معماری کامپیوتر

۱۳۹۵-۱۱-۱۳

جلسه‌ی چهاردهم

پردازنده‌های چندسیکلی

دانشگاه شهید بهشتی

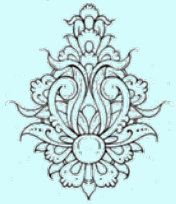
دانشکده‌ی مهندسی برق و کامپیوتر

بهار ۱۳۹۵

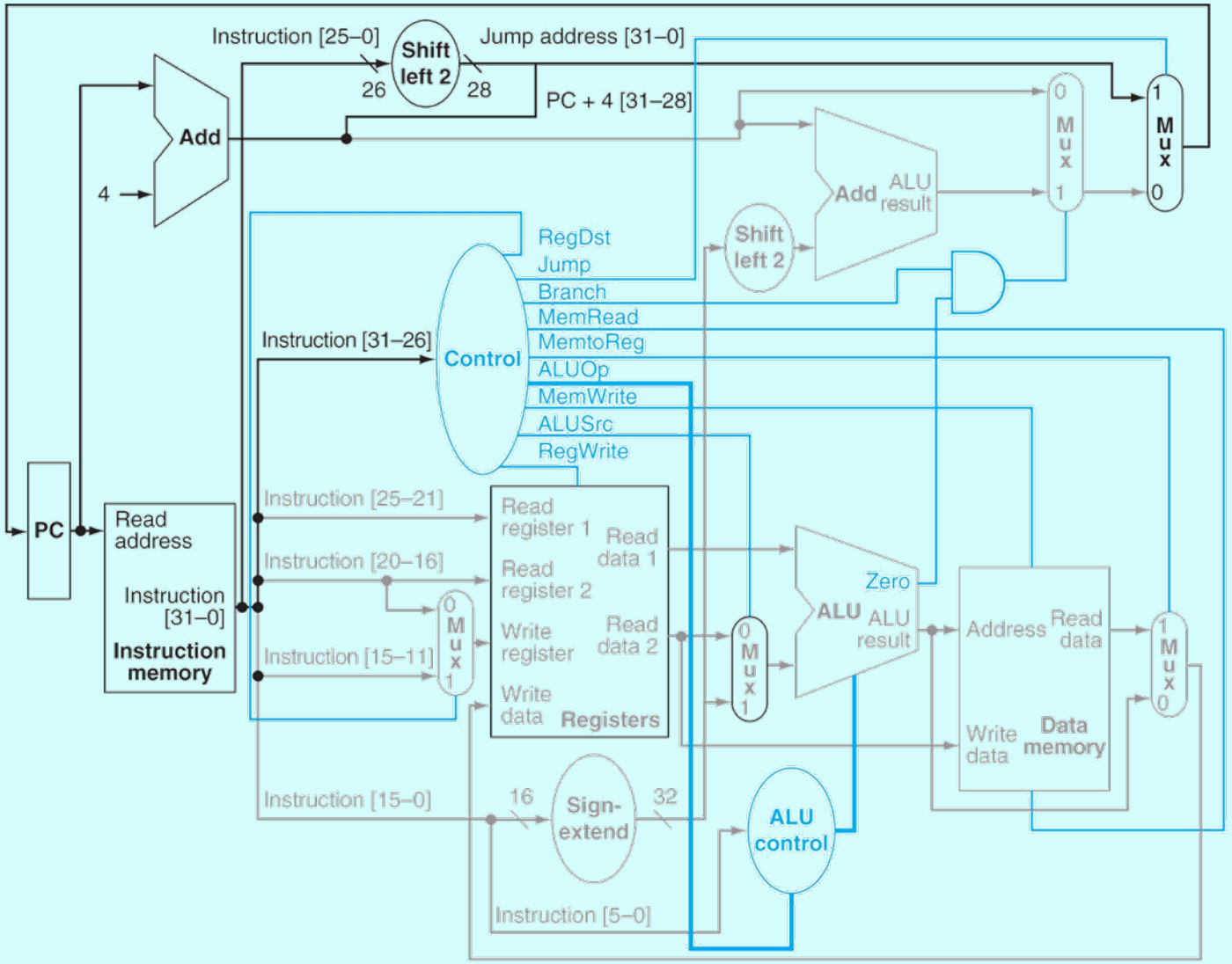
احمد محمودی ازناوه

فهرست مطالب

- گذرداده‌ی تک سیکلی در برابر گذر داده‌ی چندسیکلی



مسیرگذار داده + واحد کنترل

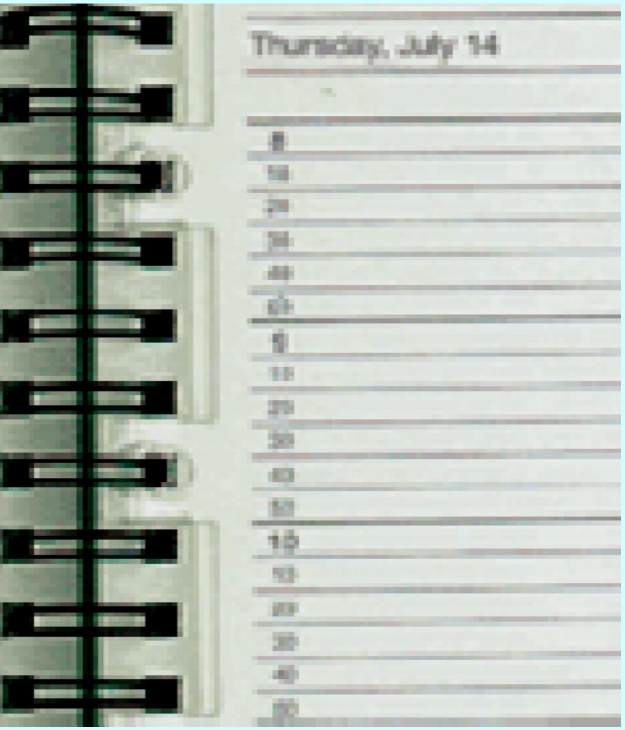


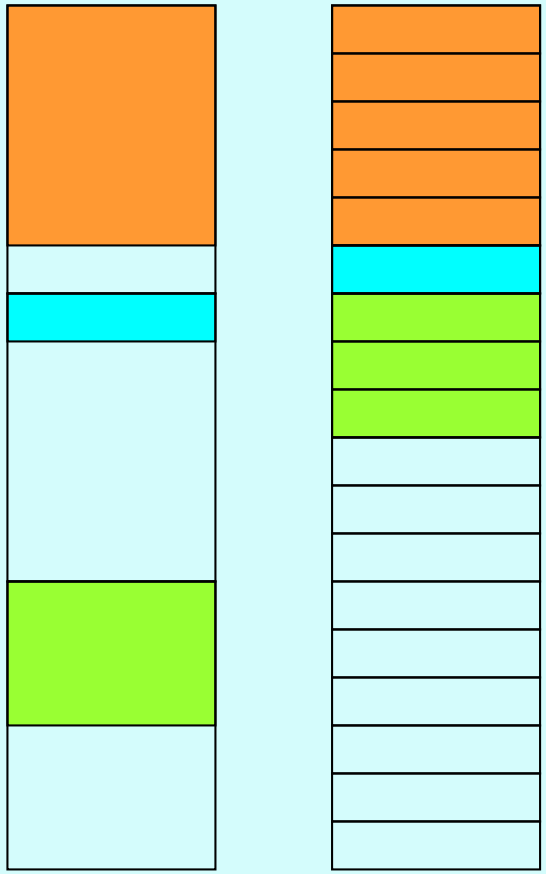
کارایی مسیر گذار داده‌ی تک سیکلی

- فرض می‌کنیم زمان مورد نیاز برای هر کدام از بخش‌های پردازنده به صورت زیر باشد:
  - برای نوشتن و خواندن ثبات 100ps
  - دستیابی به حافظه و محاسباتی و ... 200ps
- در معماری یک سیکلی، باید طول پالس را برابر با طول کندترین دستوالعمل در نظر گرفت.

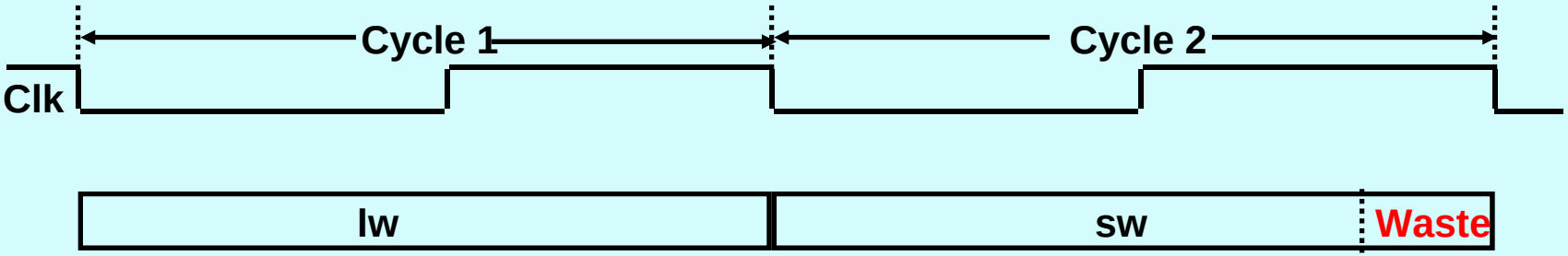
Instr.	Fetch	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	200	100	200		100	600
load	200	100	200	200	100	800
store	200	100	200	200		700
beq	200	100	200			500
jump	200					200

اجرای هر دستورالعمل در چند سیکل(در برابر اجرا در یک سیکل)





معایب و مزایای سیستم‌های تک‌سیکلی



- پالس ساعت باید به اندازه‌ی کندترین دستور در نظر گرفته شود، این مساله موجب کندی کلی پردازنده خواهد شد.
- در عمل منجر به افزایش مساحت هم خواهد شد، چرا که برخی واحدهای عملیاتی باید تکرار شوند. (در طول یک سیکل به صورت مشترک قابل استفاده نیستند).

