دستورهای معمول انتقال و ریاضی در کامپیوترها

جلوی این دستورها باید به تعداد کافی OPERAND قرار داده شود.

Type	Operation Name	Description
Data Transfer	Move (transfer) انتقال از مبدا به مقصد	Transfer word or block from source to destination
	Store ذخیره از رجیستر به حافظه	Transfer word from processor to memory
	Load (fetch) بازیابی حافظه به رجیستر	Transfer word from memory to processor
	Exchange جابه جایی مبدا و مقصد	Swap contents of source and destination
	Clear (reset) ریختن صفر در مقصد	Transfer word of 0s to destination
	Set ریختن یک در مقصد	Transfer word of 1s to destination
	Push انتقال مبدا به بالای stack	Transfer word from source to top of stack
	Pop انتقال بالای stack به مقصد	Transfer word from top of stack to destination
Arithmetic	Add جمع دو عدد	Compute sum of two operands
	Subtract تفریق دو عدد	Compute difference of two operands
	Multiply ضرب دو عدد	Compute product of two operands
	Divide تقسیم دو عدد	Compute quotient of two operands
	Absolute قدر مطلق یک عدد	Replace operand by its absolute value
	Negate منفی کردن یک عدد	Change sign of operand
	Increment افزایش یک مقداری عدد	Add 1 to operand
	Decrement کاهش یک مقداری عدد	Subtract 1 from operand

دستورهای منطقی هم جز دستورهای معمول کامپیوترها هستند.

این دستورها اعمال منطقی را به شکل بیتی اعمال می کنند.

Type	Operation Name	Description
Logical	AND	Perform logical AND
	OR	Perform logical OR
	NOT (complement)	Perform logical NOT
	Exclusive-OR	Perform logical XOR
	Test بررسی یک شرط خاص و مقدار دهی به flagبراساس آن.	Test specified condition; set flag(s) based on outcome
	Compare مقایسه دو مقدار و مقدار دهی به flagبراساس آن.	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome
	Set Control Variables مقداردهی به بیت های کنترلی	Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc.
	Shift	Left (right) shift operand, introducing constants at end
	Rotate	Left (right) shift operand, with wraparound end

- مثال:

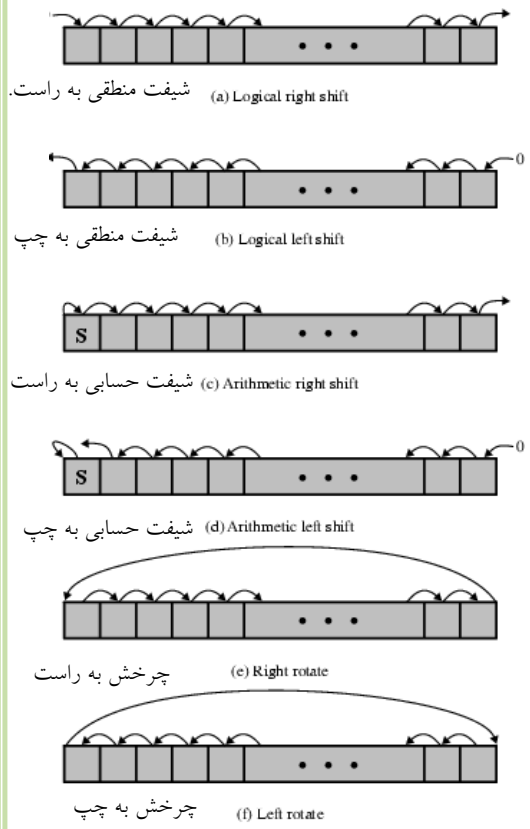
(R1) = 10100101
(R2) = 11111111

(R1) = 10100101
(R2) = 00001111

(R1) XOR (R2) = 01011010

(R1) AND (R2) = 00000101

دستورهای شیفت و چرخش از دستورهای معمول کامپیوترها هستند.
در اینجا شش نوع معمول معرفی شده است.
در پایین هم مثالی از عملکرد آنها آورده شده است.
این دستورها روی مبدا اعمال می شوند و نتیجه را هم در همان منبع ذخیره می کنند.



مثال زیر نتیجه سه بار اعمال هر یک از دستورهای مقابل را نشان می دهد.

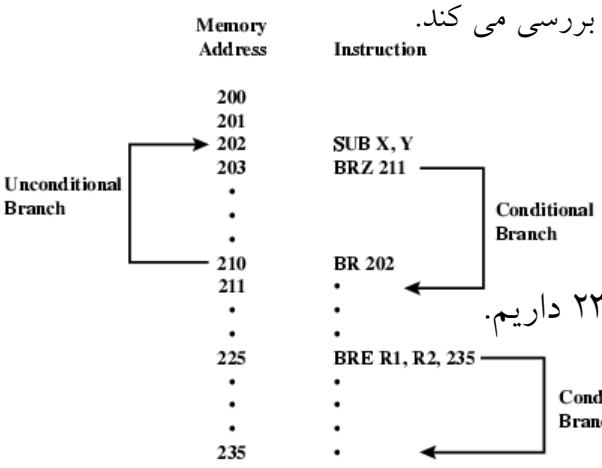
Input	Operation	Result
10100110	Logical right shift (3 bits)	00010100
10100110	Logical left shift (3 bits)	00110000
10100110	Arithmetic right shift (3 bits)	11110100
10100110	Arithmetic left shift (3 bits)	10110000
10100110	Right rotate (3 bits)	11010100
10100110	Left rotate (3 bits)	00110101

دستورهای انتقال کنترل از دستورهای اساسی هر کامپیوتری است.
با این دستورها می توان پرش، تصمیم گیری و زیربرنامه را ایجاد کرد.

Transfer of Control	Jump (branch) پرش اجرای برنامه به آدرس داده شده	Unconditional transfer; load PC with specified address
تصمیم گیری →	Jump Conditional پرش اجرای برنامه به آدرس داده شده اگر شرط درست باشد.	Test specified condition; either load PC with specified address or do nothing, based on condition
CALL ایجاد زیر برنامه	Jump to Subroutine پرش اجرای برنامه به آدرس داده شده و ذخیره مقدار فعلی PC	Place current program control information in known location; jump to specified address
	Return پرش اجرای برنامه به آدرس ذخیره شده	Replace contents of PC and other register from known location
	Execute	Fetch operand from specified location and execute as instruction; do not modify PC
	Skip نادیده گرفتن دستور بعدی	Increment PC to skip next instruction
تصمیم گیری →	Skip Conditional نادیده گرفتن دستور بعدی با برقراری شرط	Test specified condition; either skip or do nothing based on condition
	Halt متوقف کردن اجرای برنامه	Stop program execution
تصمیم گیری →	Wait (hold) متوقف کردن اجرای برنامه ادامه با برقرار شدن شرط	Stop program execution; test specified condition repeatedly; resume execution when condition is satisfied
	No operation	No operation is performed, but program execution is continued

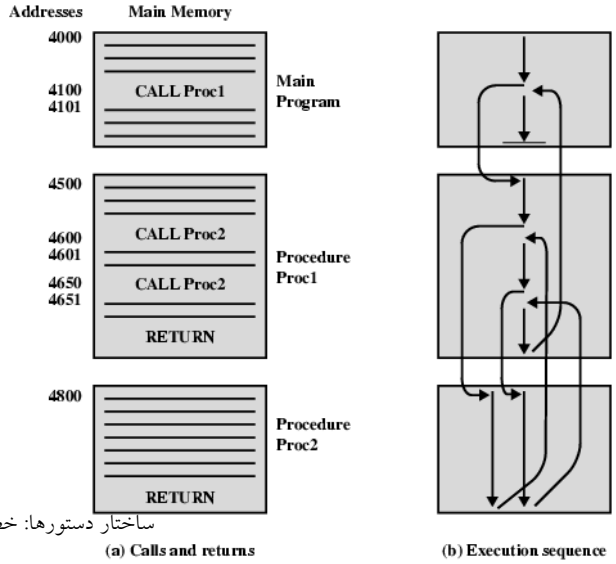
نمونه ای از اجرای دستورهای پرش و پرش شرطی

- در برنامه زیر در خط ۲۰۳ دستور پرشی شرطی به خط ۲۰۲ انجام می شود.
  - دستور پرش شرطی صفر بودن نتیجه قبلی را بررسی می کند.
  - اگر نتیجه صفر باشد، پرش انجام می شود.
  - اگر نباشد، خط ۲۰۴ اجرا می شود.
- در خط ۲۰۲ دستور پرش به ۲۰۲داریم.
  - پرش همیشه انجام می شود.
- در خط ۲۲۵ دستور پرش شرطی به خط ۲۳۵ داریم.
  - در این دستور R1 و R2 مقایسه می شوند.
  - اگر برابر باشند پرش انجام می شود.
  - اگر نباشند خط ۲۲۶ اجرا می شود.
- دقت کنید که پرش های شرطی روش اصلی تصمیم گیری در زبان ماشین است.



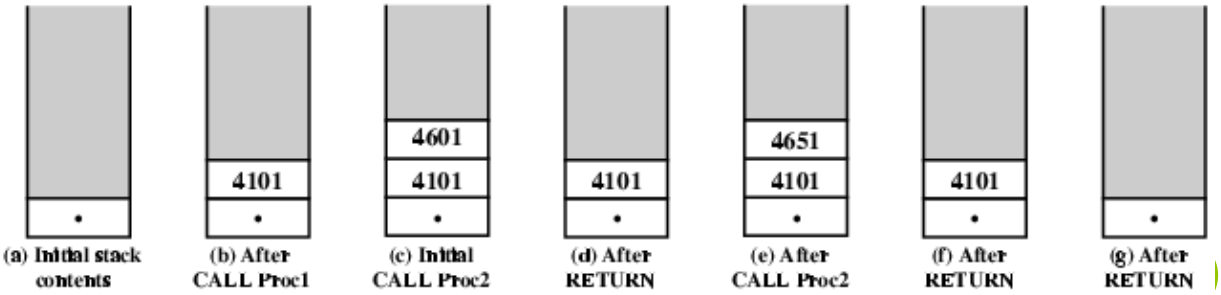
معمول است که زیربرنامه ها در زبان ماشین با دستورهای CALL و RETURN تشکیل شود.

