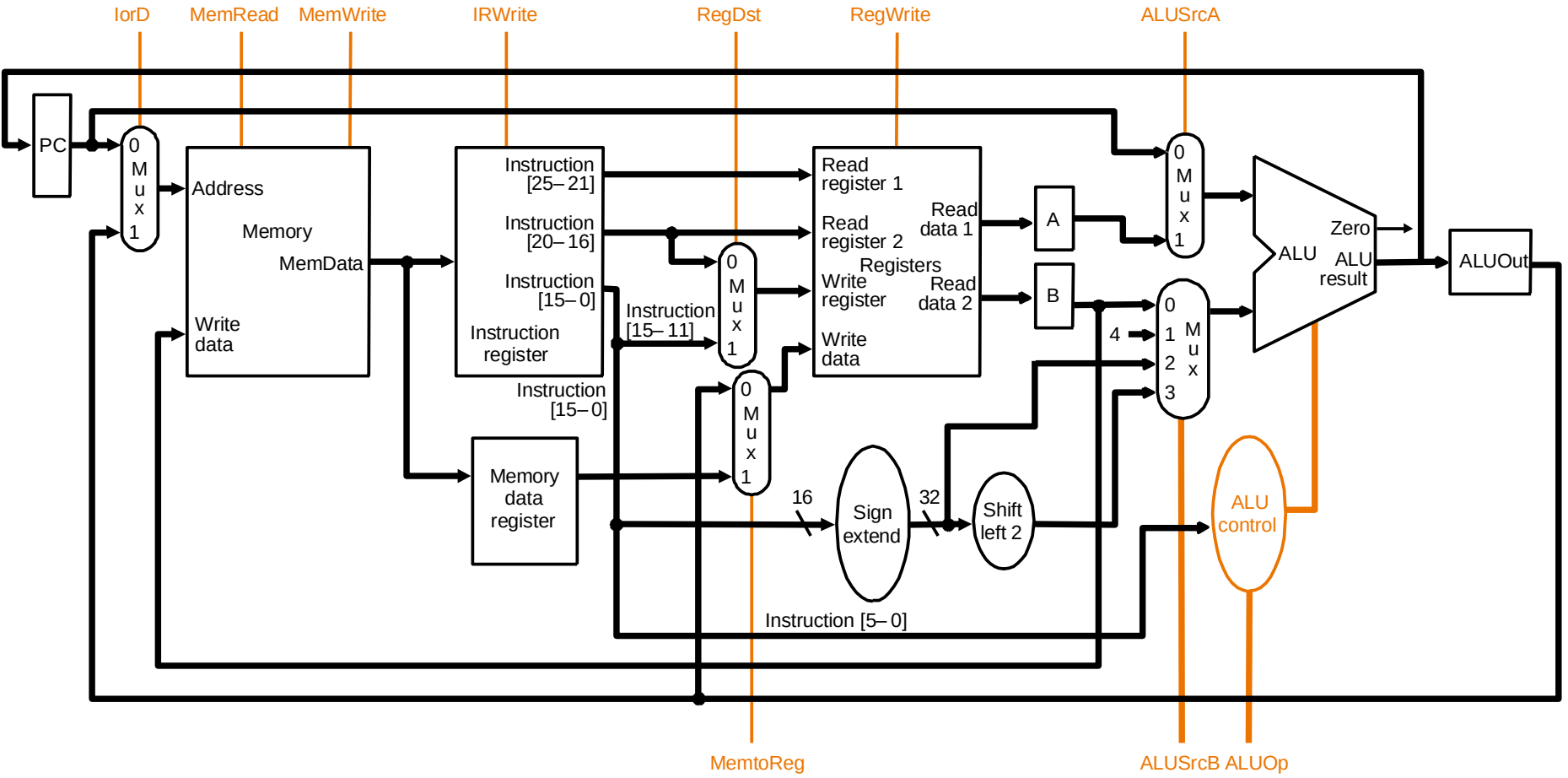


Multicycle Execution Step (5): Memory Read Completion (`lw`)

```
Reg[IR[20-16]] = MDR;
```

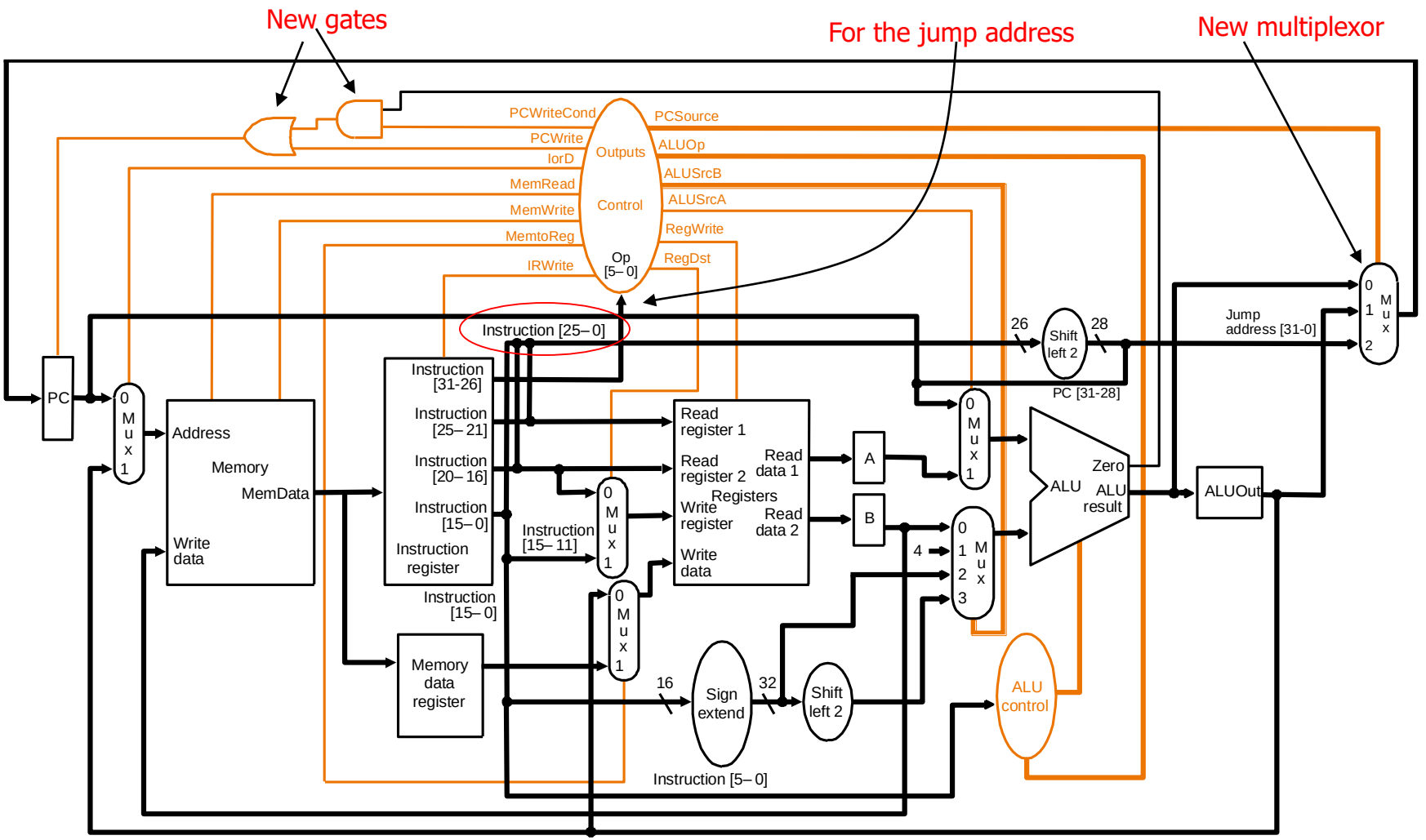


Multicycle Datapath with Control I



… with control lines and the ALU control block added – *not all* control lines are shown

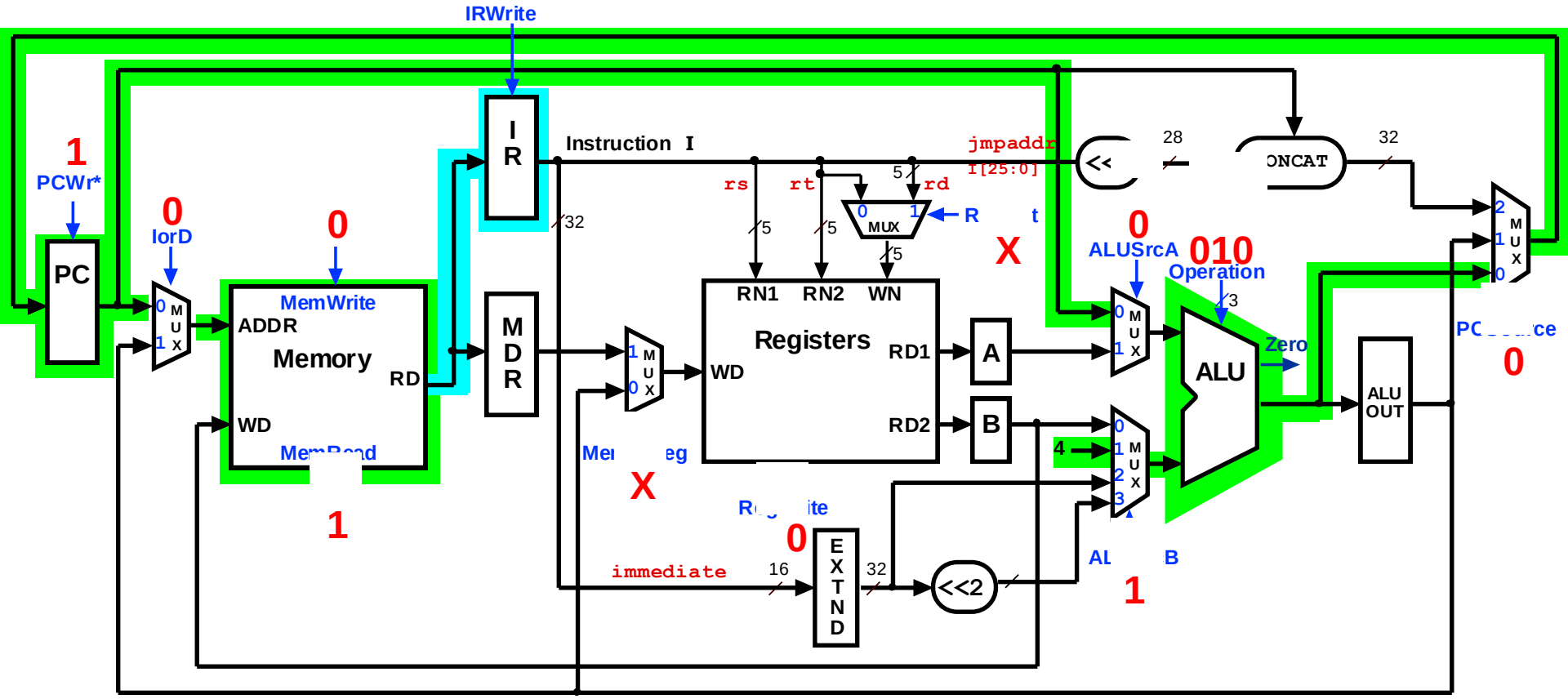
Multicycle Datapath with Control II



Complete multicycle MIPS datapath (with branch and jump capability) and showing the main control block and all control lines