- طراحی سیستم‌های تک‌سیکلی بسیار ساده است.

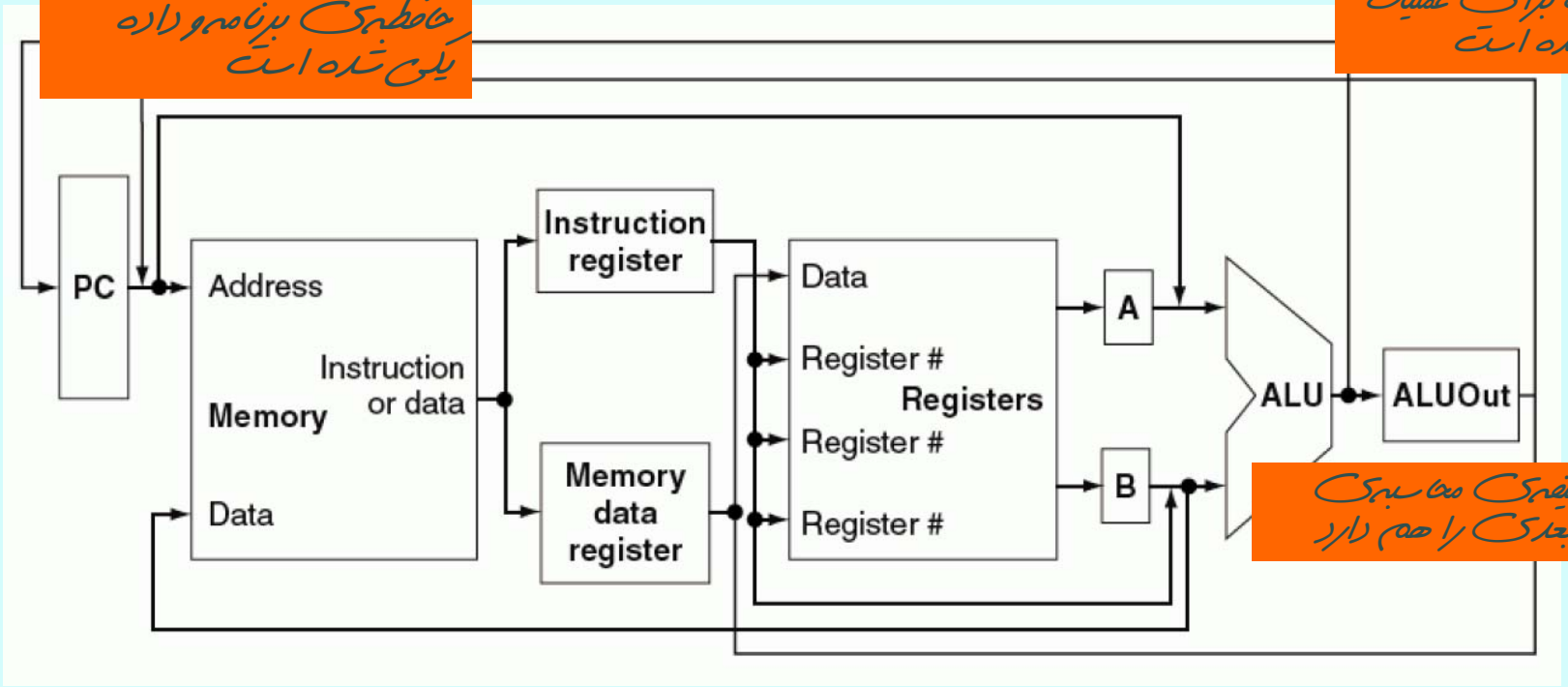
در ادامه‌ی بحث از ویرایش قدیمی کتاب استفاده خواهیم کرد.

پردازنده‌ی چندسیکلی

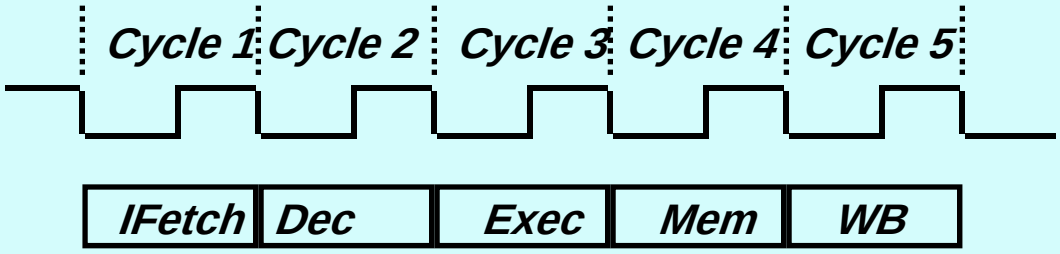
سه تفاوت عمده با حالت تک سیکلی دارد:

حافظه‌ی برنامه و داده یکی شده است

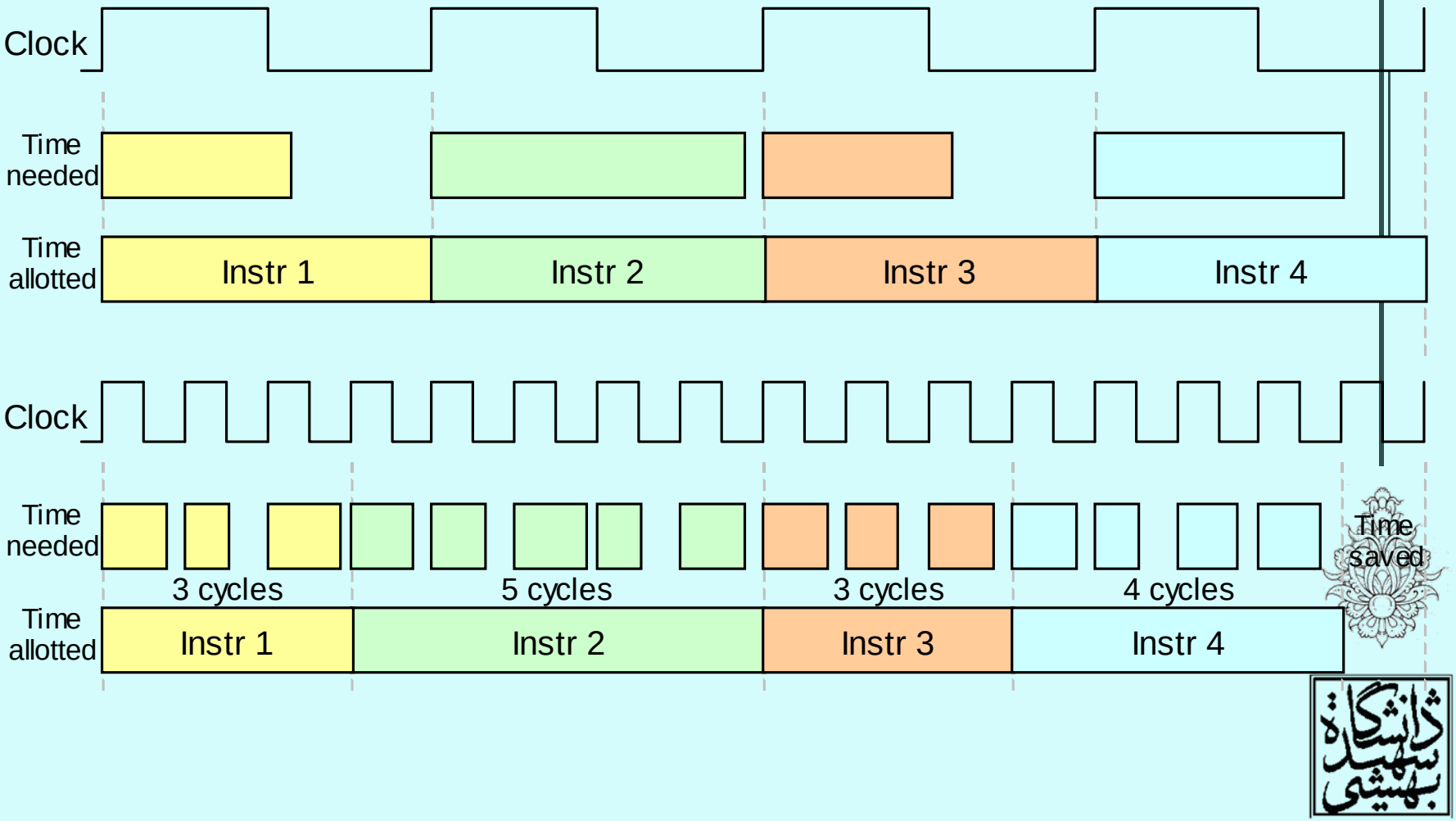
تعدادی ثبات برای عملیات میانی اضافه شده است