- در شکل زیر مثالی از اجرای این دو دستور دیده می شود.
- روند فلش را ها پشت سرهم دنبال کنید.
- دقت کنید در زیربرنامه اول، زیربرنامه های دیگری فراخوانی شده است.

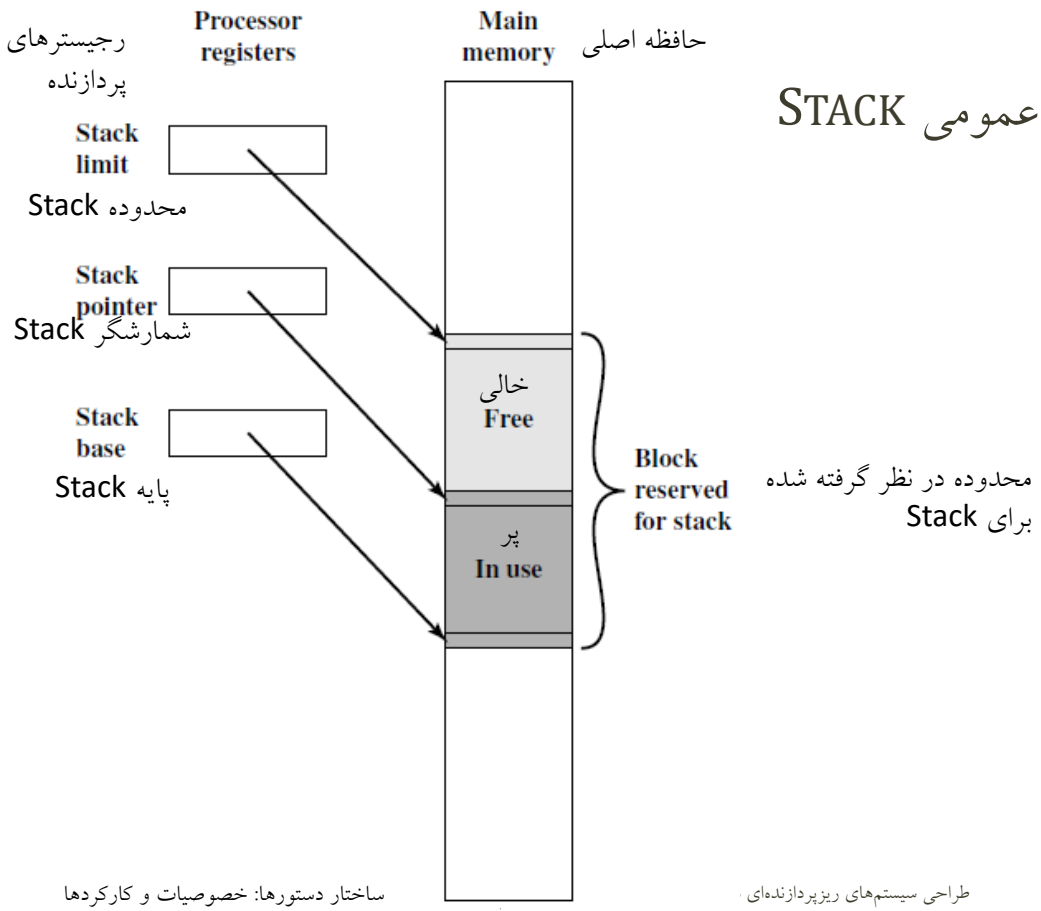


در دستور CALL آدرس برگشت معمولاً در فضایی به نام STACK ذخیره می شود. با دستور RETURN از این فضا برداشته می شود.

- Stack: مکانی قراردادی در حافظه اصلی است. (پس جای جدیدی نیست.)
  - ساختار عمومی آن در اسلاید بعدی آورده شده است.
- داده ای که آخر قرار داده شده اول برداشته می شود. (مانند کتاب های داخل یک جعبه)
- یک رجیستر آدرس مکان خالی این فضا را نشان می دهد: SP
- مثال زیر روند ذخیره آدرس مثال اسلاید قبل را در stack نشان می دهد.
- علاوه بر ذخیره آدرس مقدارهای دیگری هم می تواند در Stack ذخیره شود.
  - این کار با دو دستور PUSH و POP در اختیار برنامه نویس است.



ساختار عمومی STACK



دستورهای I/O و دستورهای تبدیل از دیگر دستورهای متداول هستند.

- دستورهای I/O برای صحبت با ماجول I/O هستند.
  - خواندن و نوشتن داده در مکانی در ماجول I/O (مشابه حافظه)
  - دستورهایی برای فرمان به دستگاه ها و بررسی وضعیت آنها هم وجود دارد.
- دستورهای تبدیل دارای کاربرد خاص هستند. (برای مطالعه)

Input/Output	Input (read) انتقال از مکانی در مجول I/O به مقصد	Transfer data from specified I/O port or device to destination (e.g., main memory or processor register)
	Output (write) انتقال از مبدا به مکانی در مجول I/O	Transfer data from specified source to I/O port or device
	Start I/O فرمان آغاز به کار دستگاه	Transfer instructions to I/O processor to initiate I/O operation
	Test I/O انتقال وضعیت دستگاه به مقصد	Transfer status information from I/O system to specified destination
Conversion	Translate	Translate values in a section of memory based on a table of correspondences
	Convert	Convert the contents of a word from one form to another (e.g., packed decimal to binary)

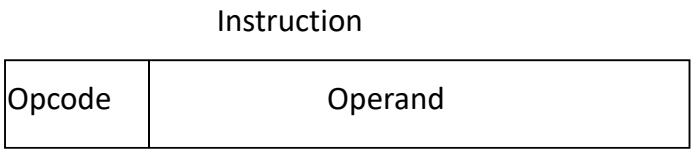
مجموعه دستورها: مودهای آدرس دهی و شکل دستورها

- در ادامه معرفی دستورهای ماشین دو موضوع مهم درباره دستورها را مطرح می کنیم.
- مودهای آدرس دهی
  - اینکه در هر دستور مقدار عملگر (Operand) از کجا آورده شود می شود: مود آدرس دهی.
- مودهای آدرس دهی متداول عبارتند از:
  - Immediate (صریح)
  - Direct (مستقیم)
  - Indirect (غیرمستقیم)
  - Register (رجیستر)
  - Register Indirect (غیرمستقیم از طریق رجیستر)
  - Displacement (Indexed) (جابجایی)
  - Stack (پشته)
- شکل دستور
  - هر بخش کد ماشین چگونه تعریف می شود؟ (چند تعداد بیت به Opcode و operandها اختصاص می یابد؟)

مود آدرس دهی IMMEDIATE (صریح)

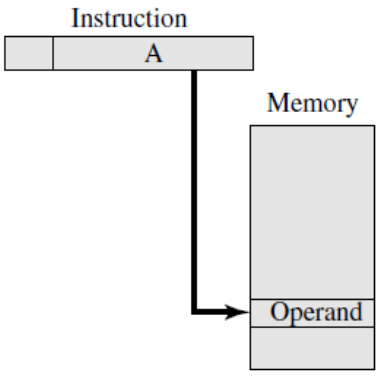
در این روش مقدار اپرند در دستور آورده می شود.

- در این روش Operand جزیی از دستور است.
- یعنی مقداری که باید روی آن عملیات انجام شود درون دستور آورده شده است.
- مثال: ADDI R1,6 (یعنی R1 را با ۶ جمع کن و نتیجه را در R1 ذخیره کن.)
- در دستور بالا عملگر 6 به صورت صریح آورده شده است.
- در این روش آدرس دهی دسترسی به حافظه اصلی نداریم. بنابرین اجرای سریعی دارد.
- ولی محدوده عددی کوچکی خواهد داشت.
- شکل زیر یک دستور یک اپرندی را با آدرس دهی صریح نشان می دهد



مود آدرس دهی مستقیم (DIRECT).

در این مود مقدار اپرند از حافظه اصلی برداشته می شود.



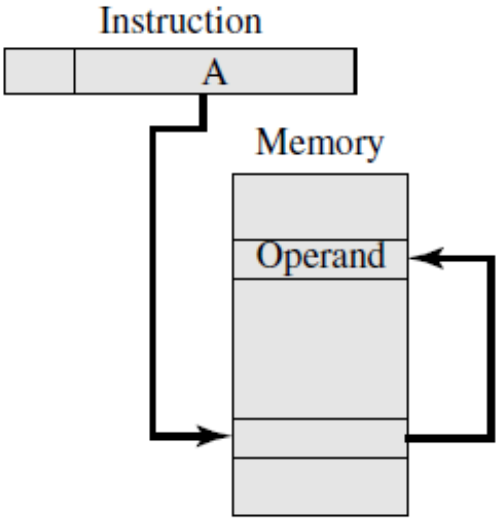
- در این روش آدرس اپرند در حافظه اصلی در دستور آورده می شود.
- اصطلاحاً می گویند: در این روش (آدرس موثر) A =EA
- یعنی آدرسی که باید اپرند از آن برداشته شود مستقیماً در دستور آورده شده است.
- مثال: ADD R1,A (یعنی محتوای آدرس A را با R1 جمع کن و نتیجه را در R1 ذخیره کن.)
- در دستور بالا A با مود آدرس دهی مستقیم آورده شده است.
- به یک دسترسی به حافظه برای برداشتن اپرند نیاز است.
- محاسبه ای برای تعیین آدرس انجام نمی شود. اما فضای آدرس دهی محدود است.