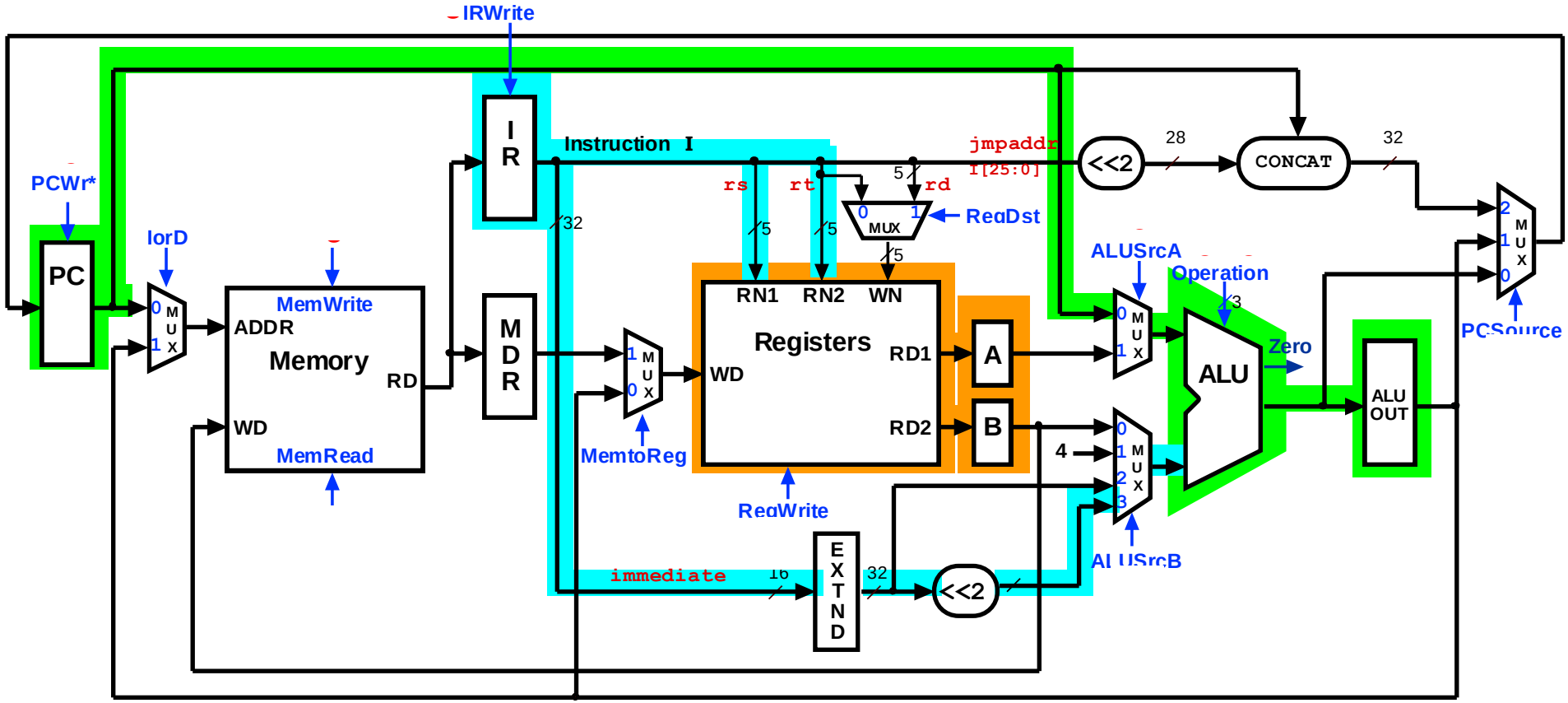
Multicycle Control Step (1): Fetch

```
IR = Memor  PC];
PC = PC + 4;
```



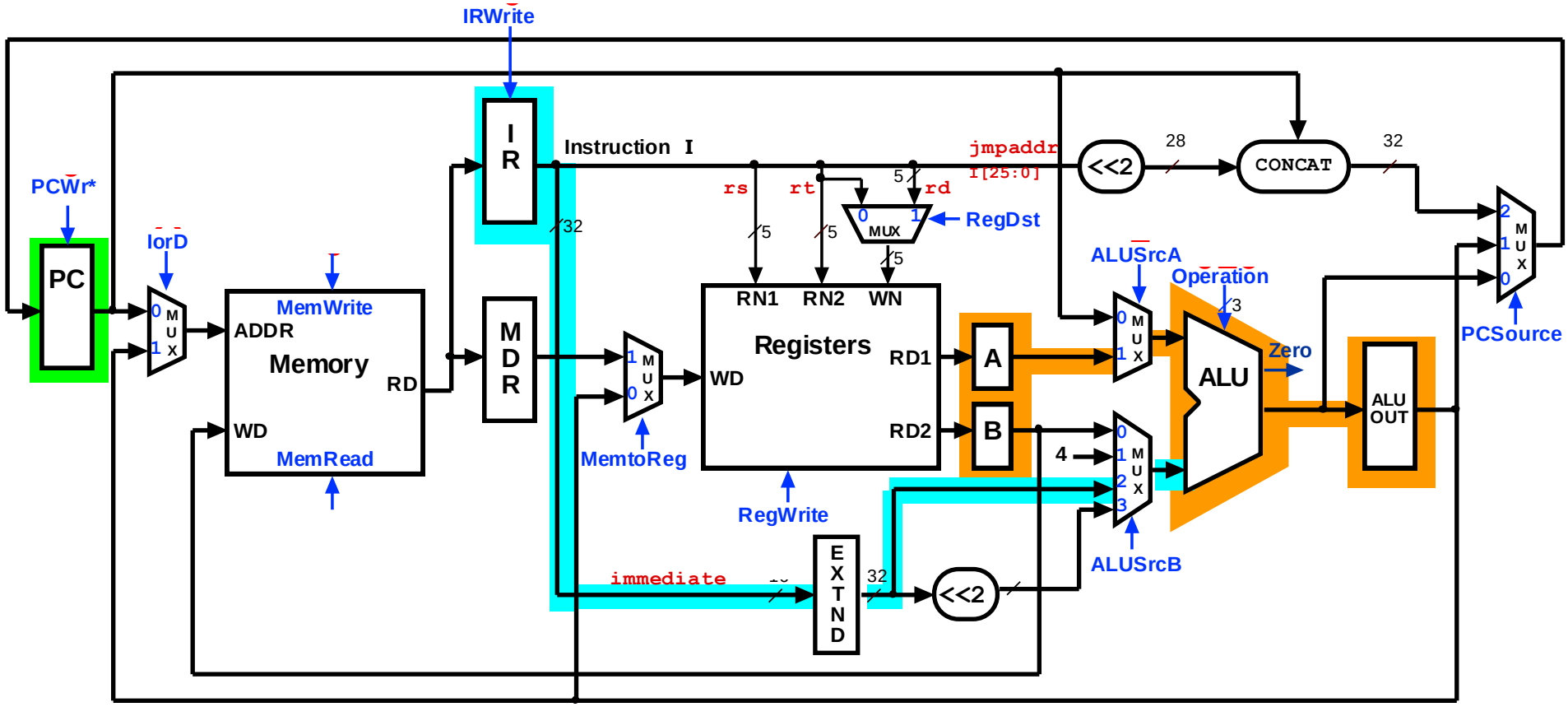
Multicycle Control Step (2): Instruction Decode & Register Fetch

```
A = Reg[IR[25-21]];          (A = Reg[rs])
B = Reg[IR[20-15]];          (B = Reg[rt])
ALUOut = (PC + sign-extend(IR[15-0]) << 2);
```