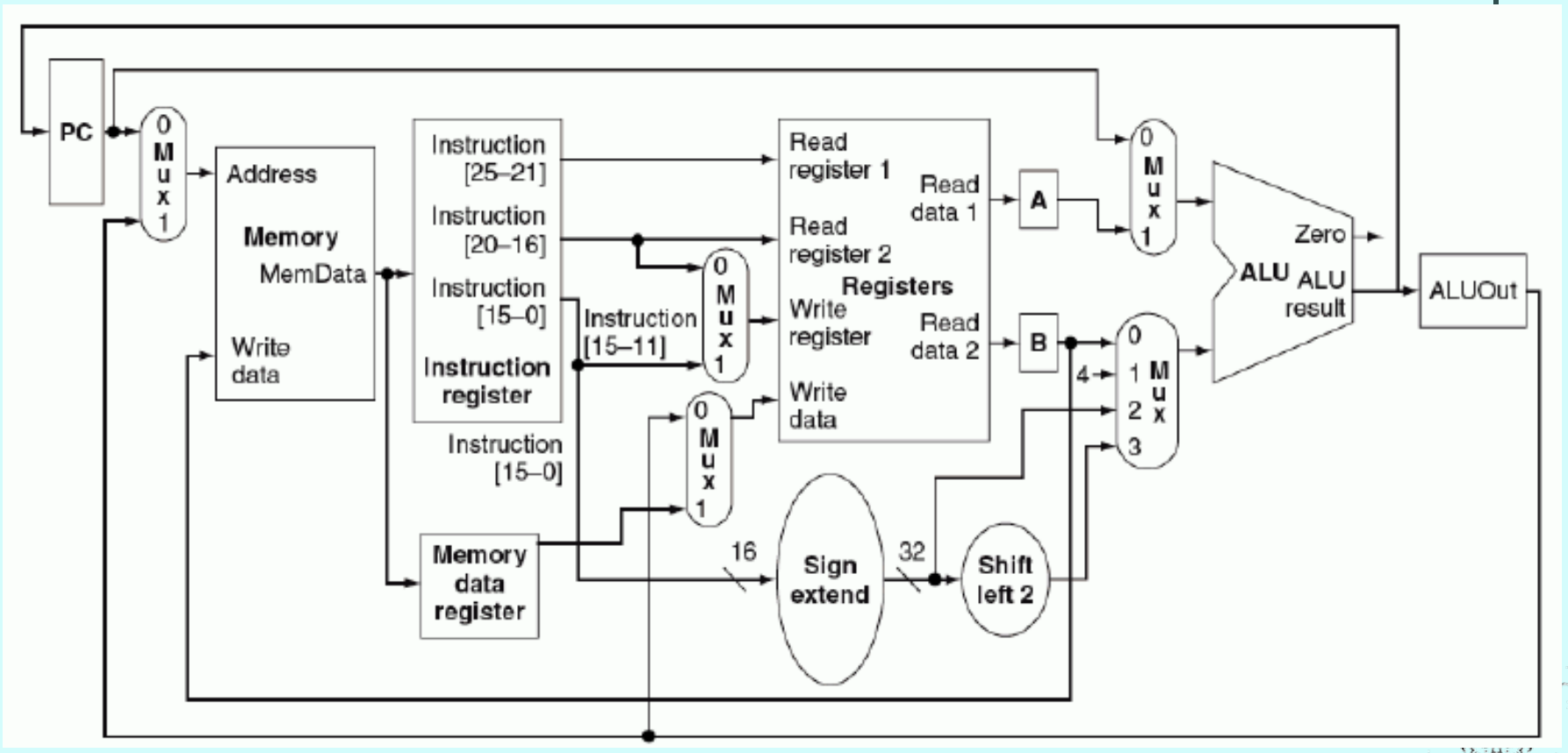
واحد ALU وظیفه‌ی محاسبه‌ی آدرس دستور بعدی را هم دارد