مود آدرس دهی غیر مستقیم (INDIRECT).

در این مود مقدار اپرند با واسطه از حافظه اصلی برداشته می شود.

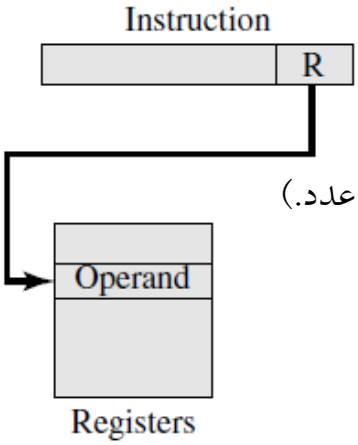
- در این روش بخش اپرند آدرسی را شامل می شود.. محتوای این آدرس، خود نشاندهنده آدرس مکان اپرند در حافظه اصلی است.
- در این روش می گویند. EA = (A) (یعنی: آدرس موثر در محتوای آدرس A است.)
- مثال ADD R1, (A) (یعنی محتوای آدرس A را پیدا کن و اپرند را از محتوای آن آدرس بیاور، سپس آن را با R1 جمع کن و نتیجه را در R1 ذخیره کن.)
- فضای آدرس دهی بزرگی خواهیم داشت. (یعنی می توان به مکان های بیشتری از حافظه دسترسی پیدا کرد.)
- اگر طول کلمه حافظه n باشد. به 2^n آدرس دسترسی خواهیم داشت.
- به دوبار دسترسی به حافظه نیاز خواهد بود. بنابرین اجرای دستور کندتر ست.

مود آدرس دهی غیر مستقیم (INDIRECT). (ادامه)



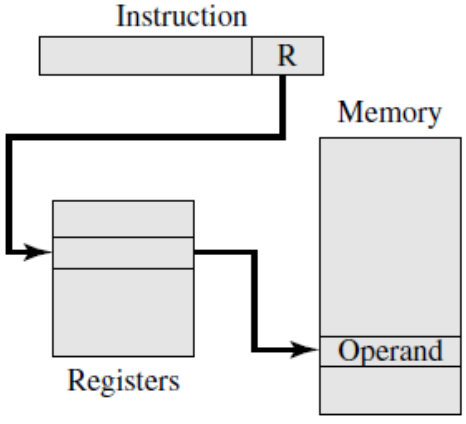
مود آدرس دهی رجیستر(REGISTER).
در این مود مقدار اپرند از داخل یکی از رجیسترها برداشته می شود

- در این روش شماره رجیستر مورد نظر در دستور آورده می شود. و مقدار اپرند از درون این رجیستر برداشته می شود.
- بنابرین EA = R: (آدرس موثر برابر است با شماره رجیستر)
- مثال: ADD R1, R2
  - یعنی محتوای R1 را با R2 جمع کن. نتیجه را در R1 بریز.
- معمولاً تعداد رجیسترها بسیار کم است (در حدود ۳۲ یا ۶۴ عدد.)
- بنابرین اندازه آدرس در دستور بسیار کوچک می شود.
- دستور کوچک می شود = fetch سریعتر.
- دسترسی به حافظه هم نداریم. = اجرای سریع.
- اجرای بسیار سریع دستورها.
- محدوده آدرس دهی محدود = نیاز به برنامه نویسی خوب.



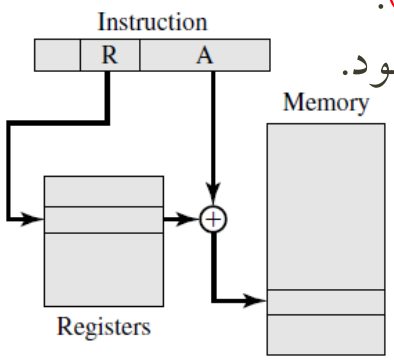
مود آدرس دهی غیر مستقیم با رجیستر(REGISTER INDIRECT).
در این مود مقدار اپرند از حافظه برداشته می شود.

- در این روش در دستور شماره رجیستر آورده می شود. محتوای این رجیستر آدرسی را در حافظه اصلی نشان می دهد که اپرند باید از آنجا برداشته شود.
- بنابرین EA = (R) (یعنی آدرس موثر درون رجیستر R ذخیره شده است.)
- مثال ADD R1, (R2)
  - یعنی R1 را با محتوای مکانی از حافظه که دارای آدرسی برابر با محتوای R2 است، جمع کن و نتیجه را در R1 ذخیره کن.
- به فضای بزرگی در حافظه دسترسی داریم (2^n)
- تنها به یک دسترسی به حافظه نیاز داریم.
- اجرای کمی سریعتر از روش غیرمستقیم ولی کندتر از روش رجیستر دارد.
- اندازه دستور هم کوچک است.
  - (به خاطر تعداد کم رجیسترها)



مود آدرس دهی جابجایی (DISPLACEMENT).

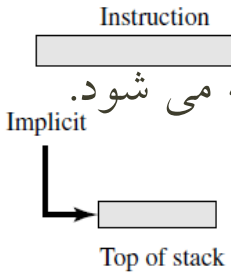
در این مود مقدار اپرند از حافظه برداشته می شود.



- در این روش برای به دست آوردن آدرس موثر هم از رجیستر و هم از مکان حافظه استفاده می شود.
- آدرس موثر در این روش: EA = A + (R)
- مثال: ADD R1, A(R2).
  - یعنی محتوای R2 را با محتوای مکان A حافظه جمع کن نتیجه را به عنوان آدرس حافظه به کار ببر و محتوای آن مکان را پیدا کن. سپس آن را با R1 جمع کن و نتیجه را R1 ذخیره کن.
  - در این روش A مقدار پایه و R مقدار جابجایی نام دارد.
- در دستور برای یک اپرند دو آدرس اختصاص یافته است. (مانند شکل بالا)
- این روش برای پیاده سازی دسترسی به آرایه ها طراحی شده است.
- A ابتدای آغاز آرایه و R2 اندیس آرایه را نشان می دهد. (و یا برعکس)
- دقت کنید که عملیات جمع سخت افزاری انجام می شود.

مود آدرس دهی پشته (STACK).

در این مود مقدار اپرند از حافظه و از بالای STACK برداشته می شود.



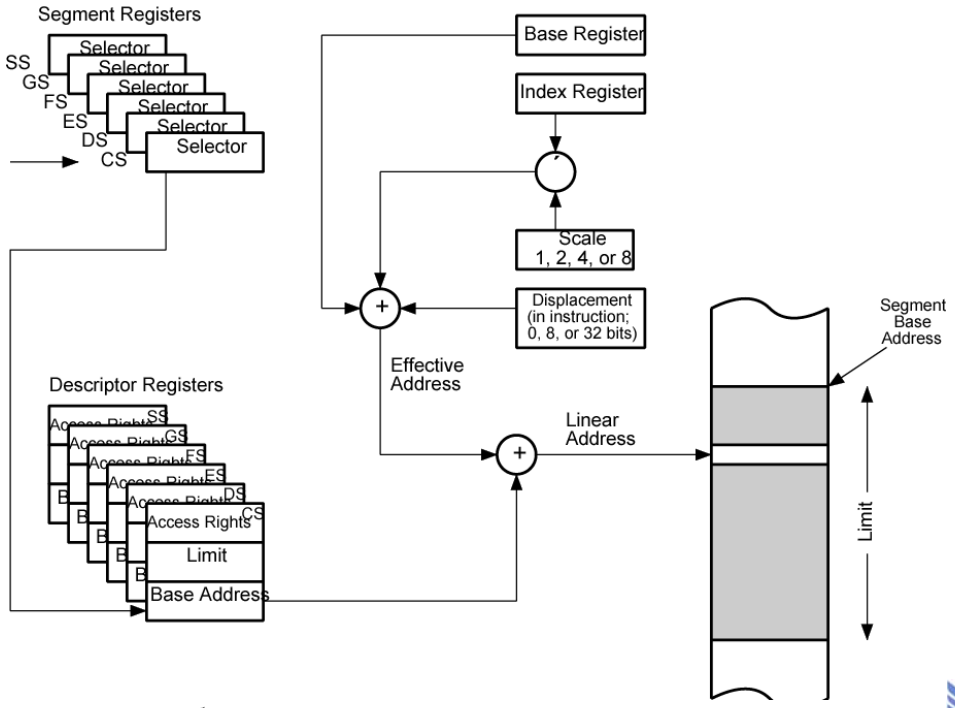
- در این روش در دستور هیچ آدرس دهی انجام نمی شود.
- به طور ضمنی در این روش اپرند در بالای stack قرار دارد.
- مثال: ADDS
  - یعنی دو مقدار بالای stack را با هم جمع کن و نتیجه را در بالای stack ذخیره کن.

- دقت کنید که در دستورها هر دو اپرند می توانند دارای روش های آدرس دهی متفاوتی باشند. چند مثال را پایین آوردیم.
- ADD (R1), A (مبدا: مستقیم و مقصد: رجیستر غیر مستقیم.)
- ADD A(R1),R2 (مبدا: رجیستر و مقصد: جابجایی)
- ADDI (A), 3 (مبدا: صریح و مقصد: غیر مستقیم)
- PUSH R1 (مبدا: رجیستر و مقصد: stack)
  - این دستور مقدار رجیستر R1 را در بالای stack ذخیره می کند.

مودهای آدرس دهی در خانواده x86 (شکل برای مطالعه)

برای به دست آوردن آدرس موثر چندین مقدار با هم جمع می شود.

این خانواده مودهای متعدد و پیچیده ای برای محاسبه آدرس دارد.