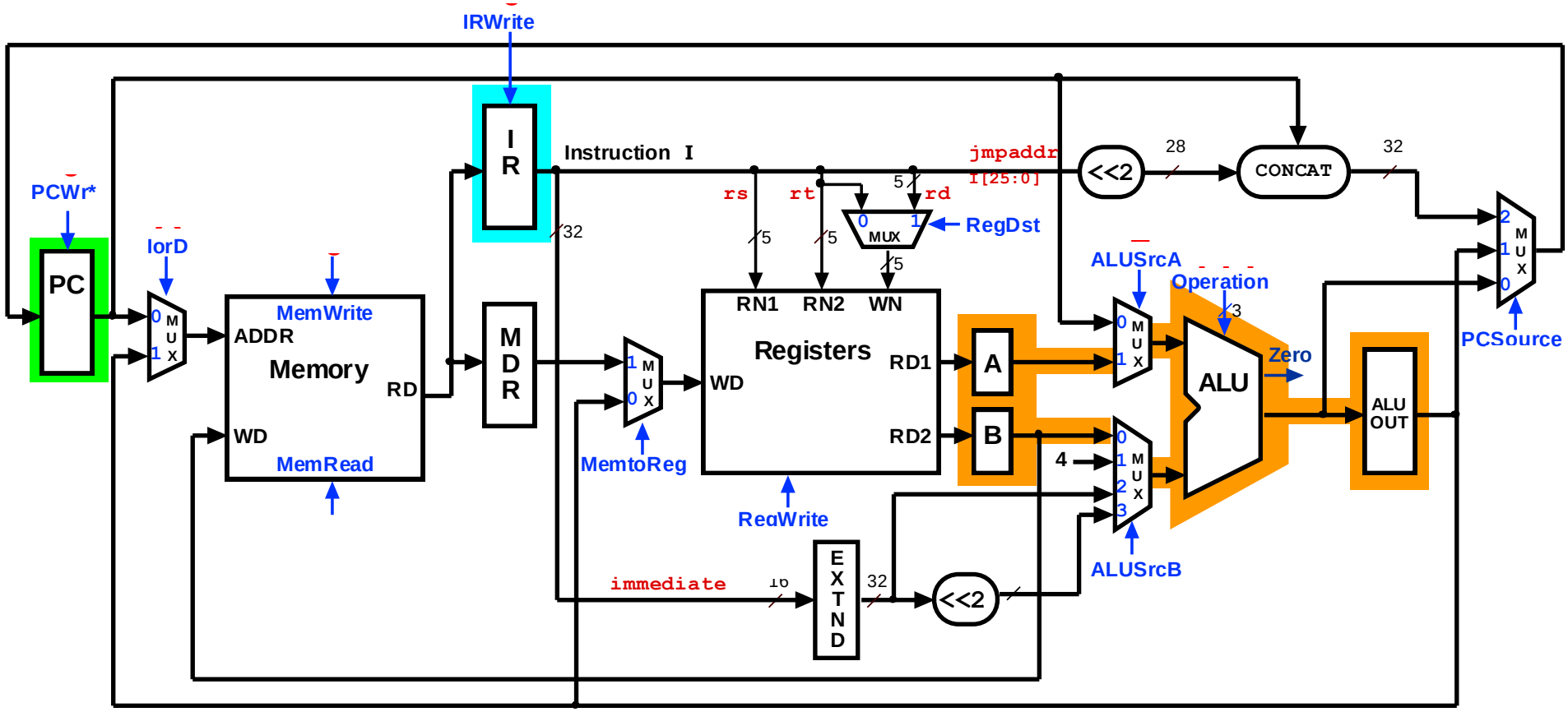
Multicycle Control Step (3): Memory Reference Instructions

```
ALUOut = A + sign-extend(IR[15-0]);
```



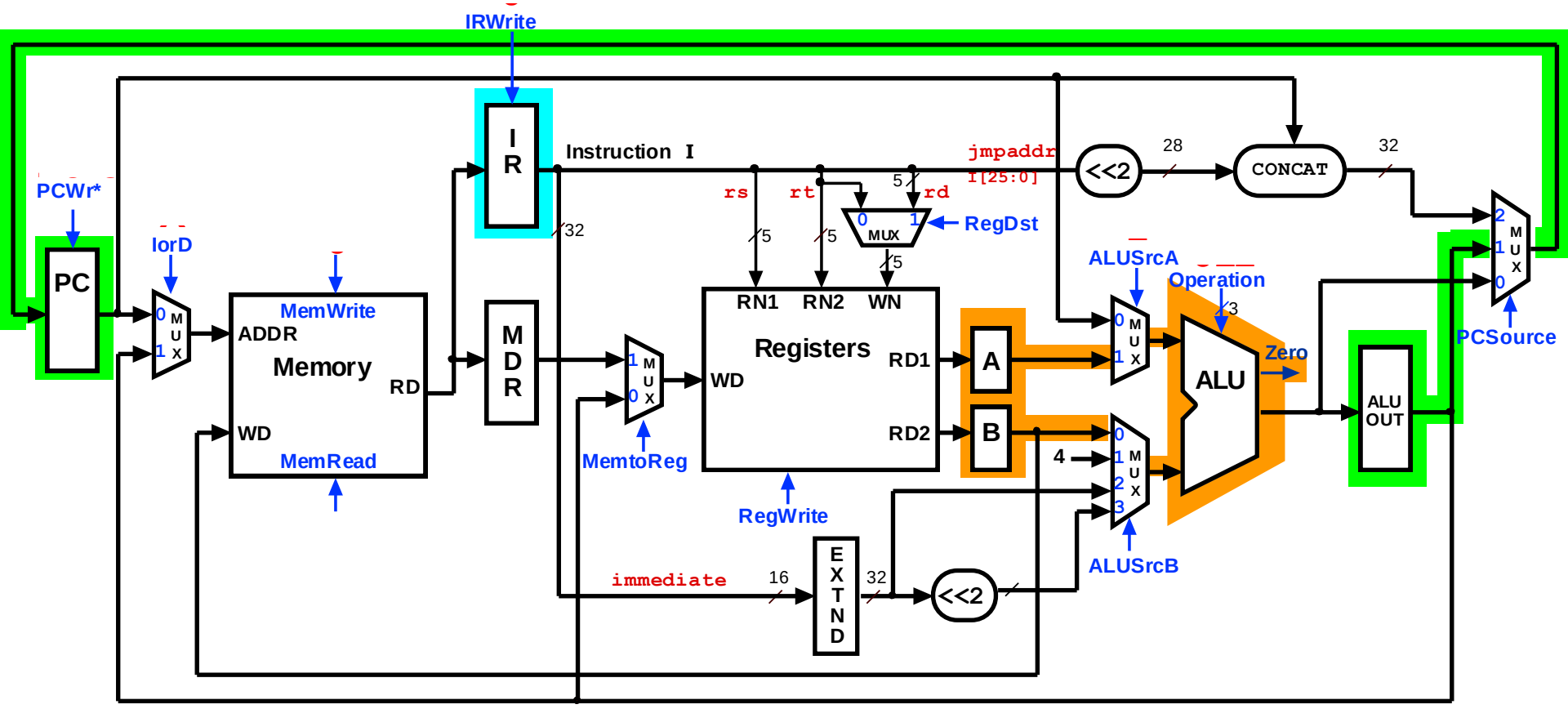
Multicycle Control Step (3): ALU Instruction (R-Type)

```
ALUOut = A   p B;
```



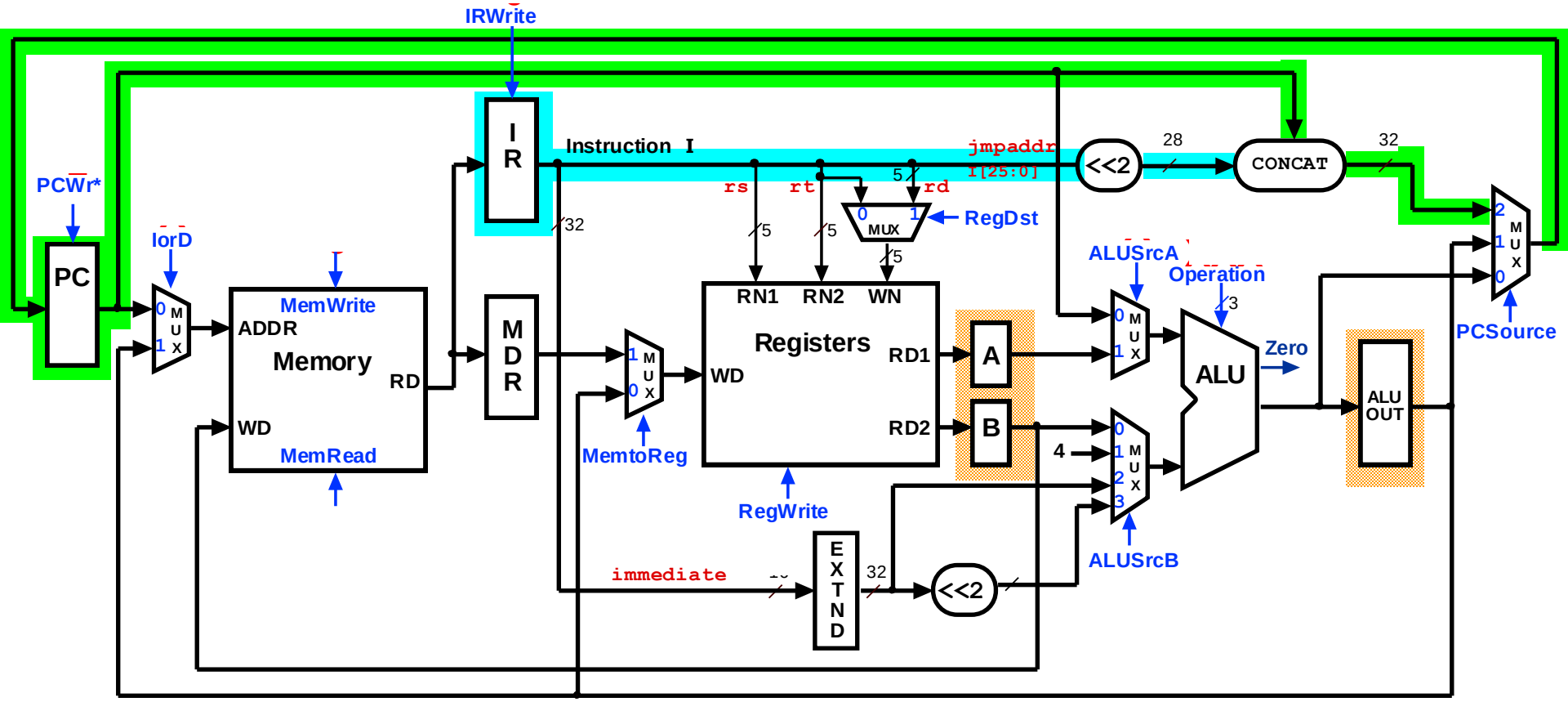
Multicycle Control Step (3): Branch Instructions

```
if (A == B` PC = ALUOut;
```