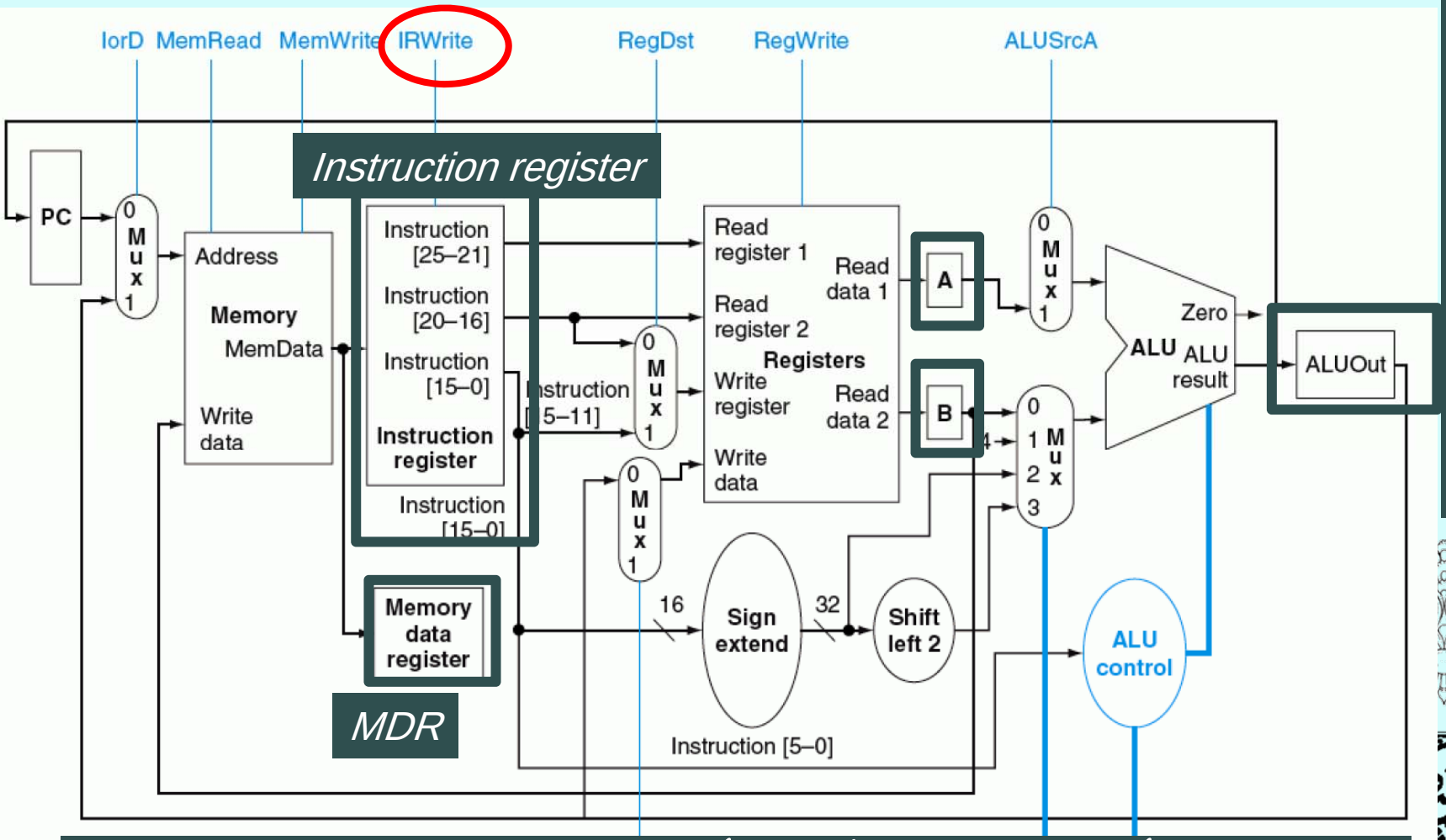
گذرداده‌ی تک سیکلی در برابر چند سیکلی



مسیر گذرداده‌ی چندسیکلی



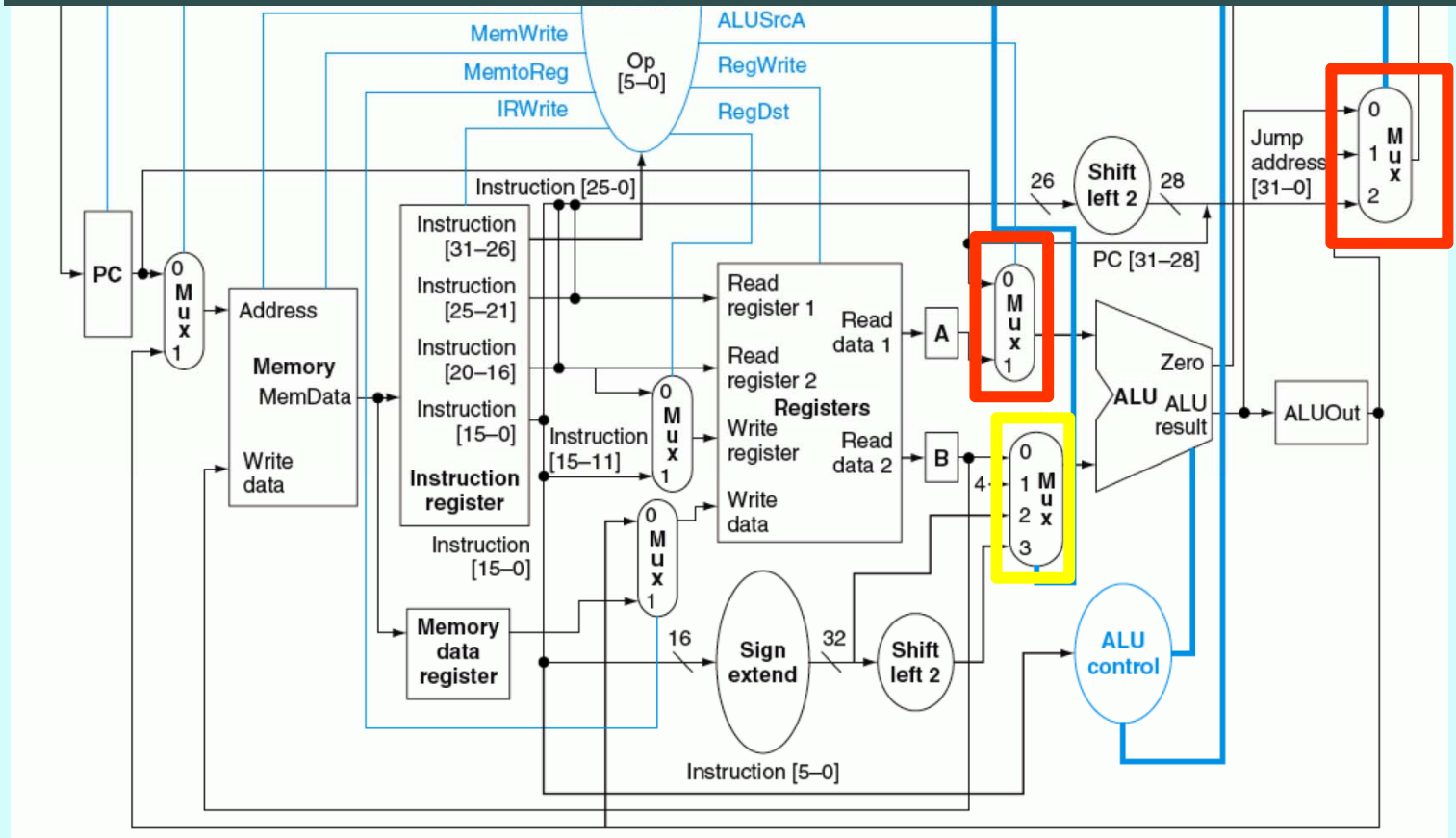
از میان state-element های اضافه شده تنها IR تا پایان مورد استفاده قرار می‌گیرد و مابقی تنها بین دو کلاک متوالی استفاده می‌شوند



در پایان هر سیکل، داده‌هایی که در سیکل‌های بعد مورد استفاده قرار می‌گیرند، باید ذخیره شوند

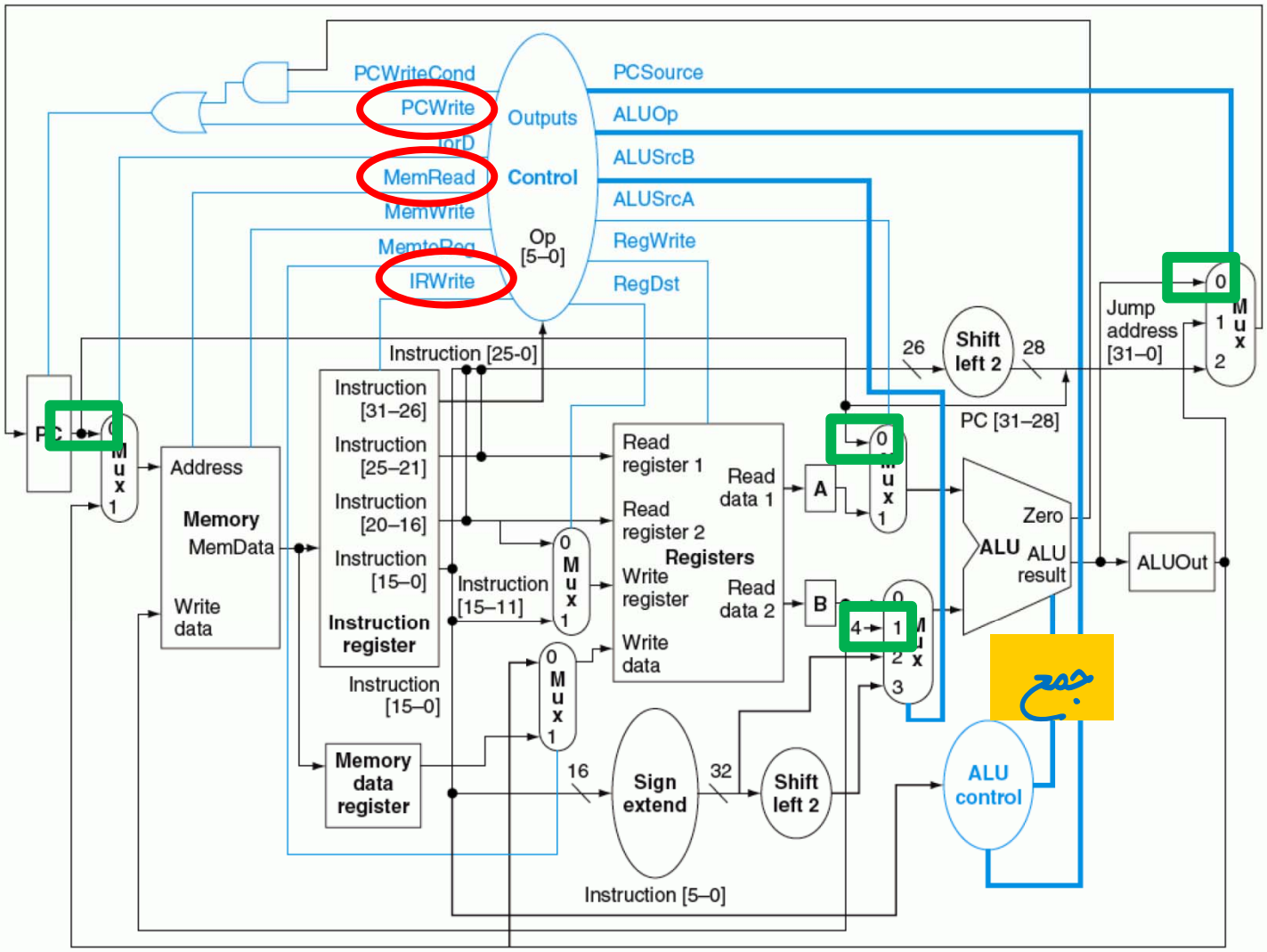
مسیر گذرداده‌ی چندسیکلی

با توجه به این که برخی واحدهای عملیاتی چندین بار مورد استفاده قرار می‌گیرند باید یک سری ماتی‌پلکسر اضافه و یا ماتی‌پلکسرهای بزرگ‌تر استفاده کرد.