مودهای آدرس دهی در خانواده x86 (جدول برای مطالعه) (ادامه)

Mode	Algorithm
Immediate	Operand = A
Register Operand	LA = R
Displacement	LA = (SR) + A
Base	LA = (SR) + (B)
Base with Displacement	LA = (SR) + (B) + A
Scaled Index with Displacement	LA = (SR) + (I) × S + A
Base with Index and Displacement	LA = (SR) + (B) + (I) + A
Base with Scaled Index and Displacement	LA = (SR) + (I) × S + (B) + A
Relative	LA = (PC) + A

- هرچه مودهای آدرس دهی بیشتر باشد ساخت CPU پیچیده تر می شود.
- برعکس در خانواده ARM: تعداد مودهای آدرس دهی کمتر و ساده تر است.
- به عنوان مثال دسترسی به حافظه فقط با دستورهای load و store انجام می شود.
- سایر دستورها (محاسبات و ...) فقط به رجیسترها دسترسی دارند.

شکل دستورها به معنای آن است که بیتهای کد ماشین چگونه به بخش های مختلف تخصیص داده می شوند.

- همانطور که گفته شد، کد ماشین شامل Operand + Opcode ها است.
- اینکه چند بیت به Opcode اختصاص دهیم و چند بیت به operand ها و کلاً طول دستور چقدر باشد وابسته به عوامل متعددی است.
- از جمله: تعداد دستورها، اندازه حافظه، سازماندهی حافظه، سرعت اجرای دستورها، تعداد مودهای آدرس دهی، تعداد رجیسترها.
- برخی کامپیوترها یک شکل ثابت برای کد ماشین (دستور) خود ندارند.
- بلکه چندین شکل مختلف دارند که هر یک را برای یک مجموعه دستور تخصیص داده اند.
- در اسلاید بعد شکل دستورهای یک کامپیوتر دیده می شود.
- دستورها به سه گروه تقسیم شده است. (دسترسی به حافظه، I/O، دسترسی به رجیستر)
- دقت کنید که اندازه Opcode در هر یک از آنها متفاوت است.

شکل مجموعه دستورهای یک کامپیوتر

Memory Reference Instructions

Opcode			D/I	Z/C	Displacement						
0		2	3	4	5						11

Input/Output Instructions

1	1	0	Device							Opcode	
0		2	3						8	9	11

Register Reference Instructions

Group 1 Microinstructions

1	1	1	0	CLA	CLL	CMA	CML	RAR	RAL	BSW	IAC
0	1	2	3	4	5	6	7	8	9	10	11

Group 2 Microinstructions

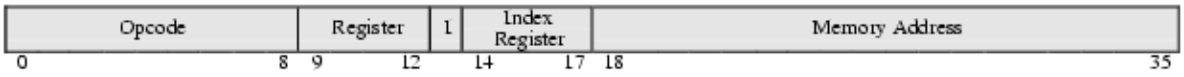
1	1	1	1	CLA	SMA	SZA	SNL	RSS	OSR	HLT	0
0	1	2	3	4	5	6	7	8	9	10	11

Group 3 Microinstructions

1	1	1	1	CLA	MQA	0	MQL	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11

D/I = Direct/Indirect address
Z/C = Page 0 or Current page
CLA = Clear Accumulator
CLL = Clear Link
CMA = CoMplement Accumulator
CML = CoMplement Link
RAR = Rotate Accumultator Right
RAL = Rotate Accumulator Left
BSW = Byte SWap
IAC = Increment ACcumulator
SMA = Skip on Minus Accumulator
SZA = Skip on Zero Accumulator
SNL = Skip on Nonzero Link
RSS = Reverse Skip Sense
OSR = Or with Switch Register
HLT = HaLT
MQA = Multiplier Quotient into Accumulator
MQL = Multiplier Quotient Load

برخی از کامپیوترها هم یک شکل ثابت برای همه دستورهاشان دارند.

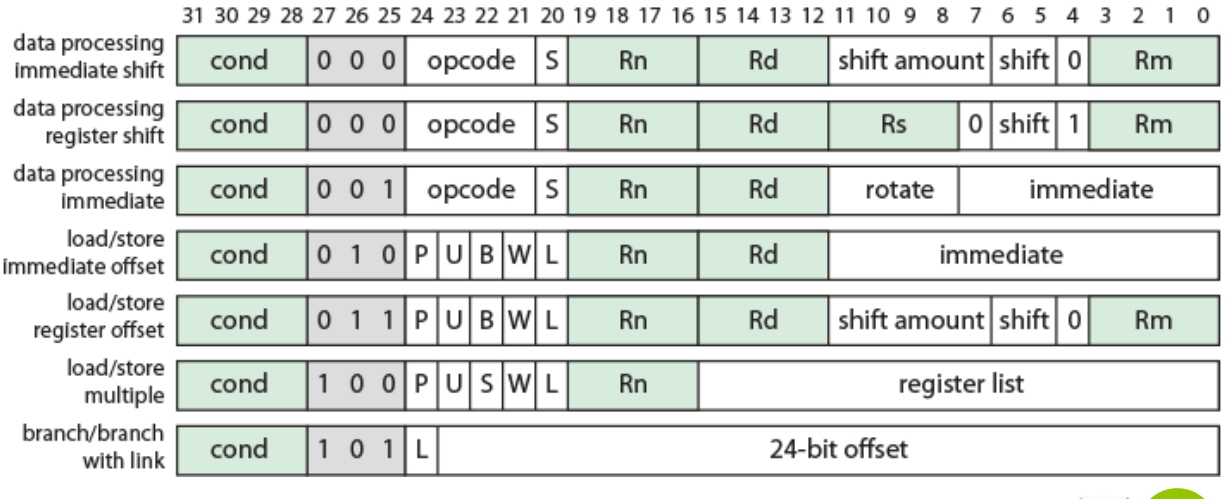


I = indirect bit

- کامپیوتر بالا ۹ بیت opcode دارد: پس دارای ۵۱۲= $۲^۹$ دستور مختلف است.
- دو اپرند دارد. یکی مقصد و دیگری مبدا است.
- یک اپرند با آدرس دهی رجیستر است. (از بیت ۹ تا ۱۲) (۴ بیت = ۱۶ رجیستر)
- اپرند دیگر با آدرس دهی جابجایی است.
- برای این اپرند با دوبخش مشخص شده: یک بخش برای رجیستر (بیت ۱۴ تا ۱۷) و یک بخش دیگر برای آدرس حافظه قرار داده شده است (بیت ۱۸ تا ۳۵).
- ۱۶ رجیستر می تواند انتخاب شود. و 2^{18} مکان آدرس به عنوان پایه می تواند انتخاب شود.
- در بیت ۱۳ هم می توان آدرس دهی اپرند اول را از رجیستر به رجیستر غیرمستقیم تغییر داد.

مثال دیگر: مجموعه دستورهای کامپیوترهای ARM (جدول برای مطالعه)

- طول دستور ثابت است. اما گروه های مختلفی تعریف شده است.
- دقت کنید که اندازه opcede مختلف است و تعداد اپرندها هم برای هر دستور متفاوت است

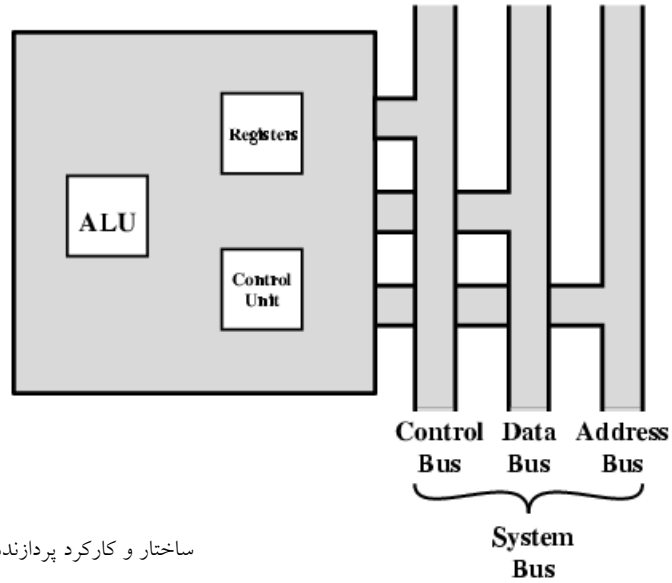


ساختار و کارکرد پردازنده

- در این بخش می خواهیم کمی بیشتر درباره ساختار رجیسترهای داخل CPU، نحوه انصال اجزا داخل CPU و همچنین نحوه اجرای دستورها کمی بیشتر بدانیم.
- تا کنون می دانیم که وظیفه پردازنده برای اجرای یک دستور اینگونه است.
  1. Fetch دستور از حافظه.
  2. تفسیر دستور.
  3. (درصورت لزوم) برداشتن داده از حافظه یا رجیستر.
  4. (در صورت لزوم) انجام پردازش روی داده.
  5. (در صورت لزوم) نوشتن نتیجه در حافظه یا رجیستر
- برای اجرای دستور بعدی PC افزایش می یابد و این روند تکرار می شود.