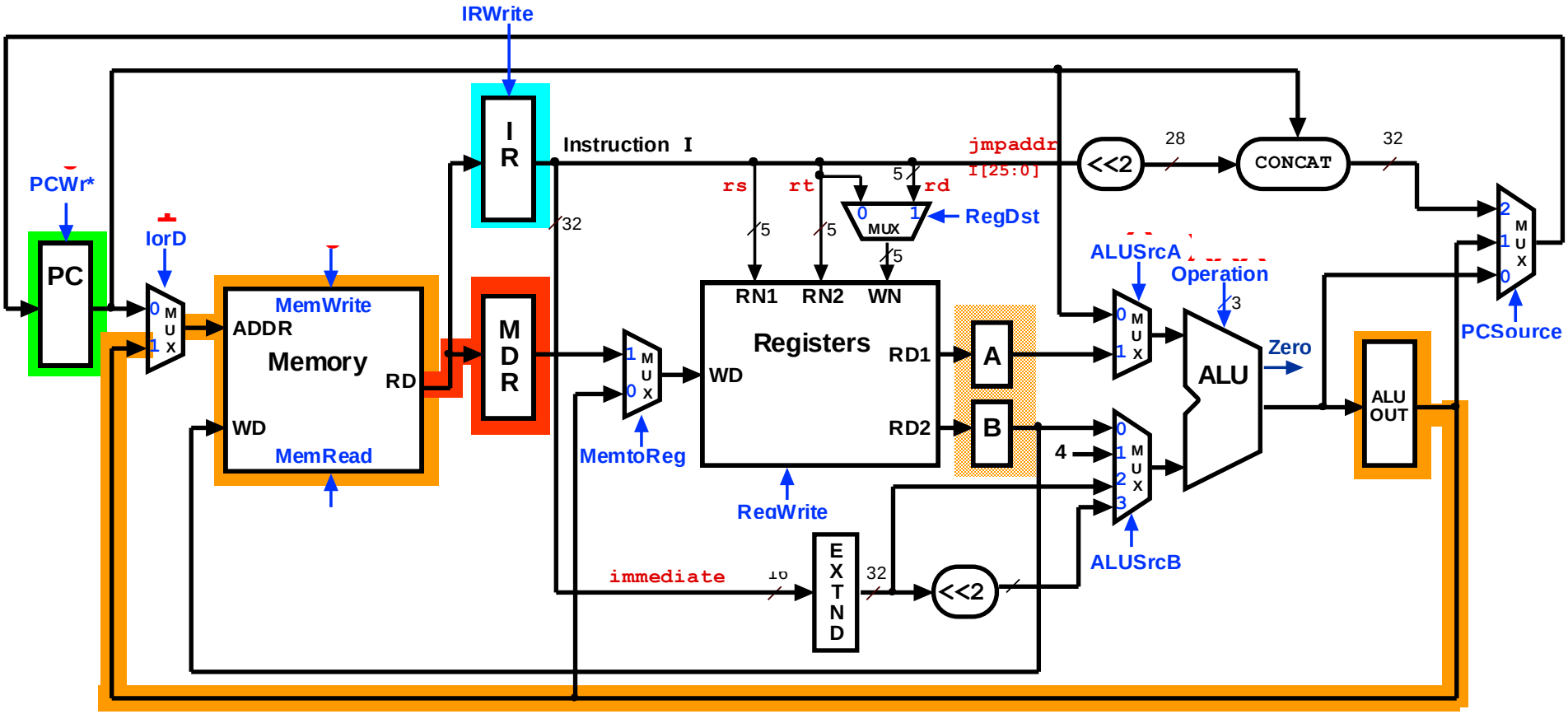
Multicycle Execution Step (3): Jump Instruction

```
PC = PC[21-28] concat (IR[25-0] << 2);
```



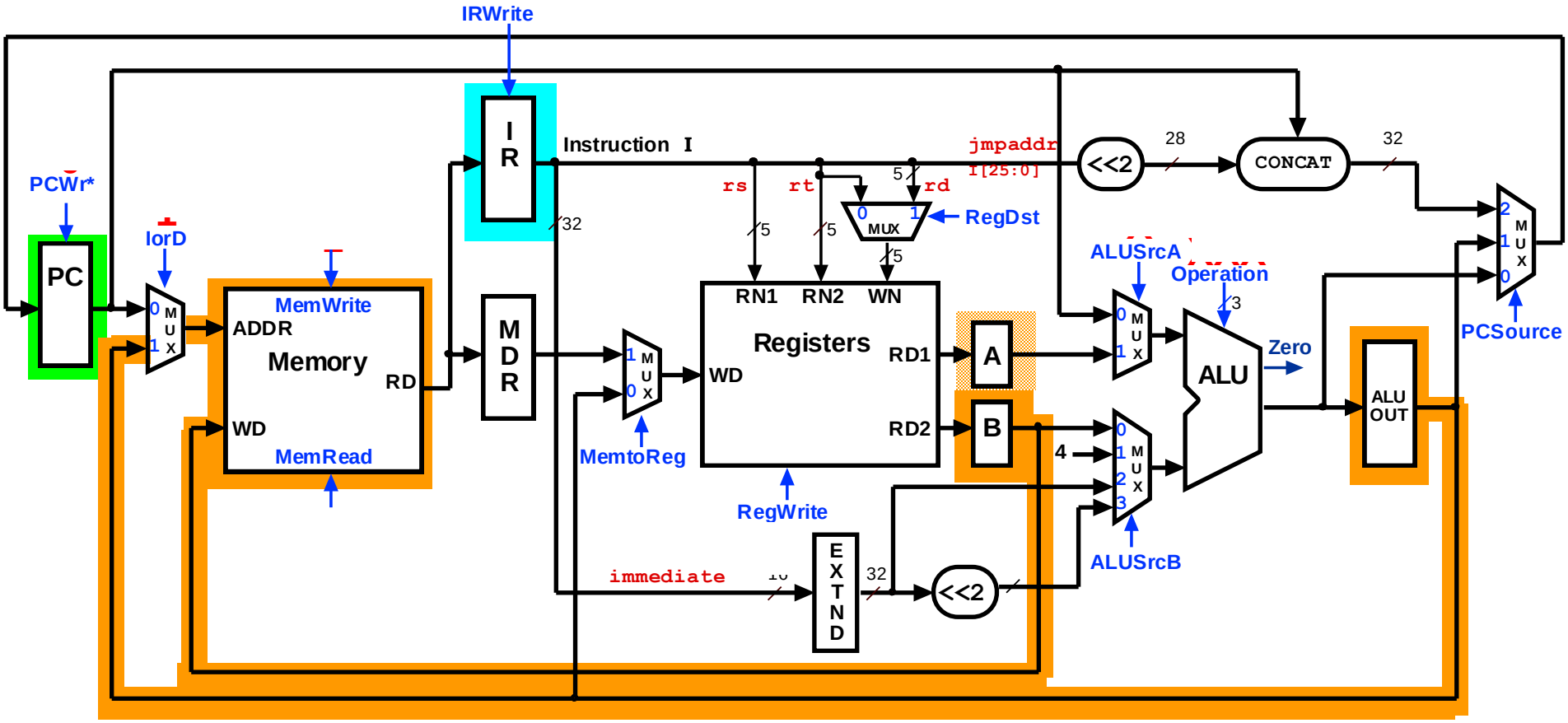
Multicycle Control Step (4): Memory Access - Read (`lw`)

```
MDR = Memory[ALUOut];
```



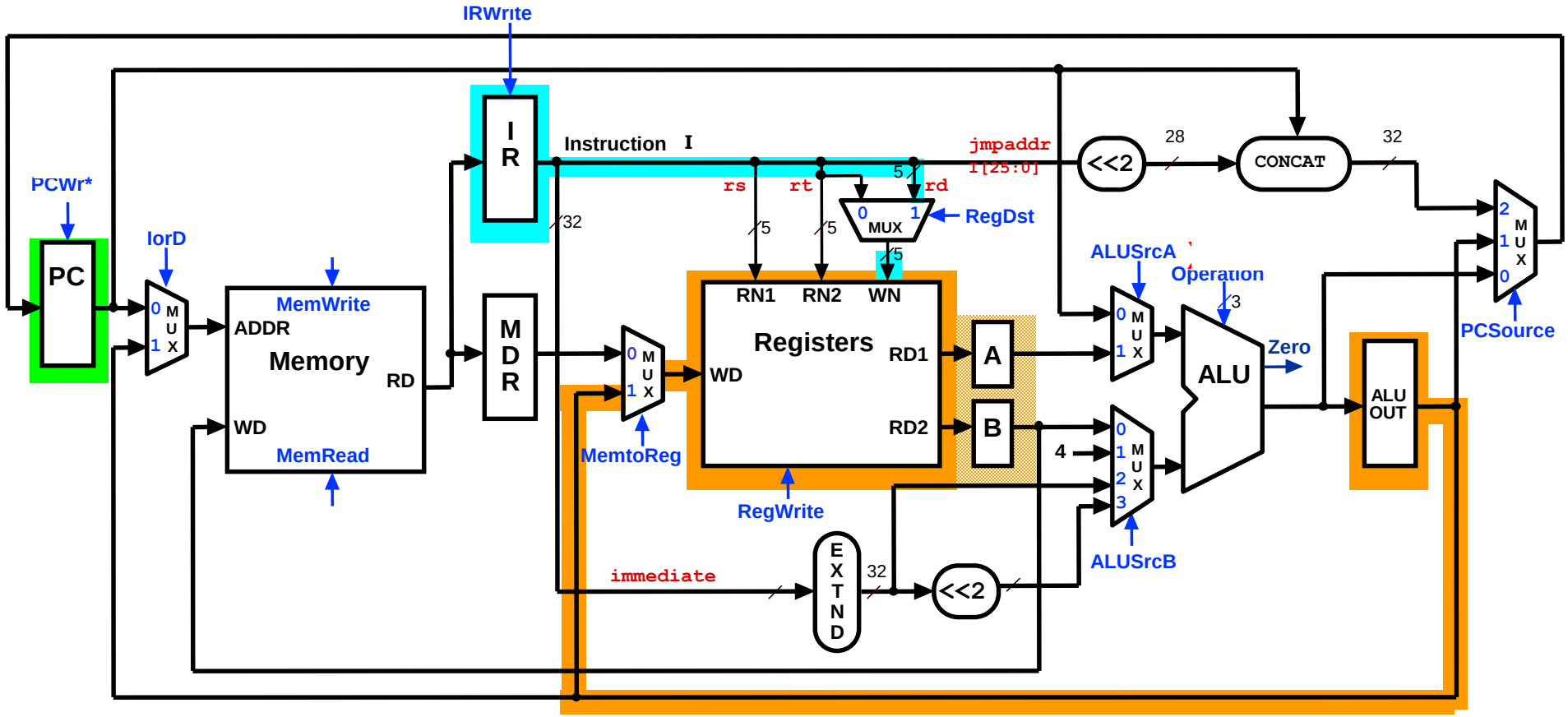
Multicycle Execution Steps (4) Memory Access - Write (sw)

```
Memory[ALUOut] = B;
```