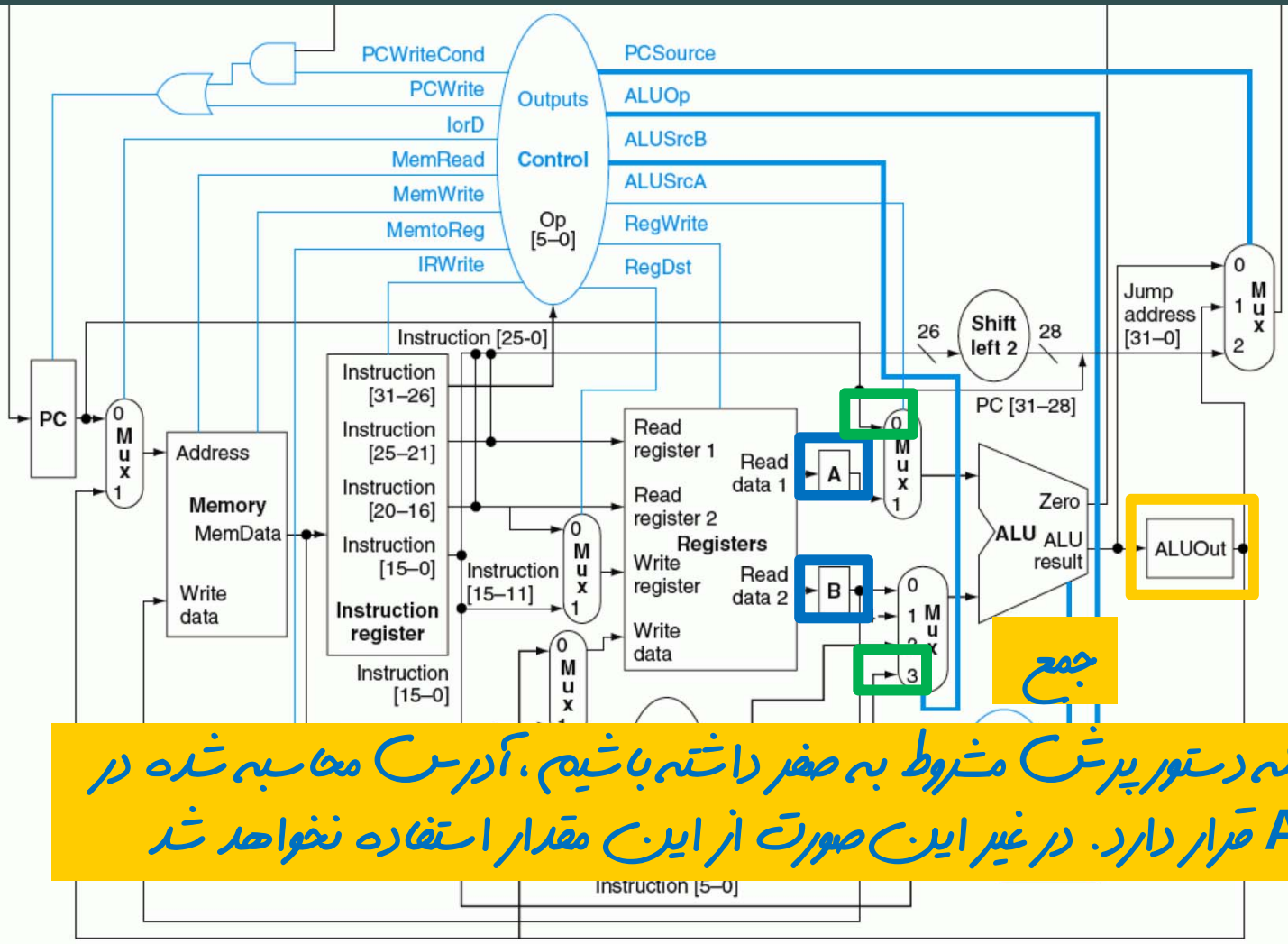
```
IR <= Memory[PC];
PC <= PC + 4;
```

گام نخست–واکشی



گام دوم– رمزگشایی و خواندن ثبات‌ها

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend(IR[15:0]) << 2);
```



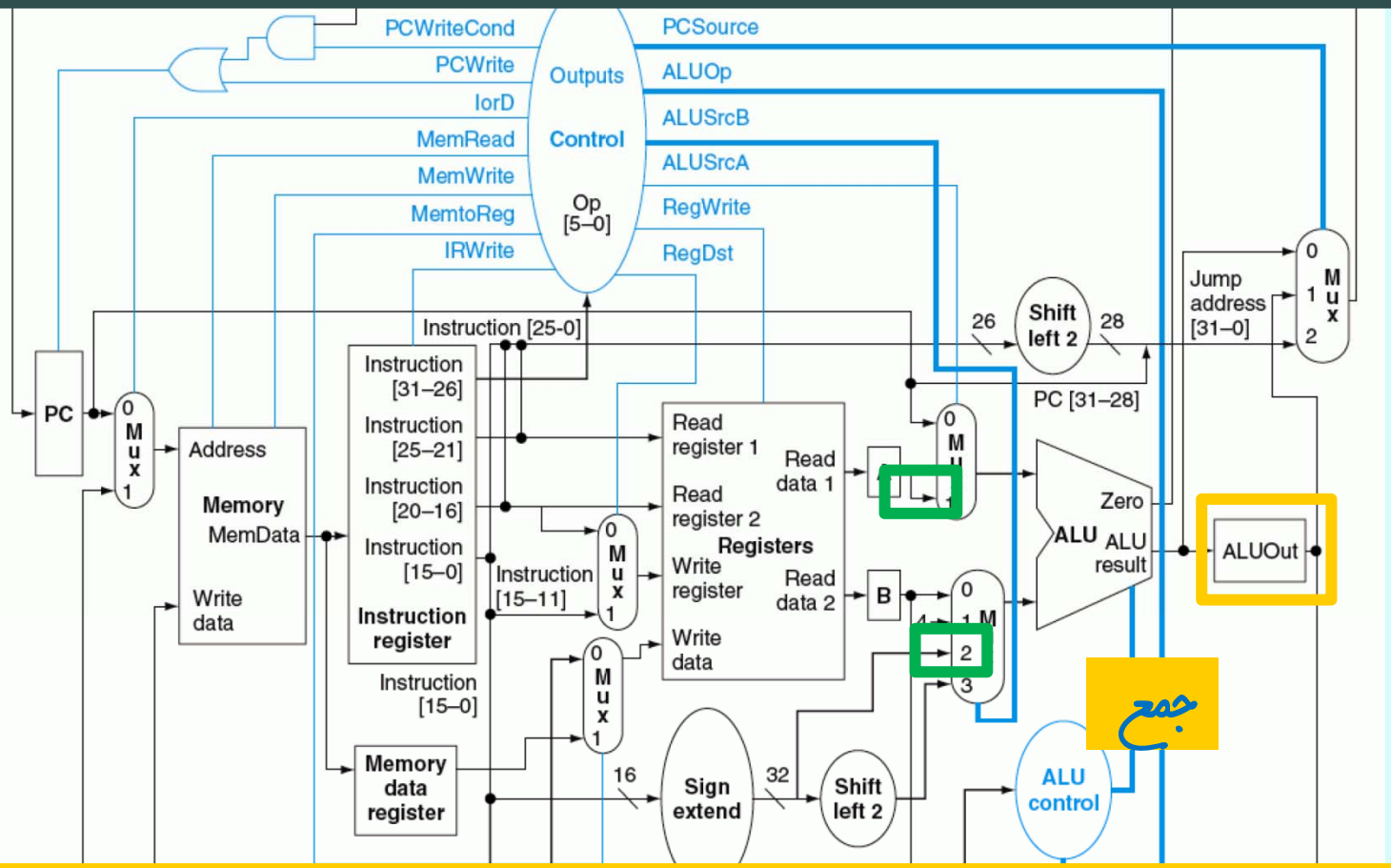
جمع

در صورتی که دستور پرش مشروط به صفر داشته باشیم، آدرس محاسبه شده در ALUOut قرار دارد. در غیر این صورت از این مقدار استفاده نخواهد شد

Memory Reference

گام سوم– اجرا

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