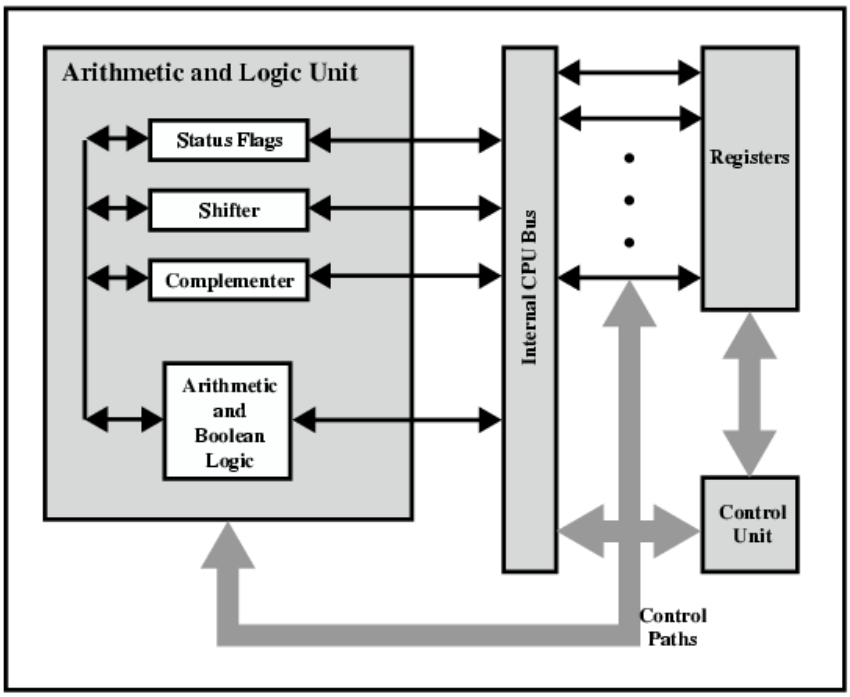
ارتباط CPU با باس سیستم

- همچنین می دانیم که CPU اینگونه با باس سیستم در ارتباط است.
- و CPU هم دارای این سه بخش است (که البته درباره واحد کنترل هنوز چیزی نگفته ایم.)



ساختار داخل CPU

ارتباط بین بخش های مختلف داخل CPU را با دقت بیشتری می بینیم.



ساختار داخل CPU (در اینجا شکل اسلاید قبل را توضیح می دهیم.)

- با بخش ALU تا حد زیادی آشنا هستیم. در شکل بخش هایی آورده شده است.
  - یک بخش ریاضی و منطقی داریم (بخش پایین): عملیات ریاضی چهارگانه، و عملیات منطقی (AND و Orو....) داخل اینجا انجام می شود.
  - یک بخش مکمل گیر داریم. که عملیات مکمل ۲ یا منفی کردن را انجام می دهد.
  - یک مدار برای انجام دستورهای شیفت داریم.
  - و یک رجیستر flag هم داخل ALU داریم که وضعیت محاسبات در آن گزارش می شود. (مانند سرریز شدن، carry، علامت نتیجه و ....)
- رجیسترها هم که حافظه داخلی CPU هستند کنار هم قرار دارند.
- برای اتصال بخش های داخلی CPU هم از باس استفاده شده است. دقت کنید که این باس داخلی CPU است و متفاوت از باس سیستم است. (آدرس مجزا داریم.)
- یک بخش کنترل هم داریم: به نظر می رسد که این بخش فرمان ها را به همه بخش ها می فرستد (از جمله باس، ALU و رجیسترها).
- یک شباهت سازی با اغماض: به نظر می رسد CPU خودش شبیه یک کامپیوتر است. و بخش کنترل هم CPU آن است. (تا حدود زیادی این قضیه درست است.)

در اینجا کمی بیشتر درباره رجیسترها صحبت می کنیم.

- می دانیم که
  - فضای کار داخلی CPU رجیسترها هستند. (برای نگهداری موقتی)
  - تعداد و کاربرد رجیسترها در کامپیوترهای مختلف متفاوت است.
  - سریع ترین حافظه ای هستند که کامپیوتر به آنها دسترسی دارد. (از نوع فلیپ فلاپ)
- رجیسترها در دو گروه طبقه بندی می شوند:

1. رجیسترهای قابل دید توسط کاربر (برنامه نویس):
   - یعنی کاربر می تواند در دستورهایش از این رجیسترها استفاده کند. مقدار بدهد یا مقدار بخواند.
2. رجیسترهای غیر قابل دید توسط کاربر.
   - این رجیسترها هر کدام وظیفه ای را انجام می دهند تا دستور اجرا شود ولی از دید کاربر مخفی هستند.

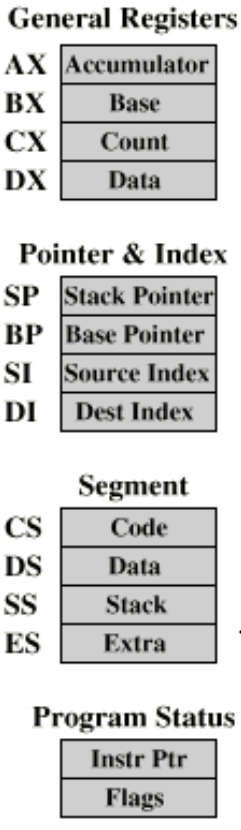
کمی بیشتر درباره رجیسترهای قابل دید توسط کاربر

- رجیسترهای قابل دید توسط کاربر چند نوع مختلف دارد.
  - رجیسترهای همه منظوره: می توان در آن همه چیز (آدرس، داده) نگهداری کرد.
  - رجیسترهای داده: فقط می توان در آن داده نگهداری کرد.
  - رجیستر آدرس: فقط آدرس مکانی از حافظه در آن نگهداری می شود.
  - رجیسترهای وضعیت: مثال رجیستر flag.
- معمولاً ۸ تا ۳۲ رجیستر همه منظوره در CPU داریم. هر چه کمتر باشند دسترسی به حافظه بیشتر می شود. و تعداد بیش از حد آن هم قابل ساخت نیست به دلیل گرفتن فضای CPU و همچنین بی تاثیر بودن بر کاهش دسترسی به حافظه.
- رجیسترهای وضعیت هم عموماً به صورت بیتی قابل خواندن هستند و معمولاً با دستورها قابل نوشتن نیستند. و همچنین در دستورهای پرش شرطی به طور ضمنی به کار می روند (یعنی پرش مشروط بر آنها انجام می شود).

کمی بیشتر درباره رجیسترهای غیر قابل دید توسط کاربر

- معروف ترین این رجیسترها عبارتند از:
  - PC: شمارنده برنامه. که با آن آشنا هستیم.
  - IR: رجیستری که دستور در حال اجرا در آن قرار دارد و تفسیر از روی این رجیستر انجام می شود.
  - MAR (Memory Address Register): پس از محاسبه آدرس موثر، مقدار آن درون این رجیستر قرار می گیرد. مقدار این رجیستر به باس آدرس سیستم متصل است.
  - MBR(Memory Buffer Register): هر داده ای که روی باس داده سیستم ردوبدل می شود ابتدا در این رجیستر نگهداری می شود. مقدار این رجیستر به باس داده سیستم متصل است.

مثال
ساختار رجیسترها در معماری 8086

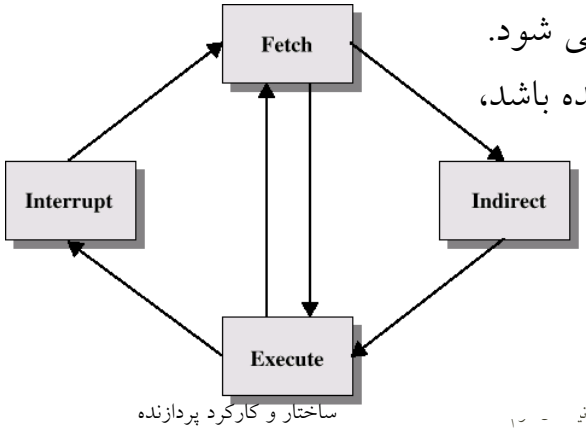


(b) 8086

- همه رجیسترها ۱۶ بیتی هستند.
- چهار رجیستر اول همه منظوره هستند.
- البته کاربرد های دیگری هم می توانند داشته باشند.
- به عنوان مثال BX در آدرس دهی جابجایی کاربرد دارد.
- SP: نشانگر stack که با آن آشنا هستیم.
- SI و DI برای آدرس دهی رجیسترغیر مستقیم و جابجایی هستند.
- رجیستر های segment برای انتخاب بین بخش های حافظه هستند.
- Instr Ptr: همان PC است.
- Flags: با این رجیستر آشنا هستیم.

در اینجا روند انجام چرخه دستور را در CPU بررسی می کنیم.

- همانطور که گفتیم برای اجرای هر دستور سه مرحله انجام می شود:
  - Fetch، Execute، بررسی وقفه (interrupt).
- با توجه به مودهای آدرس دهی که دیدیم شاید نیاز باشد تا چندین دسترسی دیگر به حافظه انجام شود تا Operand مورد نظر به دست آید.
- به این مرحله (یا مراحل) اضافی می گوییم: indirect.
- پس از fetch اگر نیاز باشد، indirect می شود.
- پس از execute اگر درخواست وقفه آمده باشد، interrupt انجام می شود.
- حالا می خواهیم اجرای هر مرحله را با دقت بیشتری دنبال کنیم.