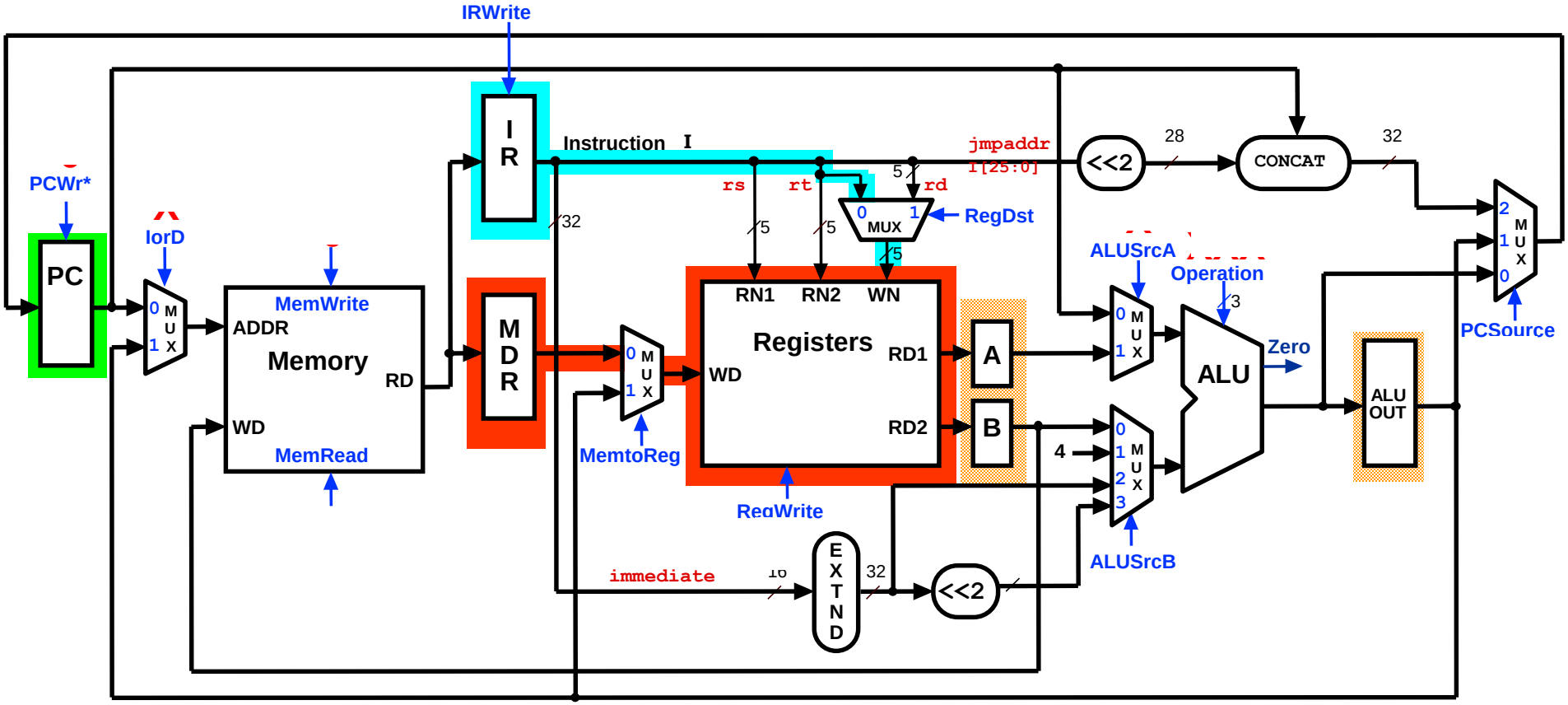
Multicycle Control Step (4): ALU Instruction (R-Type)

```
Reg[IR[15:11]] = ALUOut;          (Reg[Rd] = ALUOut)
```



Multicycle Execution Steps (5)
Memory Read Completion (lw)

```
Reg[IR[20-16]] = MDR;
```



Basit Sorular

- *Bu kodların yürütülmesi kaç cycle da tamamalanır?*

```
            lw $t2, 0($t3)
            lw $t3, 4($t3)
            beq $t2, $t3, Label #assume not equal
            add $t5, $t2, $t3
            sw $t5, 8($t3)
Label:      ...
```

- *Yürütülen 8. cycle'da ne oluşur?*

Clock time-line

- *In what cycle does the actual addition of $t2 and $t3 takes place?*

Örnek: multicycle CPU'da CPI

- Farzedelim
  - Önceki stayt'ın kontrol tasarımı
  - 22% yükleme , 11% depolama, 49% *R-tip işlemler*, 16% dallanma, and 2% *jump komutları olsun*
- Herbir adım 1 clock cycle gerektirirse CPI nedir.
- Çözüm:
  - Her bir komut sınıfı için önceki slayttan clock cycle'ların sayısı:
    - *yükleme* 5, depolama 4, *R-tip komutlar* 4, dallanma 3, jump 3
  - CPI = CPU clock cycles / komut sayısı count

    $= \Sigma$ (komut sayısı$_{\text{class i}} \times$ CPI$_{\text{class i}}$) / komut sayısı

    $= \Sigma$ (komut sayısı$_{\text{class I}}$ / komut sayısı) $\times$ CPI$_{\text{class I}}$

    $= 0.22 \times 5 + 0.11 \times 4 + 0.49 \times 4 + 0.16 \times 3 + 0.02 \times 3$

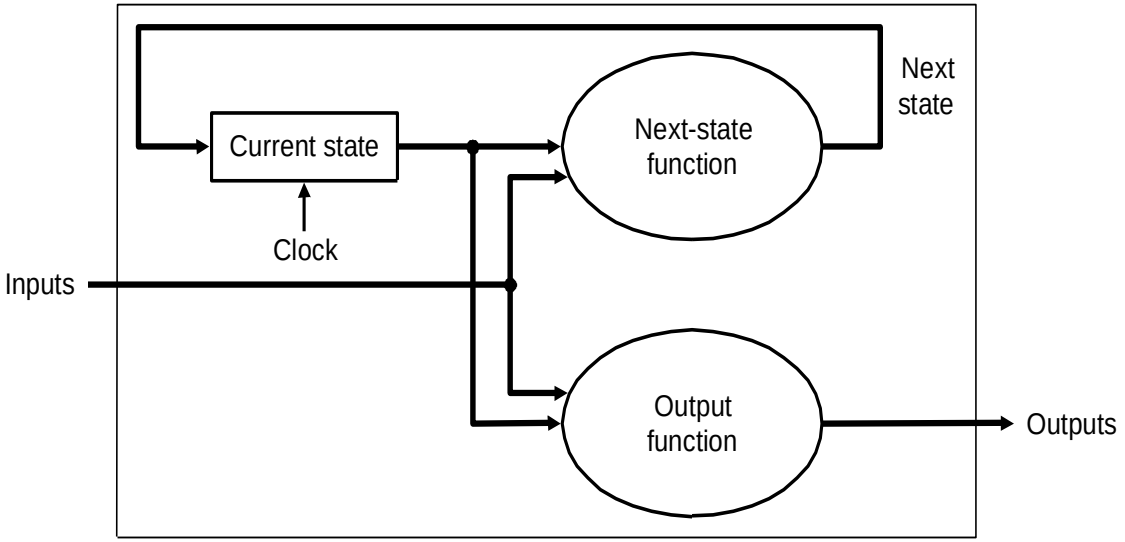
    $= 4.04$

Kontrol Gerçekleştirme

- Kontrol sinyallerinin değeri aşağıdakilere bağlıdır.
  - Hangi komut yürütülüyor
  - Hangi adım icra ediliyor
- Bir sonlu durum makinası kullanmak için bilgilerinizi kullanın
  - finite state machine grafiksel olarak belirtin, veya
  - Microprogramming kullanın
- Gerçekleştirme belirtimlerden(tanımlardan) elde edilir.

Finite State Machines

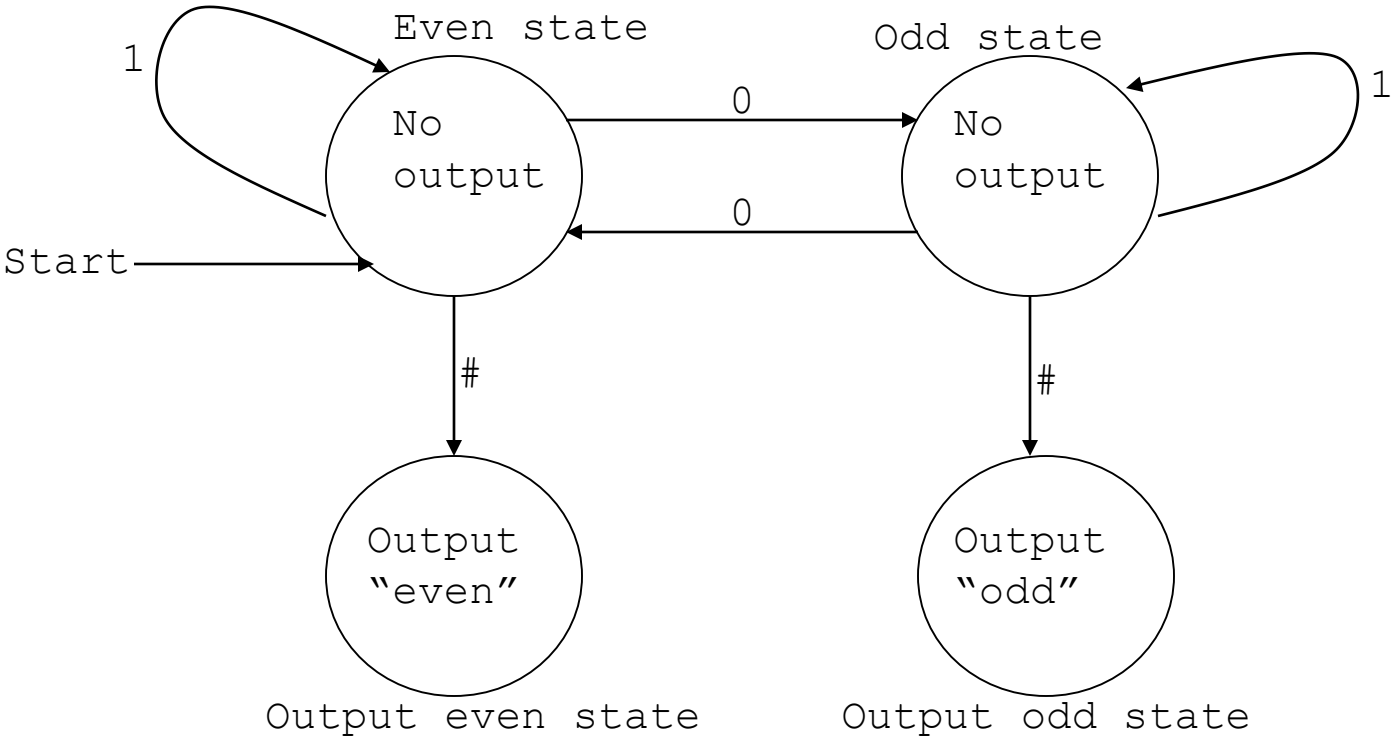
- Finite state machines (FSMs):
  - Durumlar kümesi ve
  - Sonraki durum fonksiyonu, giriş ve şimdiki durum tarafından hesaplama
  - Çıkış fonksiyonu, olası giriş ve şimdiki durum tarafından hesaplama