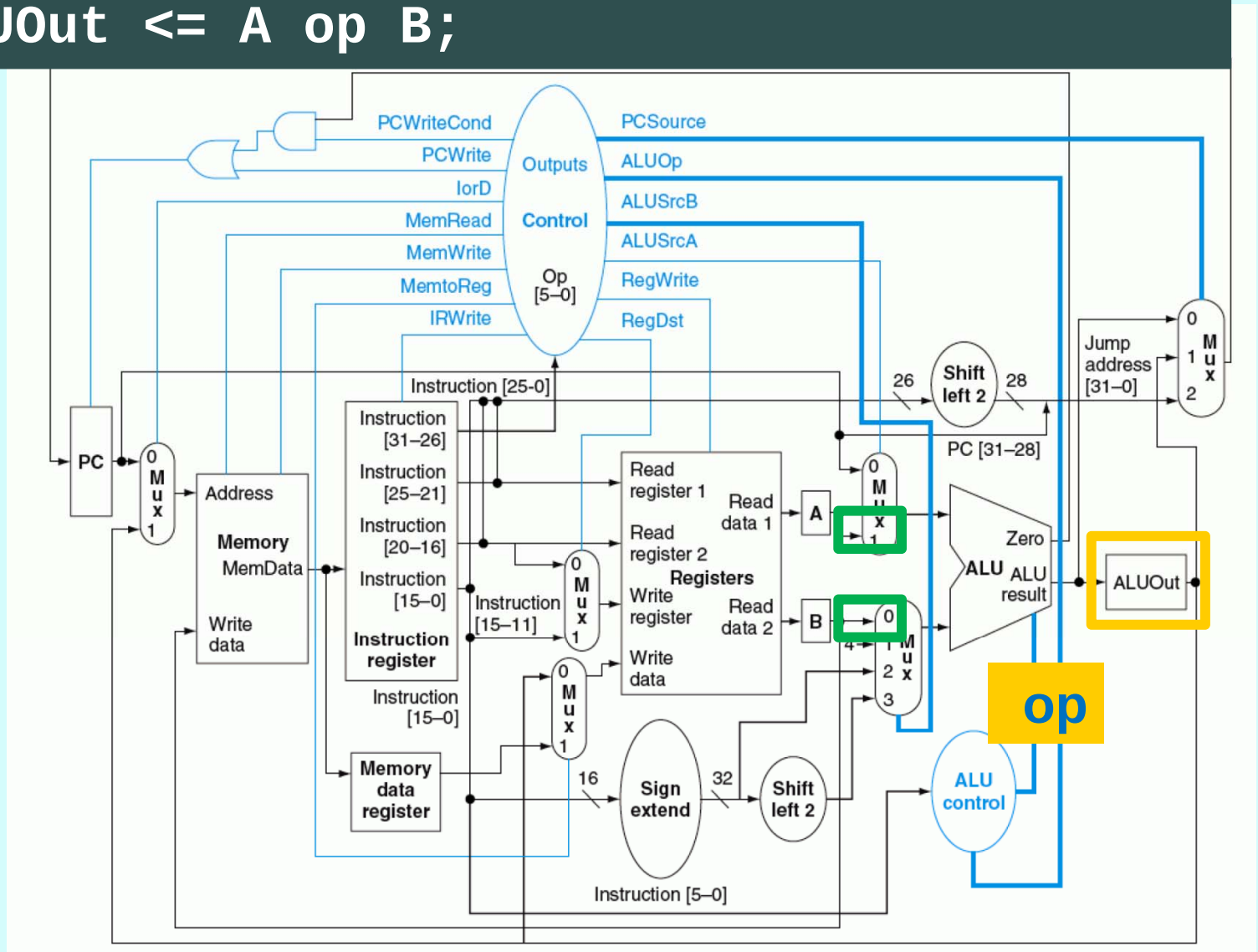


تا این جا مشخص نبود، چه دستوری باید اجرا شود، در واقع تا این مرحله برای همه دستورها سیگنال‌های کنترلی مشترک هستند، از این به بعد با توجه به نوع دستور تفاوت خواهد بود

R-type

گام سوم– اجرا

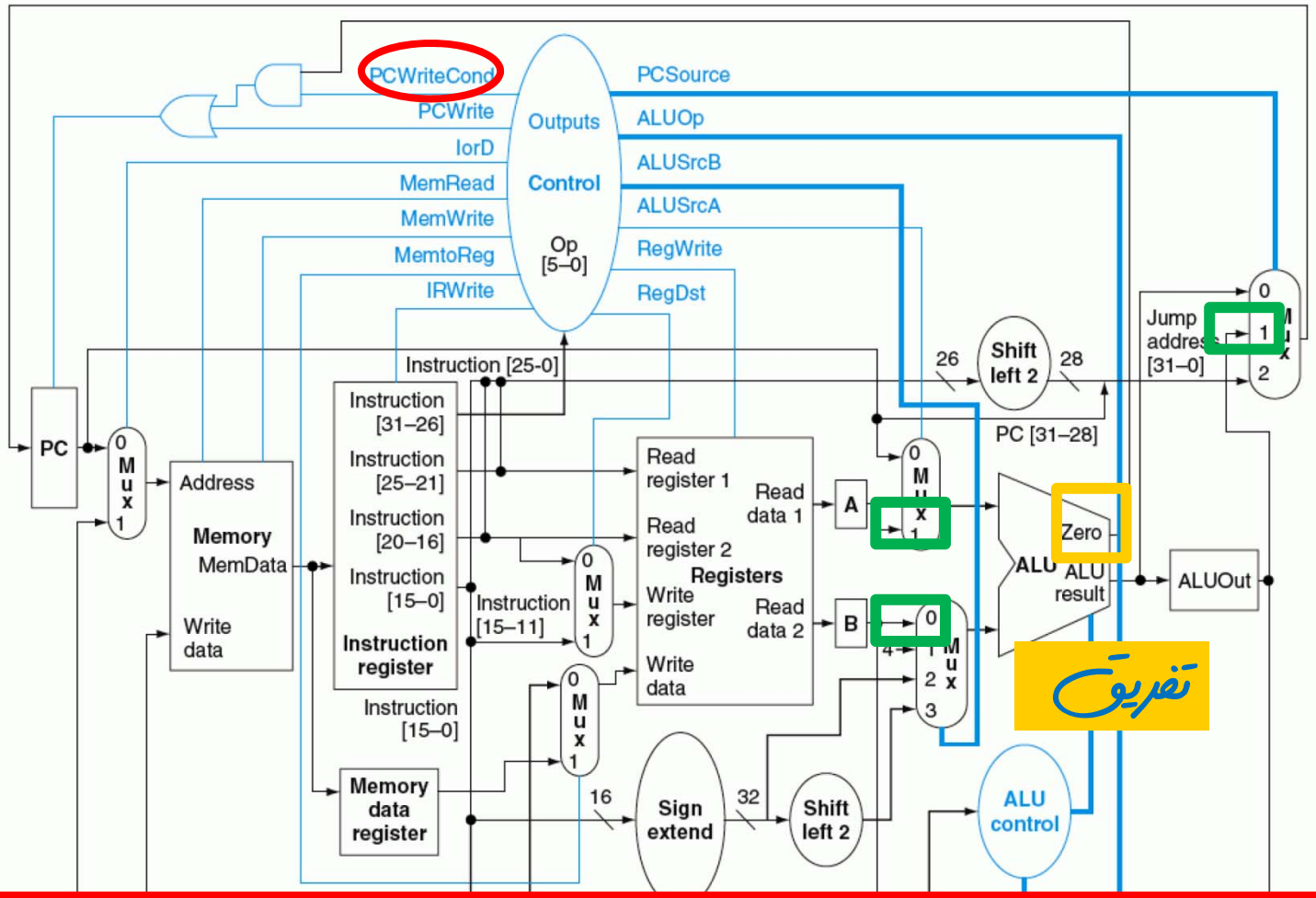
ALUOut <= A op B;



Branch

گام سوم– اجرا

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

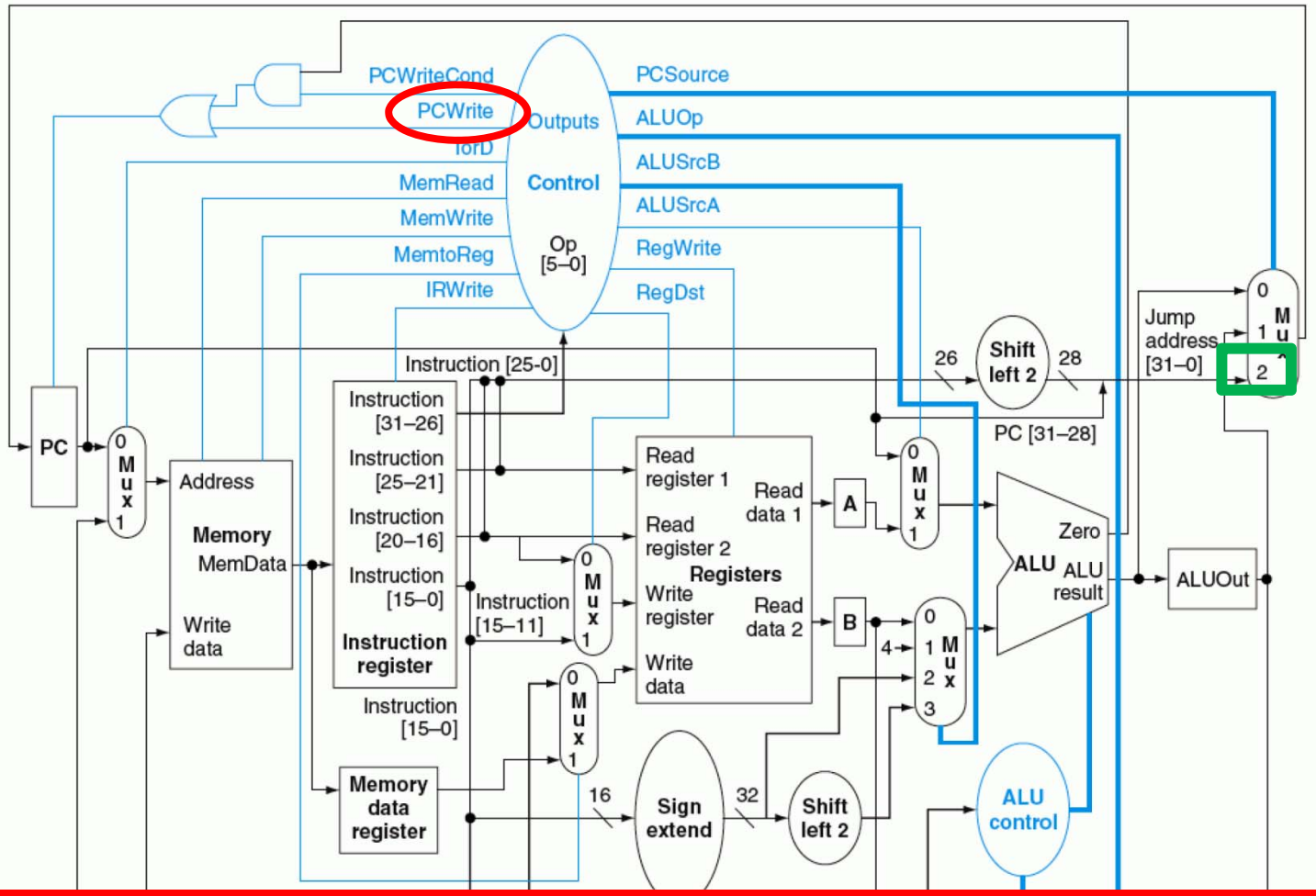


در اینجا اجرای این دستور خاتمه یافته است، به سراغ دستور بعدی خواهیم رفت

jump

گام سوم– اجرا