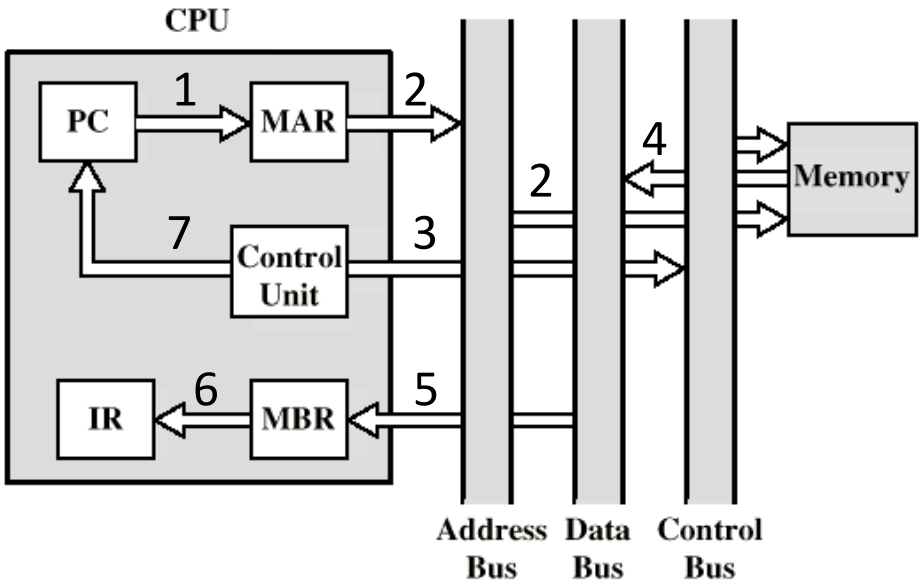


۱- برداشتن دستور از حافظه (FETCH)

- به طور معمول در CPU برای انجام Fetch مراحل زیر انجام می شود.
  - آدرس دستور بعدی در PC قرار دارد.
  1. این مقدار به MAR انتقال می یابد.
  2. مقدار MAR روی باس آدرس قرار می گیرد.
  3. واحد کنترل از حافظه درخواست خواندن می کند.
  4. حافظه مقدار مورد نظر (دستور) را روی باس داده قرار می دهد.
  5. این مقدار به رجیستر MBR انتقال می یابد.
  6. سپس مقدار MBR به IR انتقال می یابد.
  7. مقدار PC یکی افزایش می یابد.
  8. مقدار IR بررسی می شود. (قسمت Opcode بررسی می شود.)
  9. اگر لازم باشد مرحله indirect انجام می شود. و اگرنه به مرحله execute می رویم.

۱- برداشتن دستور از حافظه (FETCH) (ادامه)

در شکل زیر روند جابه جایی داده برای انجام FETCH دید می شود.



MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

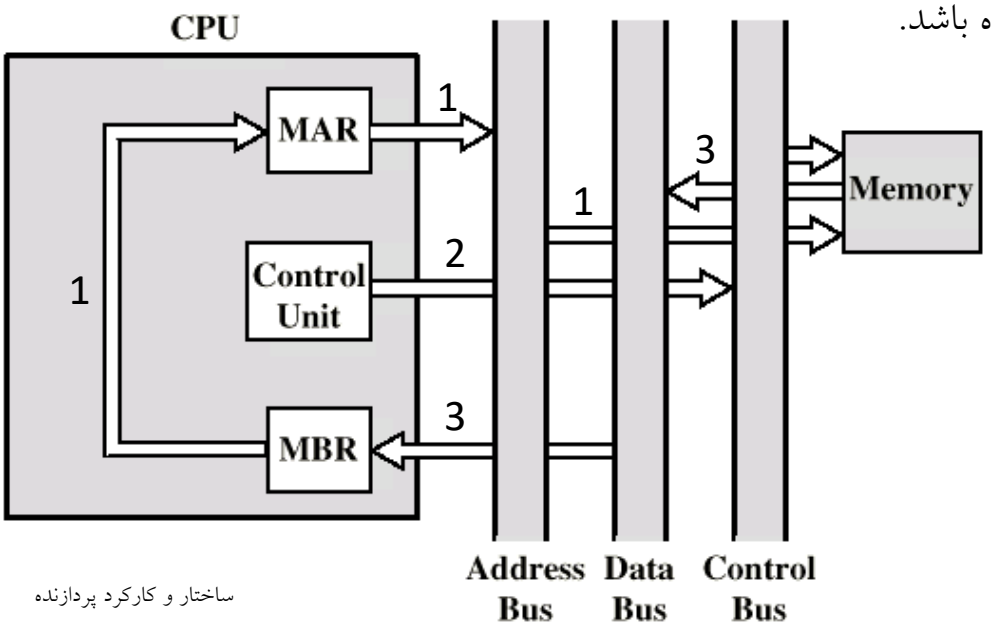
۲- مرحله INDIRECT

- در این مرحله بخش Operand از دستور (داخل IR) برداشته می شود.
- با توجه به مود آدرس دهی (که از Opcode مشخص می شود.) باید آدرس موثر محاسبه شود. (اگر operand در رجیستر باشد یا صریح باشد نیازی به مرحله indirect نیست.)
- برای انجام Indirect مراحل زیر انجام می شود:
  1. آدرس موثر محاسبه شده به MAR انتقال پیدا می کند. و روی باس آدرس می رود.
     - دقت کنید که در مودهای آدرس دهی مختلف MAR به روش های مختلفی محاسبه می شود.
  2. واحد کنترل درخواست خواندن را به حافظه می فرستد.
  3. نتیجه خوانده شده از حافظه به MBR انتقال می یابد.
- دقت کنید که در بعضی از روش های آدرس دهی (مثل روش Indirect) این مراحل باید دو بار انجام شود. چون باید ابتدا باید آدرس موثر از حافظه برداشته شود.
- اگر دستوری دو Operand داشته باشد، و هر دو نیاز به انجام indirect داشته باشند این مراحل باید برای هر Operand تکرار شود.

۲- مرحله INDIRECT

در شکل زیر مراحل اجرای یکبار INDIRECT دیده می شود.

- در اینجا فرض بر این بوده که آدرس موثر از حافظه دریافت شده است. و اکنون در MBR ذخیره شده است. این آدرس موثر ممکن است به روش دیگری به دست آمده باشد.



۳- مرحله اجرا (EXECUTE)

در این مرحله با توجه به نوع دستور ممکن است کارهای مختلفی انجام شود.

- مرحله Execute برای هر دستور متفاوت است.
- واحد کنترل پس از بررسی دستور که در IR ذخیره شده تصمیم می گیرد که برای اجرای این دستور باید چه کارهایی انجام شود.
- مرحله دستور می تواند شامل اجرای یک یا چند مورد از موارد زیر باشد.
  - خواندن یا نوشتن حافظه.
  - خواندن یا نوشتن از ماجول I/O.
  - انتقال مقدار از یا به رجیسترها.
  - فرمان انجام عملیات در ALU.
- با توجه به هر دستور باید نمودار متفاوتی برای انتقال داده ها رسم شود.
- مثال: در دستور ADD R1,(R2) چه مراحلی انجام می شود.
  - پس از یکبار indirect، محتوای یکی از منبع ها در MBR ذخیره شده.
  - به ALU دستور جمع داده می شود. ورودی آن هم دو رجیستر MBR و R1 انتخاب می شود. خروجی ALU رجیستر R1 قرار داده می شود.

۴- مرحله بررسی وقفه (INTERRUPT)

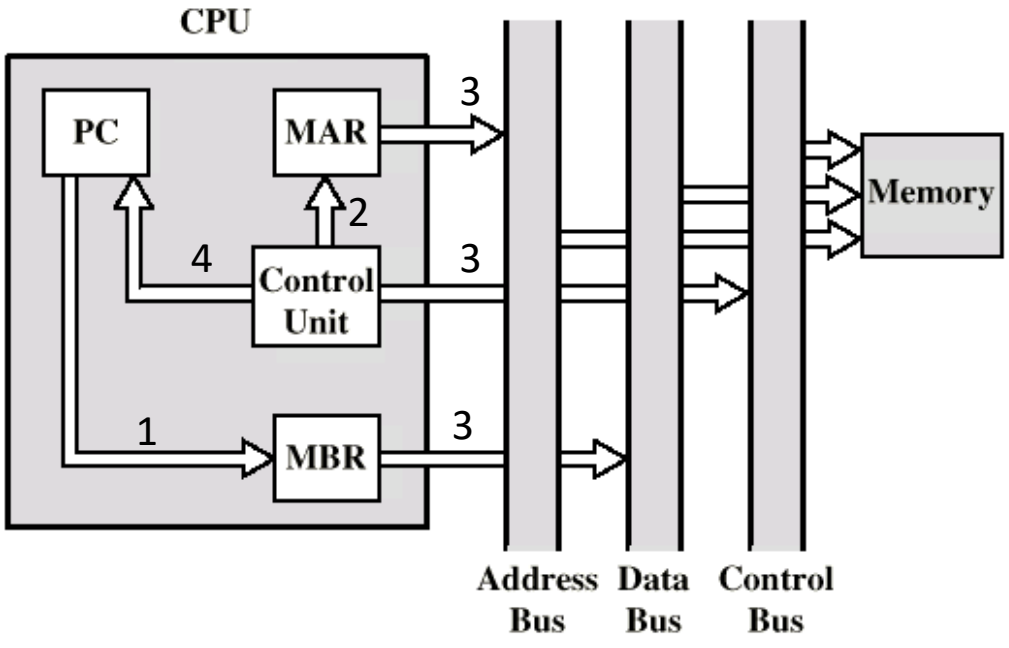
پس از پایان اجرای هر دستور درخواست های وقفه بررسی می شود.

- اگر درخواست وقفه پذیرفته شود باید مرحله ای به نام Interrupt اجرا شود.
- در این مرحله به طور عمومی مراحل زیر انجام می شود:

1. مقدار PC باید ذخیره شود. بنابرین مقدار PC به MBR انتقال می یابد.
   - برای اینکه بدانیم اجرای برنامه باید از کجا ادامه یابد، PC ذخیره می شود.
2. آدرس مکانی که می خواهیم مقدار PC را در آن ذخیره کنیم به MAR فرستاده می شود.
   - معمولاً می خواهیم این مقدار را در Stack ذخیره کنیم. پس مقدار SP به MAR فرستاده می شود.
   - دقت کنید که SP آدرس مکان خالی stack را نشان می دهد.
   - سپس مقدار SP یکی کاهش (یا افزایش) می یابد.
3. مقدار MBR به حافظه انتقال می یابد.
4. با توجه به اینکه کدام وقفه درخواست شده است، آدرس ابتدای زیربرنامه رسیدگی وقفه مورد نظر به PC انتقال می یابد.
   - روش تعیین اینکه کدام دستگاه درخواست وقفه داده است در اسلاید ۱۲۲مطرح شده است.
   - برای هر وقفه زیربرنامه ای برای رسیدگی نوشته شده است و آدرس ابتدای آن مشخص است.