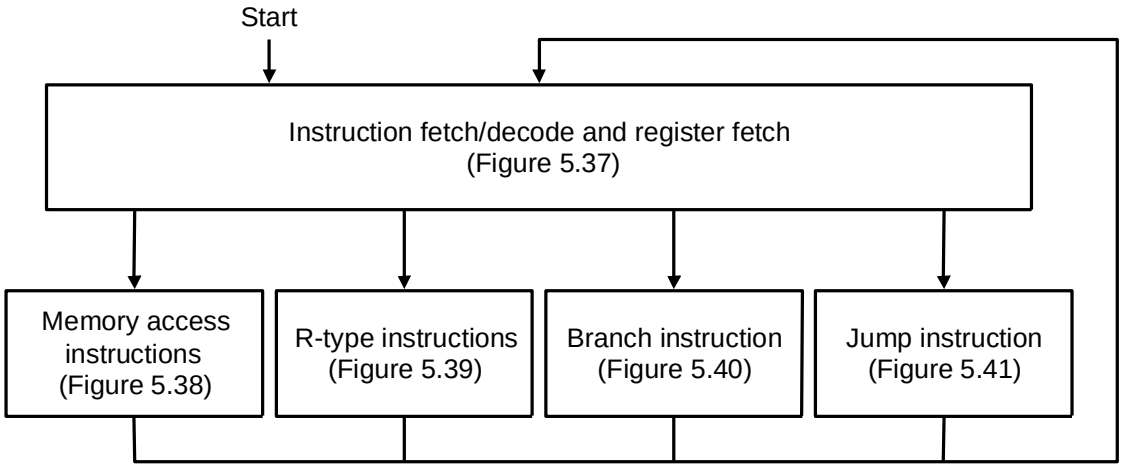
  - *Moore machine'sını kullanacağız.* Çıkış sadece şimdiki duruma bağlıdır.

Örnek: Moore Makinası

- Moore makinası aşağıdadır. # tarafından sonlandırılan giriş binary stringi ile çıkış; eğer stringdeki 0'ların sayısı çift ise "odd değilse "even olacaktır.



FSM Kontrol: yüksek Seviye bakış



FSM kontrolün yüksek seviye bakışı

Sinyaller durum dairesi içinde gösterilmektedir.

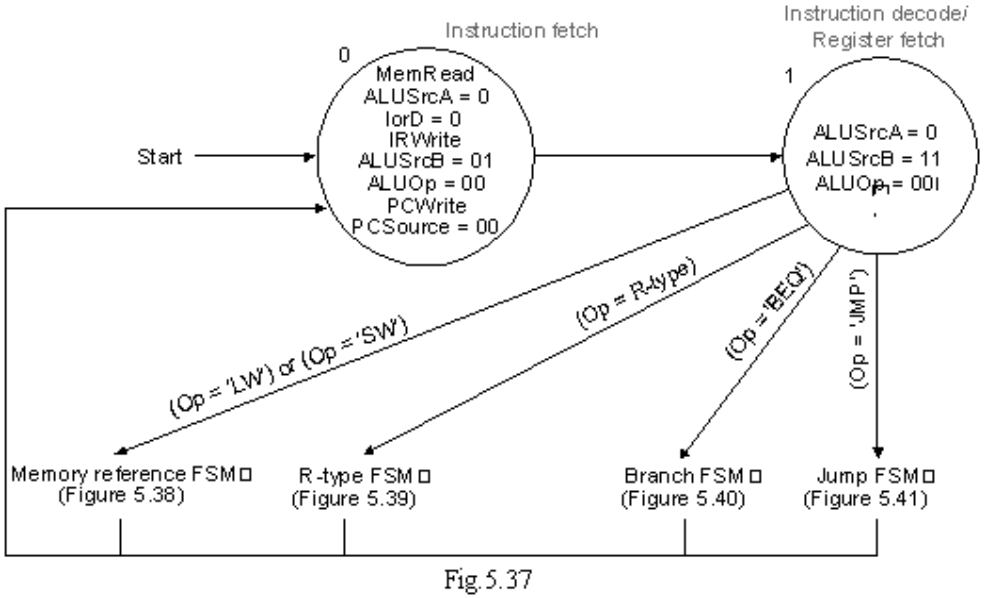
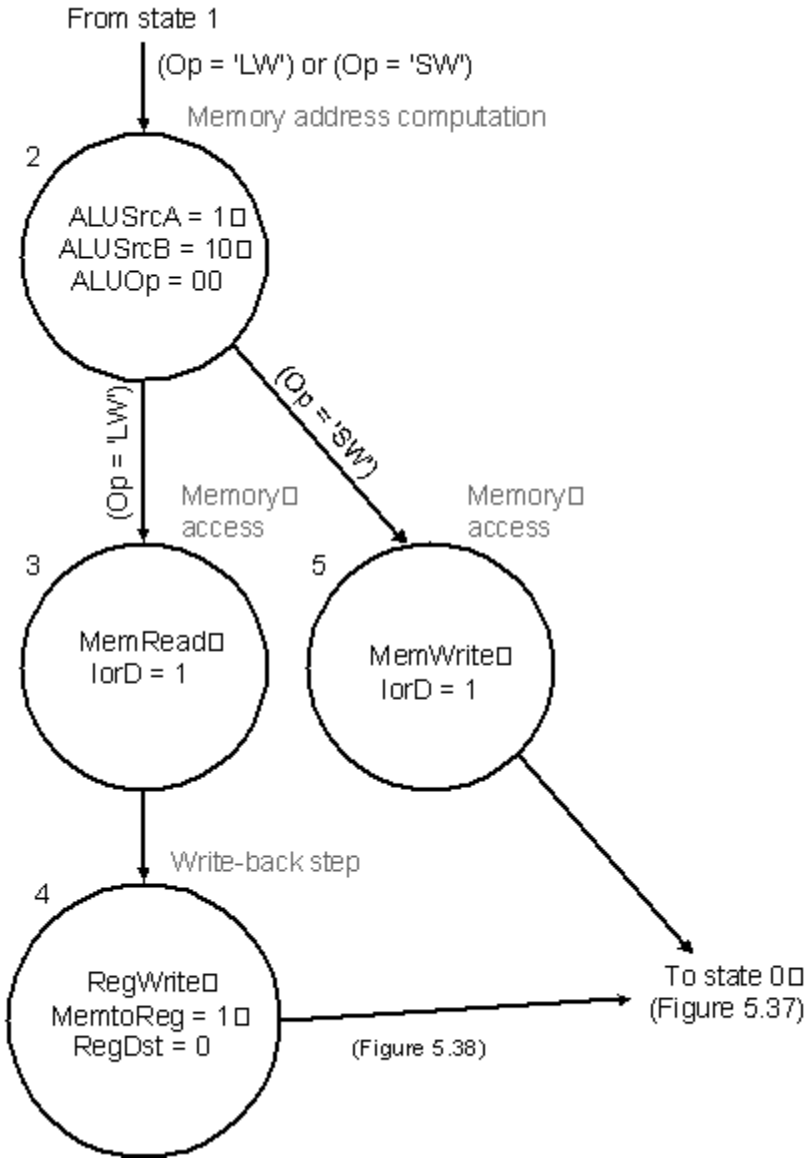


Fig. 5.37

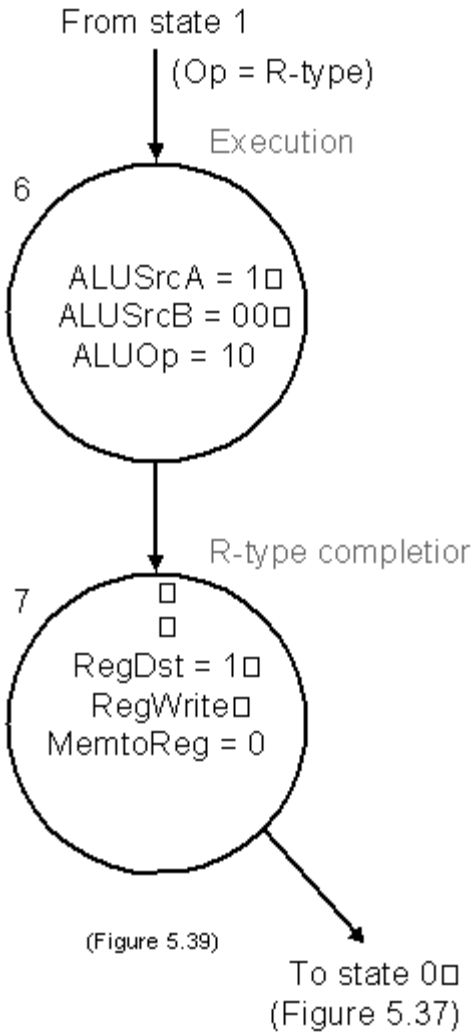
Her komutun komut yakalama ve decode işlemi benzerdir.