```
PC <= {PC[31:28],IR[25:0],2'b00};
```



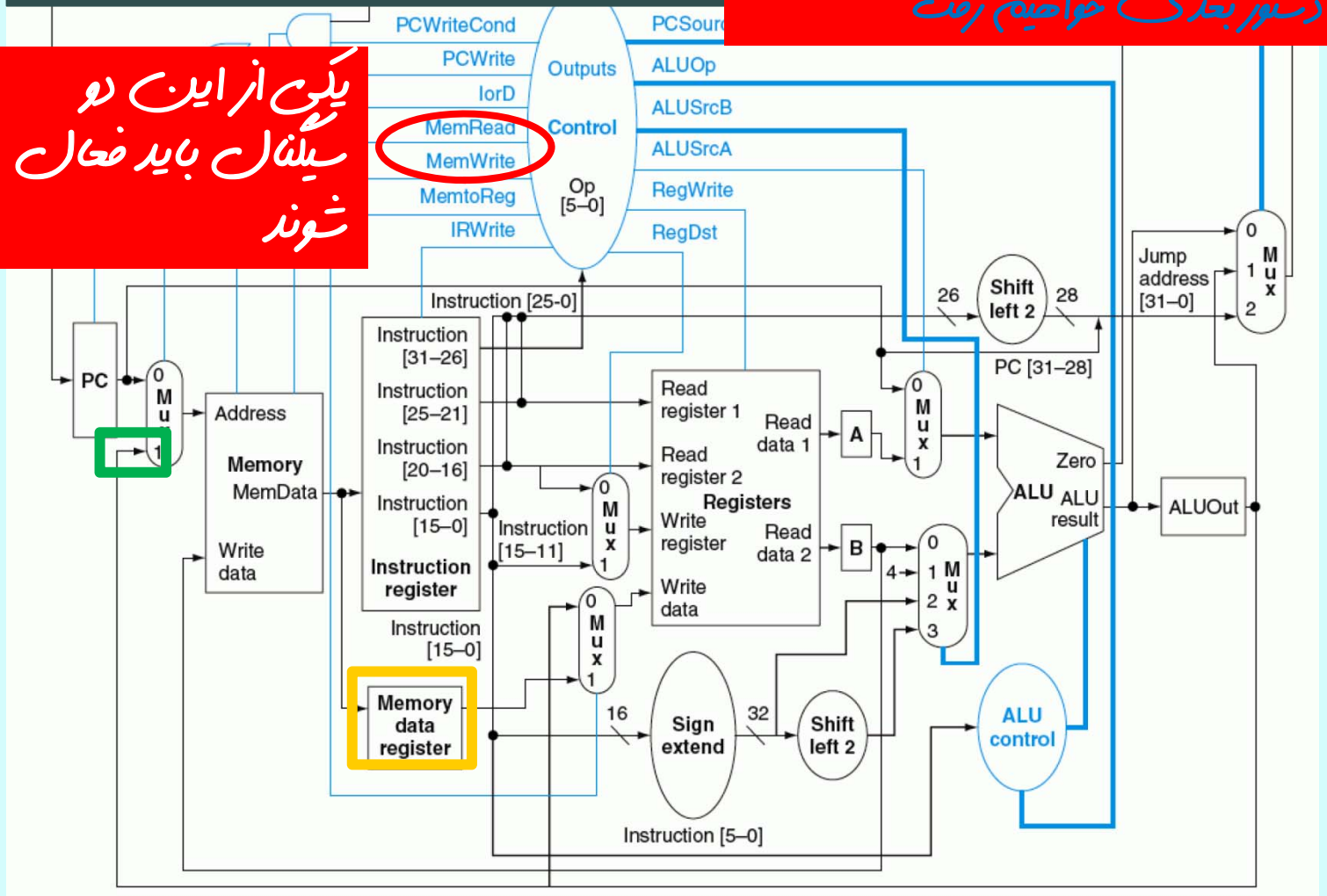
در اینجا اجرای این دستور خاتمه یافته است، به سراغ دستور بعدی خواهیم رفت

گام چهارم– دسترسی به حافظه

```
MDR <= Memory[ALUOut];
        or
Memory[ALUOut] <= B;
```

در اینجا اجرای این دستور خاتمه یافته است.
به سراغ دستور بعدی خواهیم رفت

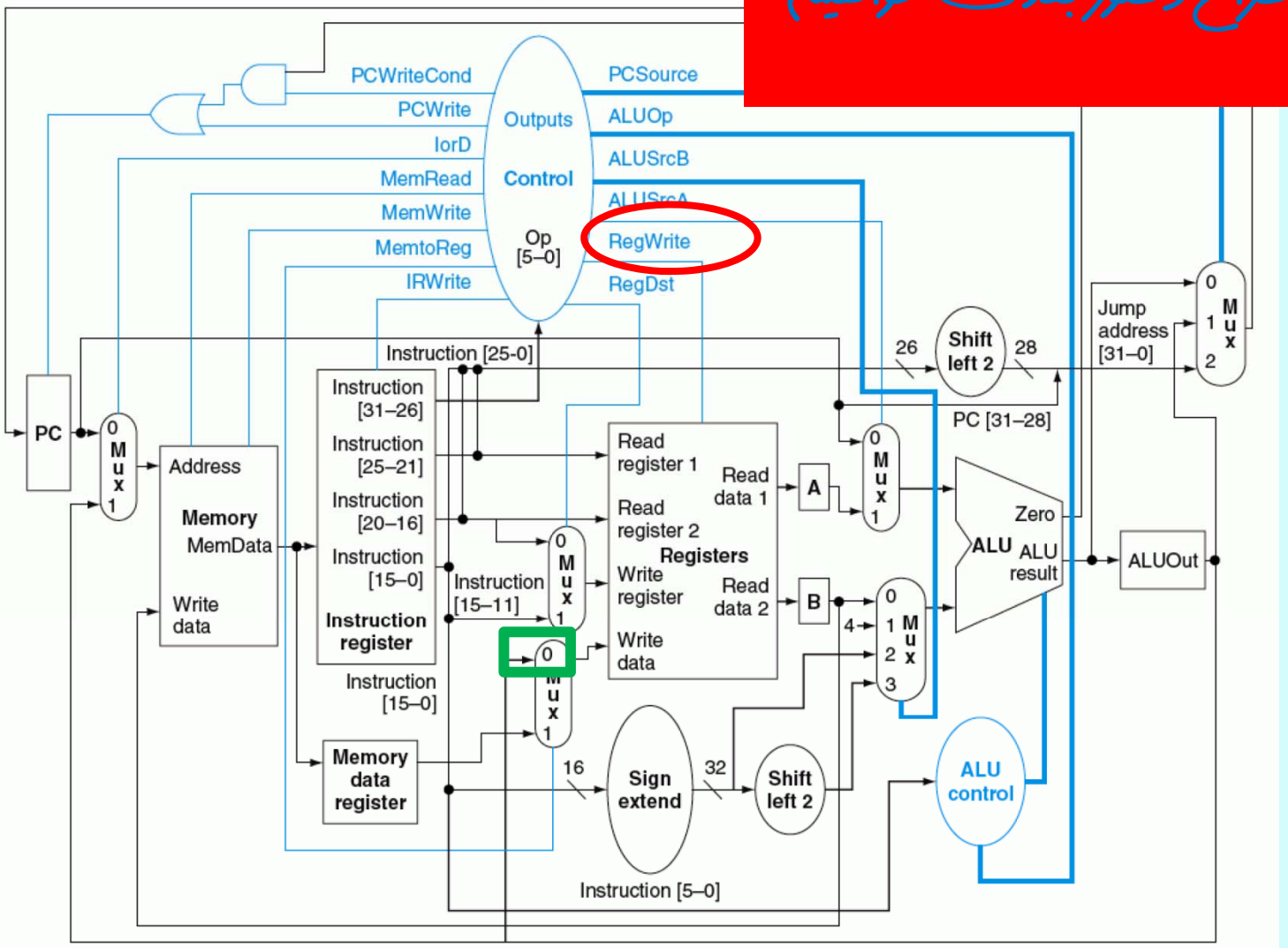
یکی از این دو سیگنال باید فعال شوند



گام چهارم– یا نوشتن نتیجه‌ی نوع R

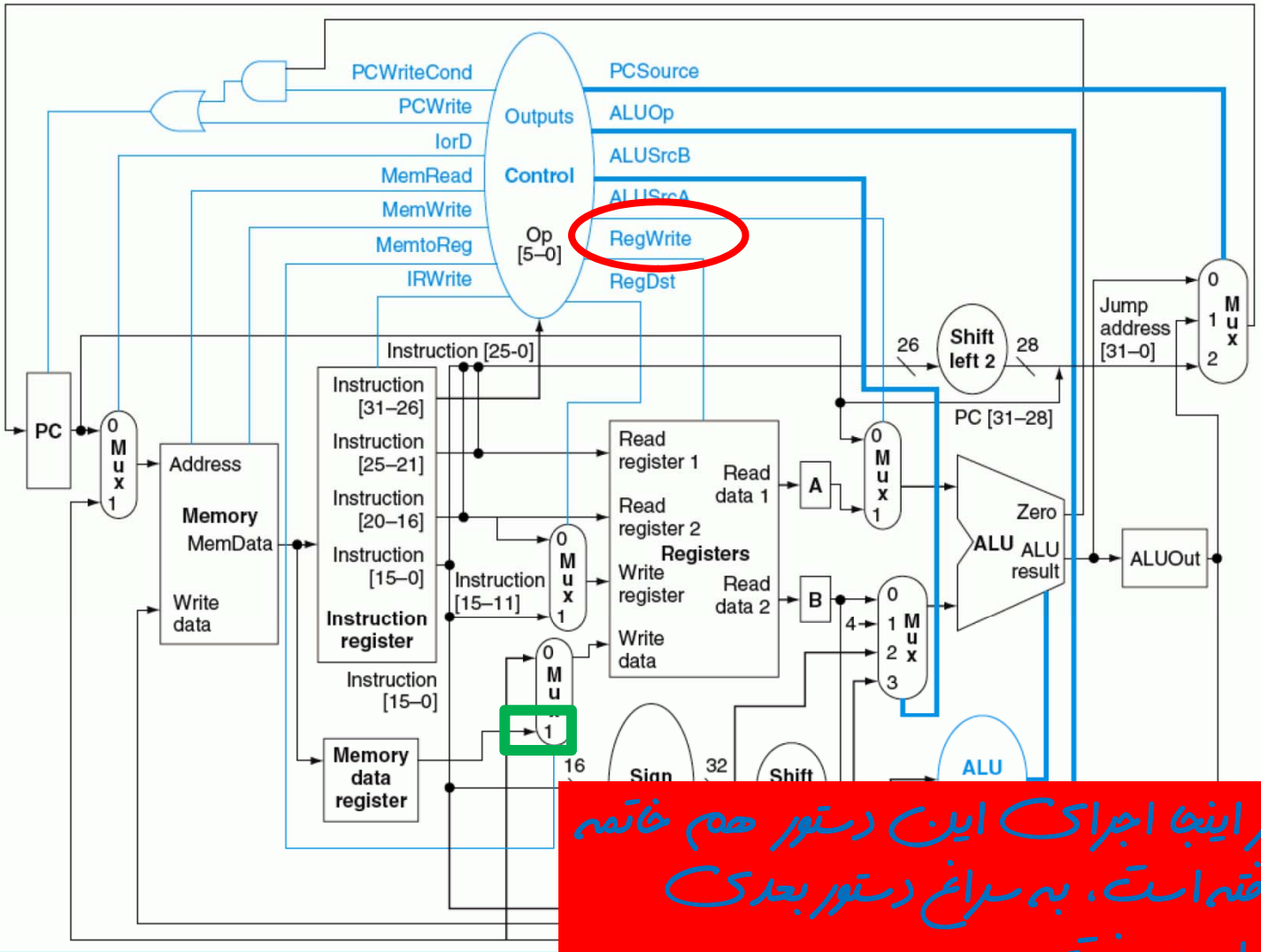
```
Reg[IR[15:11]] <= ALUOut;
```

در اینجا اجرای این دستور خاتمه یافته است، به سراغ دستور بعدی خواهیم رفت



گام پنجم– نوشتن در ثبات