۴- مرحله بررسی وقفه (INTERRUPT)

در شکل زیر روند انجام شده برای اجرای این مرحله دیده می شود.

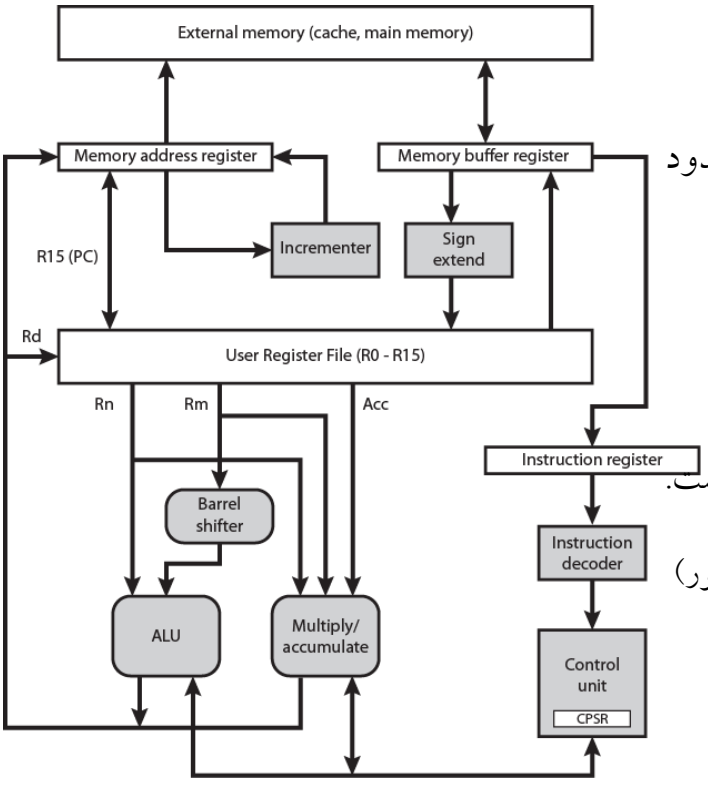


مثالی از ساختار درونی یک CPU واقعی: معماری ARM

- در اینجا برای آشنایی خصوصیات یک خانواده معماری یعنی ARM را به طور خیلی خلاصه معرفی می کنیم.
- از نوع معماری RISC (موضوع بخش بعدی) است.
- تعداد متوسطی رجیستر دارد.
- دسترسی به حافظه فقط با دستورهای load و store انجام می شود. (اسلاید ۱۸۰)
- طول دستورها ثابت است. ولی دستورها شکل های متفاوتی دارند. (اسلاید ۱۸۴)
- تعداد مودهای آدرس دهی محدود است. (اسلاید ۱۸۰)
  - مودهای indirect یا جابجایی برای مکان های حافظه وجود ندارد.
- ابزاری برای افزایش یا کاهش خودکار آدرس ها قرار داده شده است.
- برای همه دستورها قابلیت شرطی شدن گذاشته شده است. بنابرین فقط پرش نیست که می تواند به طور شرطی انجام شود، همه دستورها می تواند برمبنای درست بودن یک شرط انجام شود. اگر شرط برقرار نباشد دستور انجام نمی شود.

مثالی از ساختار درونی یک CPU واقعی: معماری ARM

ساختار ساده شده CPU



- در ARM اتصال بین بخش ها محدود شده است.
- بنابرین ساختار باس داخلی نداریم.
- بخش های آشنا
  - MBR و MAR و اتصال آنها به باس سیستم.
  - ALU: که ورودی آن دو رجیستر است. و خروجی آن هم رجیستر است.
  - رجیستر IR و واحد ID (مفسر دستور)
  - رجیستر PC (اینجا R15 است).
  - مجموعه رجیسترها (۱۶ رجیستر)
  - واحد کنترل.
  - مدار ضرب کننده. و مدار شیفت.

ساختار رجیسترها در معماری ARM

- در مجموع ۳۷ رجیستر وجود دارد که هر کدام ۳۲ بیتی است.
- ۳۱ رجیستر همه منظوره است. که بعضی وظیفه خاصی هم دارند مانند PC.
- ۶ رجیستر، رجیسترهای وضعیت و کنترل هستند.
  - بعضی از رجیسترهای وضعیت قابل دید توسط کاربر است.
- ۱۶ رجیستر از R0 تا R15 اسم گذاری شده است و قابل دید توسط کاربر است.
- همه عملیات ریاضی و چرخش و منطقی باید روی این رجیسترها انجام شود.
- معماری دارای چندین مود اجرا است.
  - در مودهای مختلف وظیفه بعضی از رجیسترها عوض می شود.
- در اسلاید بعدی فهرست رجیسترها در مودهای مختلف دیده می شود.
- مود مورد نظر ما User است. (بقیه برای مطالعه است.)

ساختار رجیسترها در ARM (ادامه)

- در جدول مقابل رجیسترهای قابل دید کاربر دیده می شود.
- رجیسترهای آشنا:
  - PC (R15)
  - SP (R13)
  - CPSR (رجیستر وضعیت) یا رجیستر flag
- در مودهای مختلف کارکرد بعضی از رجیسترها تغییر می کند.

Modes						
	Privileged modes					
		Exception modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13(SP)	R13(SP)	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14(LR)	R14(LR)	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Shading indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode.

SP = stack pointer
LR = link register
PC = program counter
CPSR = current program status register
SPSR = saved program status register

فهرست وقفه ها در ARM (جدول برای مطالعه)

Exception type	Mode	Normal entry address
Reset	Supervisor	0x00000000
Data abort	Abort	0x00000010
FIQ (fast interrupt)	FIQ	0x0000001C
IRQ (interrupt)	IRQ	0x00000018
Prefetch abort	Abort	0x0000000C
Undefined instructions	Undefined	0x00000004
Software interrupt	Supervisor	0x00000008

- در هر CPU انواع وقفه ها در ابتدا تعریف می شود.
- و برای هر وقفه آدرسی در حافظه به عنوان آدرس ابتدای زیربرنامه رسیدگی به وقفه تعیین می شود.
- آغاز زیر برنامه وقفه باید در آن آدرس نوشته شود.
- زمانی که وقفه پذیرفته شد. این آدرس داخل PC قرار می گیرد (اسلاید ۱۹۹)
- در جدول مقابل انواع وقفه های ARM و آدرس آنها دیده می شود.
- به این جدول آدرس می گویند: بردار وقفه.

RISC (Reduced Instruction Set Computers) در برابر CISC (Complex Instruction Set Computers)

- می توان دو روند قالب پیشنهادی در غالب RISC و CISC بررسی کرد.
- که با عنوان های مجموعه دستورهای پیچده یا کاهش یافته شناخته می شوند.
- طراحی کامپیوترها ابتدا در روندی به سوی CISC انجام شد. سپس گروه های دیگری طراحی RISC را پیشنهاد کردند. امروزه هر دو گروه از خصوصیت های یکدیگر بهره می گیرند.
- اما خصوصیت کامپیوترهای RISC در برابر CISC چیست؟
  - افزایش تعداد رجیسترهای همه منظوره (یا بهینه سازی به کارگیری رجیسترها در کامپایلر.)
  - مجموعه دستورهای محدود و ساده.
  - مودهای آدرس دهی ساده.
  - اندازه دستورهای معمولاً ثابت.

چه چیزی باعث شد که ابتدا طراحان به سمت طراحی CISC بروند؟

- ابتدا تصور بر این بود که هر چه زبان CPU (زبان ماشین) به زبان سطح بالا نزدیک تر باشد بهتر است.
- زیرا زبان های سطح بالا (مانند C، Basicو ....) هر روز در حال افزایش توانایی بودند. و امکانات زیادی به آنها اضافه می شد.
- بنابرین سعی کردند تا امکانات سخت افزار (CPU) را بالا ببرند.
  - مجموعه دستورهای بزرگتر. (و دستور های پیچیده تر) که منجر به ساخت مدارهای پیچیده تر داخل CPU می شود.
  - مودهای آدرس دهی متعدد و پیچیده (مثال در اسلاید ۱۷۹ و ۱۸۰) که بازهم مدارهای بیشتری را در CPU ایجاد می کرد.
  - حتی بعضی از دستورهای زبان های سطح بالا مستقیماً با مدار در CPU طراحی می شد.
- این باعث می شد که نوشتن برنامه مترجم (compiler) ساده تر شود.

چه چیزی باعث شد که طراحان متوجه شوند طراحی CISC چندان هم بهینه نیست؟

- در کامپیوتر های CISC مجموعه بزرگی از دستورها ساخته شد.
- برای انجام این کار CPU خیلی از نظر مداری بزرگ شد.
- اما در بررسی های انجام شده دیده شد که در اجرای برنامه سطح بالا تعداد استفاده از یک سری از دستورها خیلی بالا است و خیلی از دستورها به ندرت استفاده می شوند. جدول زیر نمونه از این بررسی است.
- ۴۵٪ درصد دستورها انتقال داده است.
- ۲۹٪ درصد دستورها تصمیم گیری است.
- فقط ۶٪ دستورهای دیگر استفاده شده.
- پس دستورهای پیچیده به ندرت استفاده می شوند.