FSM Kontrol: Hafıza referens



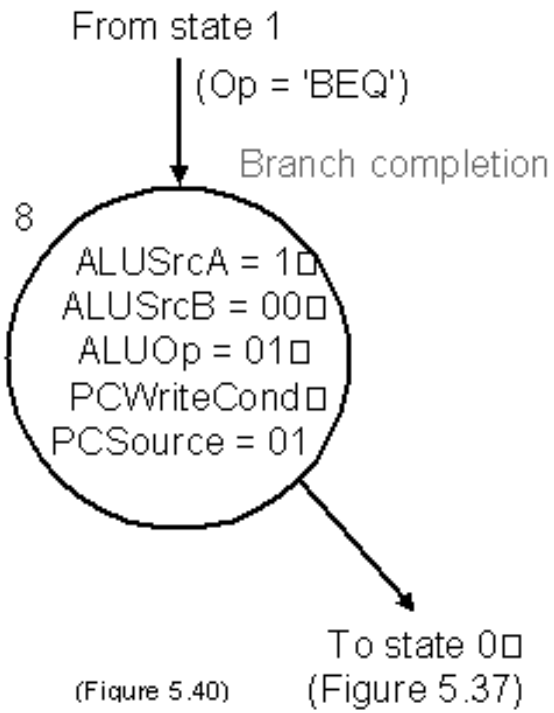
Hafıza-referansı için FSM kontrol 4 duruma sahiptir.

FSM Control: R-Tip Komut



R tip komut gerçekleştirmek için FSM kontrol 2 duruma sahiptir.

FSM Control: Branch Instruction



Dallanma komutları için FSM 1 duruma sahiptir.

FSM Kontrol: Jump Komutu

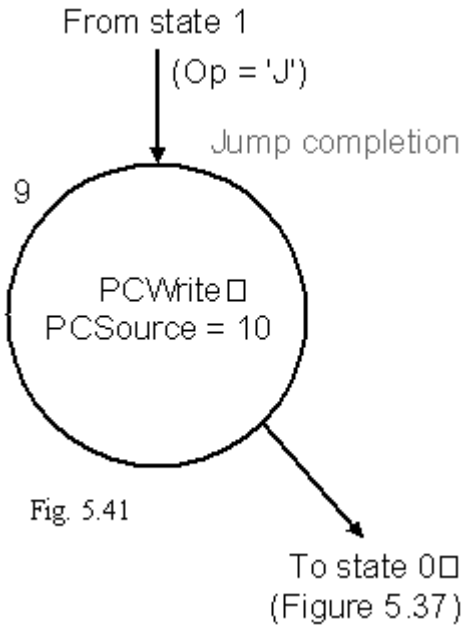
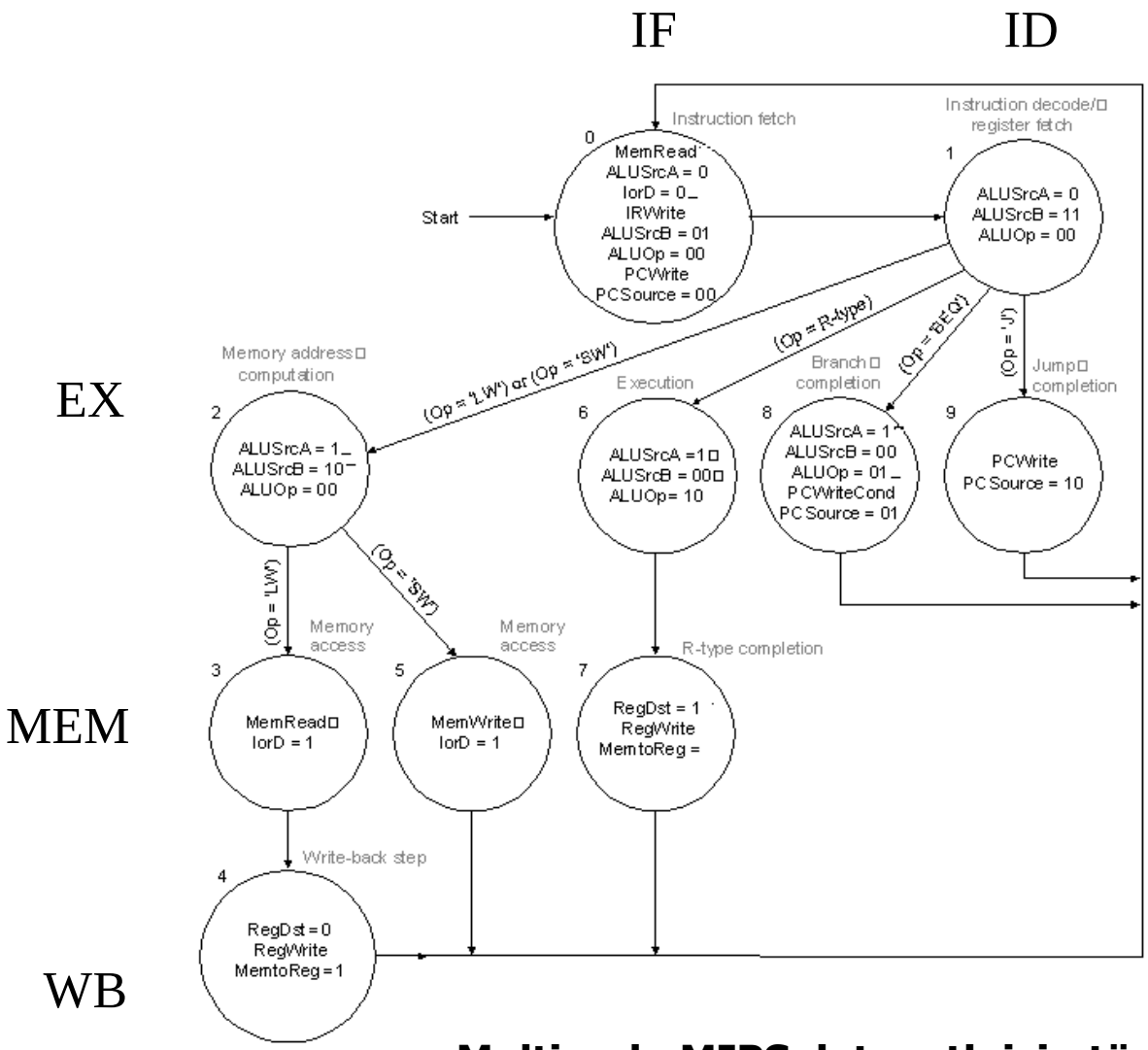


Fig. 5.41

Jump komutu için FSM kontrol 1 duruma sahiptir.

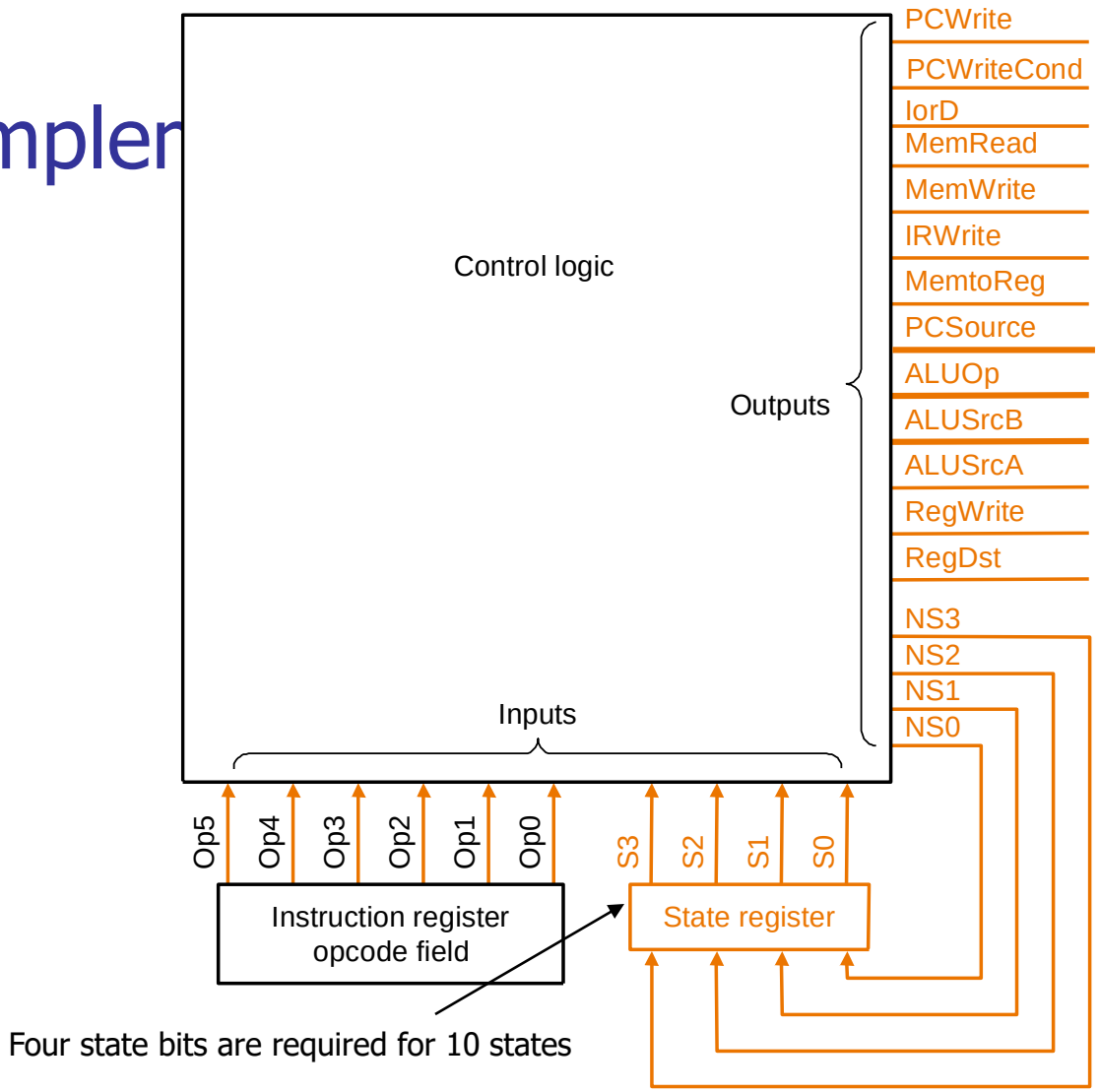
FSM Kontrola genel bakış



Yay üzerindeki etiketler hesaplanan sonraki durum şartlarıdır.