```
Reg[IR[20:16]] <= MDR;
```

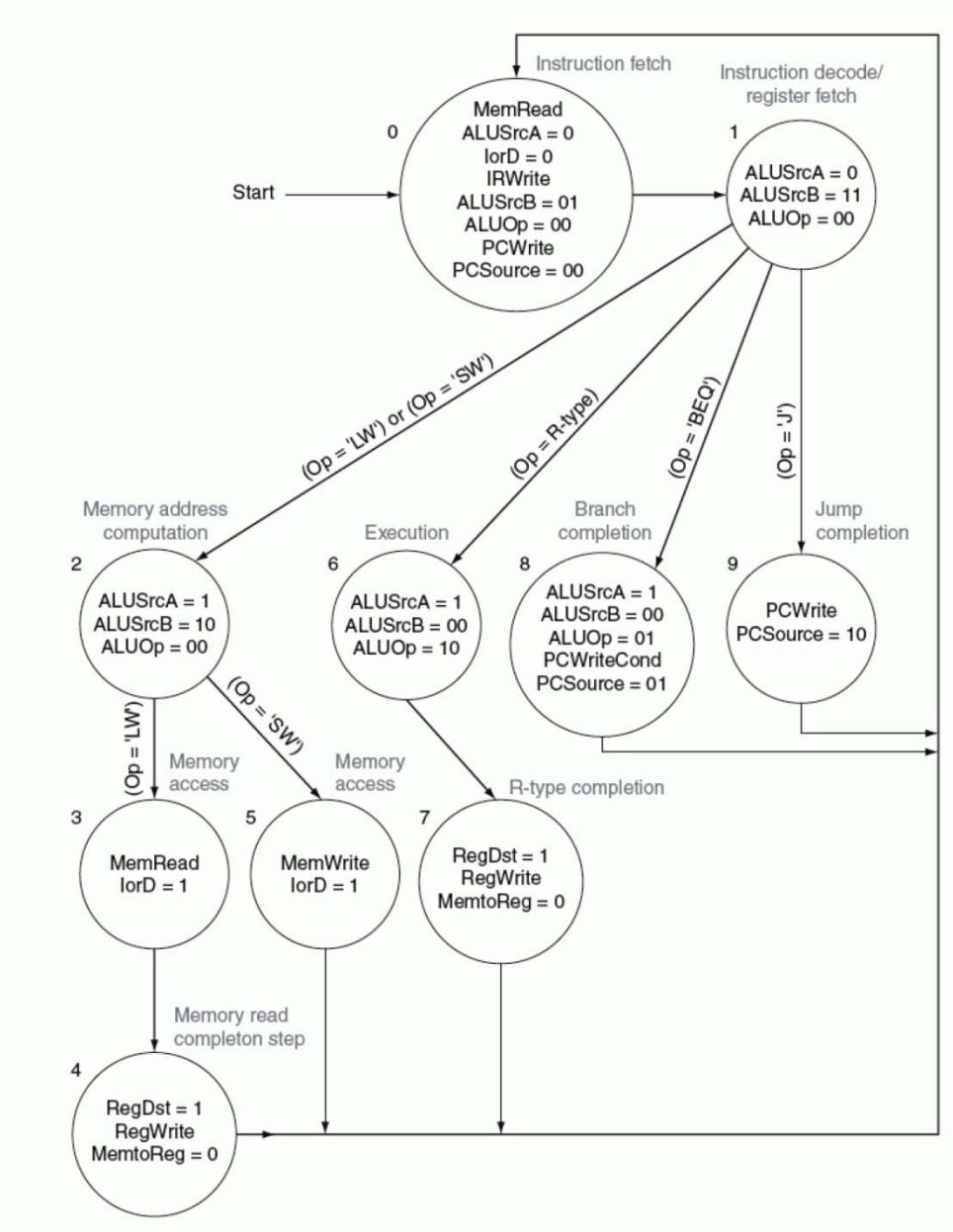


در اینجا اجرای این دستور هم خاتمه یافته است. به سراغ دستور بعدی خواهیم رفت

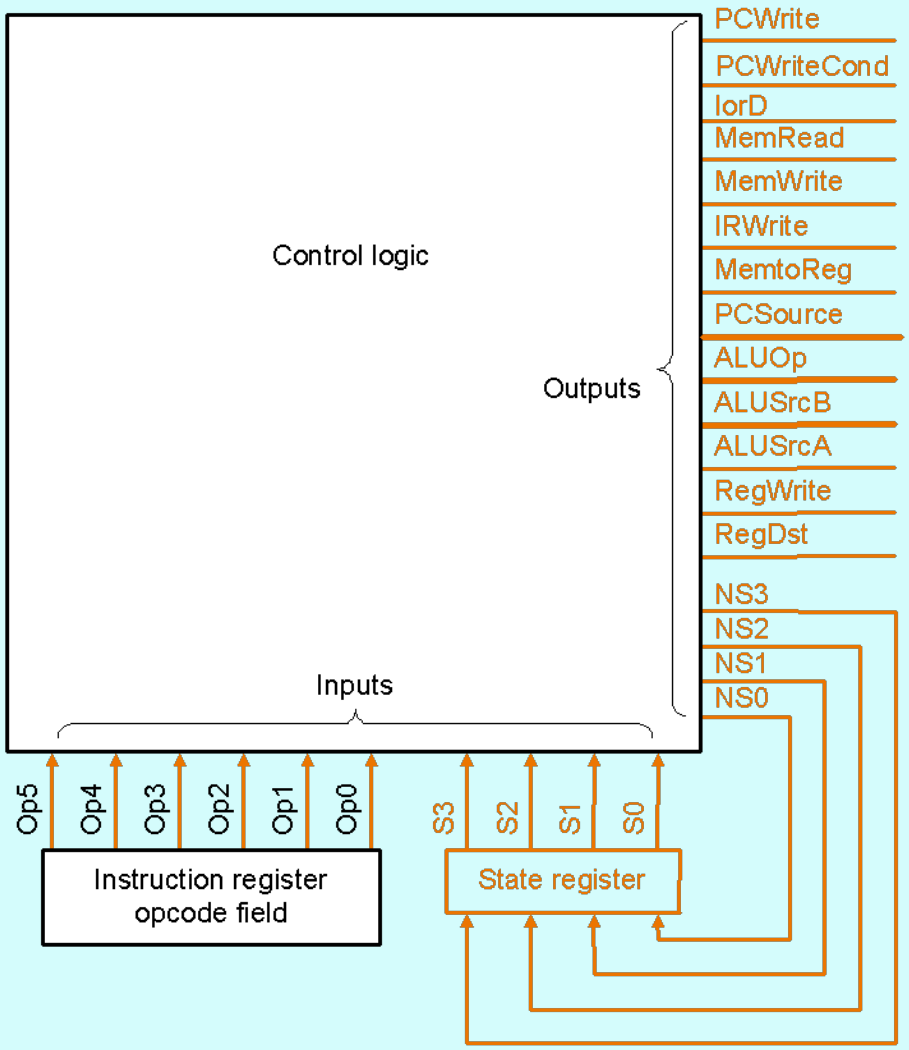
Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]),2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

ماشین حالت واحد کنترل



پیاده‌سازی


Control logic
Outputs
PCWrite
PCWriteCond
IorD
MemRead
MemWrite
IRWrite
MemtoReg
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
NS3
NS2
NS1
NS0
Inputs
Op5
Op4
Op3
Op2
Op1
Op0
S3
S2
S1
S0
Instruction register
opcode field
State register