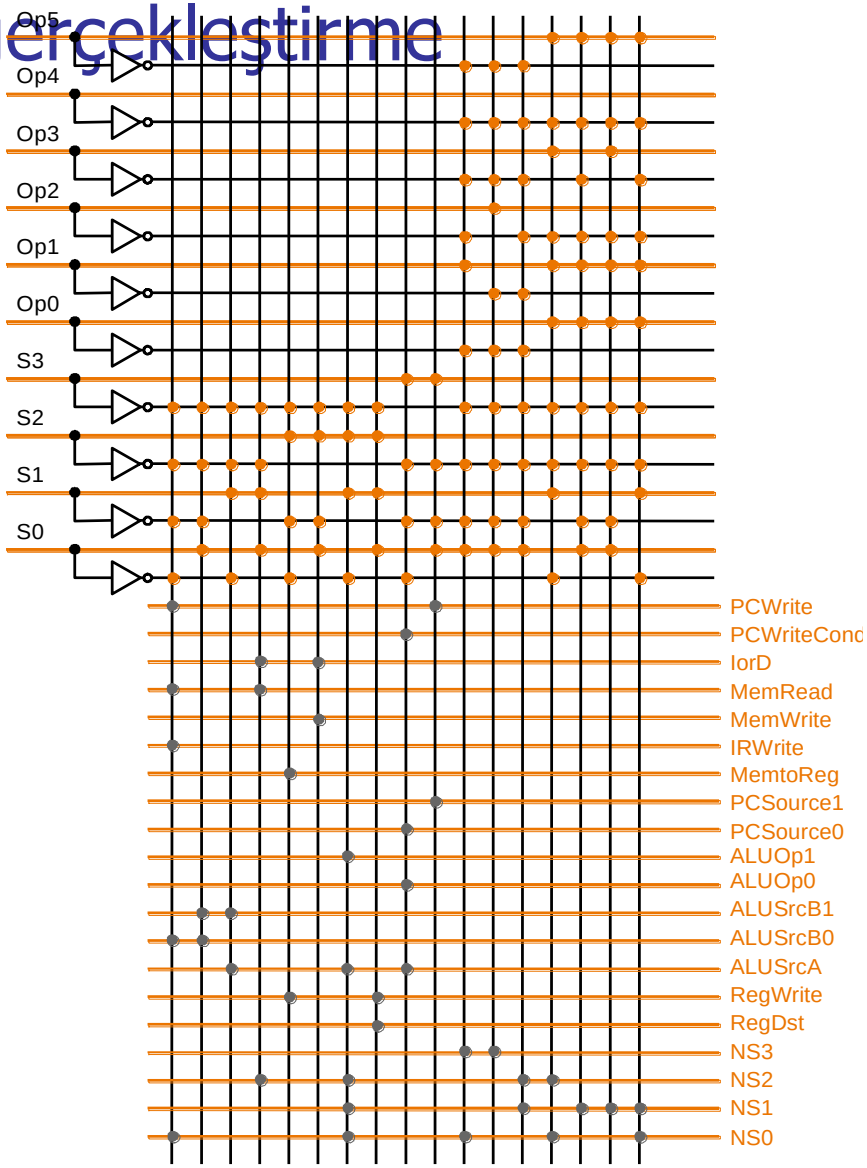
Multicycle MIPS datapath için tüm FSM kontrolü:
Multicycle datapath Kontrol II'ye bakınız

FSM Control:Implen



FSM gerçekleştirmeye yüksek seviye bakış: Kombinasyonal lojik bloklara girişler Şimdiki durum sayısı ve komut opcode bitleridir. Çıkışlar sonraki durum sayısı ve şimdiki durum için iddia edilen kontrol sinyalleridir.

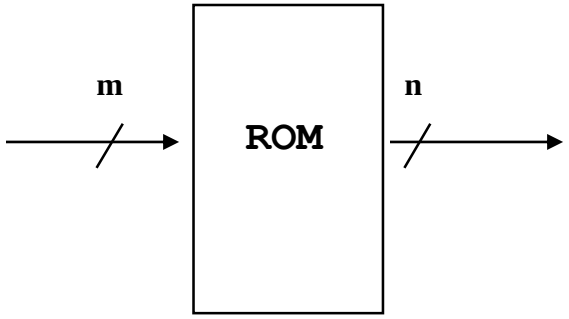
FSM Kontrol:PLA gerçekleştirme



Üst yarı AND düzlemidir ve bütün ürünleri hesaplar. Ürünler düşey hatlar tarafından düşük OR düzlemiyle taşınır. Herbir çıkış için toplam terim ilgili yatay düzlem tarafından verilir. Örnek IorD = S0.S1.S2.S3 + S0.S1.S2.S3

FSM Kontrol:ROM Gerçekleştirme

- ROM (Read Only Memory)
  - Hafıza yerlerinin değerleri zaman öncesinde sabittir.
- Bir rom doğruluk tablosu gerçekleştirmek için kullanılabilir.
  - Eğer adres m bit ise biz 2^m alanını adresleyebiliriz.
  - Çıkışlar giriş adresindeki bitlerdir.



m = 3
n = 4

address			output			
0	0	0	0	0	1	1
0	0	1	1	1	0	0
0	1	0	1	1	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	1
1	1	0	0	1	1	0
1	1	1	0	1	1	1

m-giriş n-çıkış ise ROM 2^m x n bittir. – böyle bir ROM n bitlik 2^m dizisi olarak düşünülebilir.

FSM Kontrol: ROM vs. PLA

- ROM geliştirmek:
  - 4 durum bitleri 16 çıkış sinyali verir.ROM'un 2^4 x 16 bitleri
  - Bütün 10 bit 4 sonraki durum bitlerini verir.ROM'un 2^{10} x 4 bitleri
  - Toplam – ROM 4.3K bit olur
- PLA daha küçüktür.
  - Ürün terimleri paylaşılabilir.
  - Aktif bir çıkış üretmek için yanlızca girişlere ihtiyaç vardır
- PLA boyut = (#girişler $\times$ #ürünler terimi) + (#çıkışlar $\times$ #ürünler terimi)
  - FSM Kontrol PLA = (10x17)+(20x17) = 460 PLA hücresi
- PLA hücreleri bir ROM hücre boyutundadır. (biraz daha büyük)

Microprogramming

- Microprogramming iözel FSM kontrol methodudur. (Bir programlama dilini andıran yada grafikten ziyade text)
  - Eğer komut seti büyük veya her komutun cycle sayısı büyük ise FSM çok büyük olduğundan bu uygundur.
  - Böyle grafiksel gösterim durumlarında zor olabilir. Binlerce durum olduğu ve onlarla bağlantılı yaylar olduğunda
  - Bir microprogram özelliği: Gerçekleştirme ROM veya PLA ile gerçekleştirmedir.
- Bir *microprogram microkomutlar dizisidir.*
  - Herbir microinstruction8 alana sahiptir. (Etiket + 7 fonksiyon)
    - Label: Microcode dizisini kontrol etmede kullanılır.
    - ALU Kontrol:ALU tarafından yapılacak işlemleri belirtir.
    - SRC1: İlk ALU operandı için kaynağı belirtir.
    - SRC2: ikinci ALU operandı için kaynağı belirtir.
    - Register Kontrol: Register file için read/write belirtir.
    - Hafıza: Hafıza için read/write belirtir.
    - PCWrite Kontrol :PC'nin yazmasını belirtir.
    - Sequencing: sonraki komutu seçmeyi belirtir.

Microprogramming

- The *Sequencing alan değeri* microprogramın yürütme sırasını belirler.
  - value *Seq* : *Sonraki microkomuta geçme kontrolü*
  - value *Fetch* : sonraki başlanacak ilk microkomuta dallanma ör: mikroprogramdaki ilk microkomut gibi
  - value *Dispatch i* : gönderme tablo girişi (*dispatching olarak adlandırılır*) ve kontrol girişi tabanlı microkomuta dallanma :
    - Dispatching bir tablo oluşturma aracı tarafından gerçekleştirilir, *dispatch tablo olarak adlandırılır*, onun girişleri microkomut etiketleridir ve o kontrol girişi tarafından indexlenir. Çoklu dispatch tabloları olabilir. –sıralama alanındaki *Dispatch i* değeri göstermeye *i ninci* dispatch tablo kullanılır.

Kontrol Microprogram

- The microprogram corresponding to the FSM control shown graphically earlier:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

Microprogram 10 mikrokomut içerir.

Dispatch ROM 1		
Op	**Opcode name**	**Value**
000000	R-format	Rformat1
000010	jmp	JUMP1
000100	beq	BEQ1
100011	lw	Mem1
101011	sw	Mem1

Gönderme Tablo 1

Dispatch ROM 2		
Op	**Opcode name**	**Value**
100011	lw	LW2
101011	sw	SW2

Gönderne Table 2

Microcode: Trade-offs

- Özellikleri Avantajları
  - Tasarlamak ve yazmak kolaydır.
  - Tipik olarak imalatcı mimari ve microkodu paralel tasarlar
- Gerçekleştirme Avantajları
  - Hafızadaki değeri değiştirmek kolaydır (e.g., off-chip ROM)
  - Diğer mimariler taklit edilebilir.
  - Dahili registerlerden yararlanılabilir.
- Gerçekleştirme dezavantajları
  - Kontrol şimdilerde aynı chip üzerinde gerçekleştirilir. Off Chip ROM'un sahip olduğu avantaja sahip değildir.
  - ROM on-board cache'den hızlı değildir.
  - Microkodu değiştirmeye az ihtiyaç vardır. Genel amaçlı bilgisayarlar özel uygulamalar için tasarlanmış bilgisayarlardan çok daha fazla kullanılmaktadır.

Özet

- *Bu bölümde datapath tasarımı ve bütün modern bilgisayar mimarisi tanımlandı*
- Multicycle datapath'ler single-cycle'a göre 2 büyük avantaj sunar.
  - Fonksiyonel birimler bir tek komutla yeniden kullanılabilir.
  - Daha kısa execution path'ler ile komutlar birkaç cycle tüketilmesiyle daha hızlı tamamlanabilir.
- Modern bilgisayarlar, gerçekte, take the multicycle paradigm to a higher level to achieve greater instruction throughput:
  - *pipelining* (bir sonraki komut) birçok komutu(farklı cycle'a sahip olsalar bile) eşzamanlı yürütme
  - *MIPS mimarisi pipeline olarak tasarlandı.*