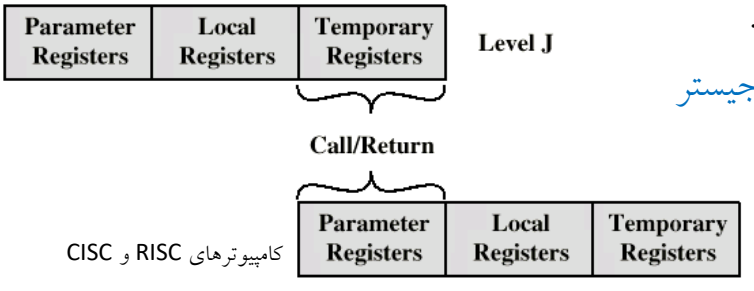
	Dynamic Occurrence	
	Pascal	C
ASSIGN	45%	38%
LOOP	5%	3%
CALL	15%	12%
IF	29%	43%
GOTO	—	3%
OTHER	6%	1%

چه چیزی باعث شد که طراحان متوجه شوند طراحی CISC چندان هم بهینه نیست؟ (ادامه)

- بنابرین در RISC تصمیم بر این شد که تعداد دستور ها کاهش یابد. (دستورهای کم کاربرد حذف شدند.)
- همچنین مودهای آدرس دهی پیچیده که کم کاربرد بودند حذف شدند.
- از سوی دیگر بررسی شد که بیشتر زمان CPU روی پرش به زیر برنامه ها و بازگشت از آنها گرفته می شود. چون باید تعداد زیادی متغیر ذخیره شود و بازیابی شود.
- در کامپیوترهای RISC از فضای به دست آمده حاصل از کاهش دستورها و مودهای آدرس دهی استفاده شد و تعداد رجیسترها افزایش یافت.
- به این ترتیب می توان متغیرهای بیشتری را در رجیسترها ذخیره کرد بدون اینکه نیاز به ذخیره کردن آنها در حافظه باشد.

ساختار رجیسترها در کامپیوترهای RISC

- در این کامپیوترها سعی می شود تا متغیرهای محلی یک برنامه همه در رجیستر ذخیره شوند. که باعث کاهش دسترسی به حافظه خواهد شد.
- اگر زیربرنامه ای فراخوانی شود: (مشابه توابع در زبان C)
  - همه متغیرهای محلی باید ذخیره شوند.
  - متغیرهایی باید برای زیر برنامه فرستاده شود.
  - متغیرهایی به عنوان نتیجه باید از زیربرنامه برگردانده شود.
- همه این موارد اگر بخواهد با دسترسی به حافظه انجام شود اجرای برنامه بسیار کند می شود. برای همین رفت و برگشت از زیربرنامه ها زمانگیر ترین روندها است.
- پیشنهاد شد که در کامپیوترهای RISC برای انتقال متغیرها بین زیر برنامه ها از ساختار زیر استفاده شود.
- نام این ساختار: پنجره رجیستر

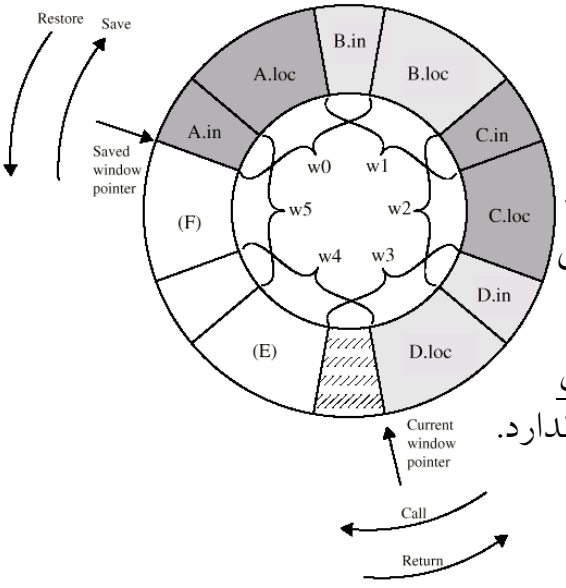


پنجره رجیستر یکی از خصوصیات کامپیوترهای RISC است.
پنجره رجیستر چیست و چگونه کار می کند؟

- ایده پنجره رجیستر:
  1. برای هر برنامه تعدادی محدود رجیستر محلی در نظر بگیریم. (مجموعه محلی)
  2. چندین مجموعه محلی از ابتدا آماده کنیم.
  3. هر بار که به زیر برنامه ی جدیدی می رویم این مجموعه را تغییر دهیم.
  4. اگر از زیر برنامه برگشتیم به همان مجموعه قبلی برمی گردیم. (متغیرهای محلی در آن ذخیره شده باقیمانده است.)
- این مجموعه رجیستر محلی به سه بخش تقسیم شده است.
  - رجیسترهای پارامتر، رجیسترهای محلی، رجیستر های موقت.
- رجیسترهای موقت یک مجموعه با رجیسترهای پارامتر مجموعه بعدی یکسان است.
- بنابرین می توان بدون اینکه داده ها را جابه جا کرد، متغیر را ها را بین زیربرنامه ها جابه جا کرد.

شکل زیر روند تعویض بین مجموعه رجیسترها را هنگام فراخوانی و بازگشت از زیر برنامه ها در روش پنجره رجیستر نشان می دهد.

- اگر بخواهیم متغیری را به زیر برنامه ای بفرستیم آن را در بخش موقت ذخیره می کنیم. و برای فرستادن نتیجه به برنامه قبلی آن را در بخش پارامترها ذخیره می کنیم.
- پنجره w0 مربوط به برنامه اصلی است.
- وقتی به اولین زیر برنامه رفتیم به پنجره w1 تغییر وضعیت می دهیم.
- در هنگام بازگشت پنجره به عقب برمی گردد.
- همین روند برای فراخوانی ها و بازگشت های بعدی نیز به کار می رود.
- دقت کنید که برنامه نویس فقط به پنجره فعلی دسترسی دارد و حق تغییر بقیه رجیسترها را ندارد.
- اگر همه پنجره ها پر شد قدیمی ترین آنها در حافظه ذخیره می کنیم.



مقایسه نهایی بین CISC و RISC

هر دو معماری در حال حاضر وجود دارند. پس برتری هر کدام چیست؟

- برنامه نویسی ساده تر می شود. (نوشتن برنامه کامپایلر منظور است.)
  - البته نشان داده شده که برنامه ریزی با دستورهای پیچیده تر الزاماً ساده تر نیست.
- تعداد دستورهای برنامه کمتر می شود و فضای کمتری در حافظه می گیرد.
  - البته هر دستور پیچیده تر است. پس فضای بیشتری خواهد گرفت.
  - چون Opcode و قسمت operand بزرگی دارند.
- سخت تر بودن در برنامه نویسی (نوشتن برنامه کامپایلر منظور است.)
  - نوشتن برنامه مسلماً سخت تر خواهد شد چون مجبور به استفاده از دستورهای ساده تر هستیم.
- برنامه از نظر تعداد دستورها بیشتر می شود.
  - حافظه ارزان شده پس مشکلی نیست.
  - دستورها کوچکترند پس جبران می شود.
  - چون Opcode کوچک شده و چون مودهای آدرس دهی ساده شده اند.، Operand ها هم کوچک شده اند.

مقایسه نهایی بین CISC و RISC (ادامه)

- به نظر می رسد باید سرعت اجرای برنامه سریعتر باشد. چون تعداد دستورها برای اجرای یک برنامه خاص کمتر است. و تعداد دسترسی به حافظه کمتر می شود.
- اما چون دستورها پیچیده شده، اندازه دستورها بزرگ می شود، زمان خواندن هر دستور طولانی می شود. زمان اجرای آن هم طولانی می شود. چون بعضی از دستورها باید در چند مرحله انجام شود. مودهای آدرس دهی پیچیده هم که اجرا را کند می کند.
- حتی اجرای دستورهای ساده هم به خاطر opcode بزرگ کند می شود.
- به نظر می رسد باید سرعت اجرای برنامه کندتر باشد. چون تعداد دستورها برای اجرای یک برنامه خاص بیشتر است. و تعداد دسترسی به حافظه بیشتر می شود.
- اما چون دستورها ساده و کوچک شده اند. برداشتن آنها از حافظه زمان زیادی نمی گیرد. و دسترسی اضافه هم که برای مودهای آدرس دهی اضافی نداریم. و انجام هر دستور سریع می شود.
- در مجموع تعداد دستورها بیشتر شده اند اما اجرای هر دستور سریعتر شده است.

مثال: یک کامپیوتر با معماری RISC (جزییات برای مطالعه)

معماری SPARC (ساختار رجیسترها و پنجره رجیسترها)

تعداد بالای رجیسترها و ساختار پنجره ای آنها مشخصه کامپیوترهای RISC است.

- ۸ رجیستر عمومی هم داریم.

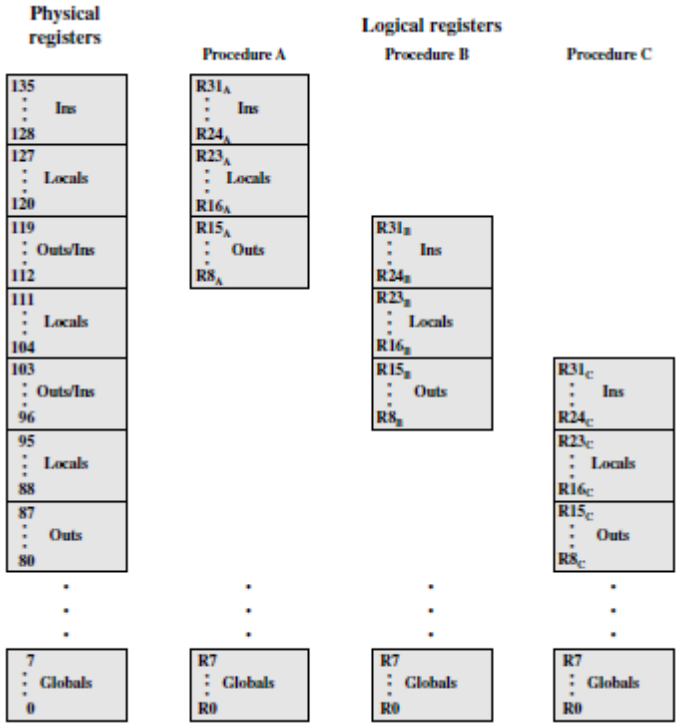
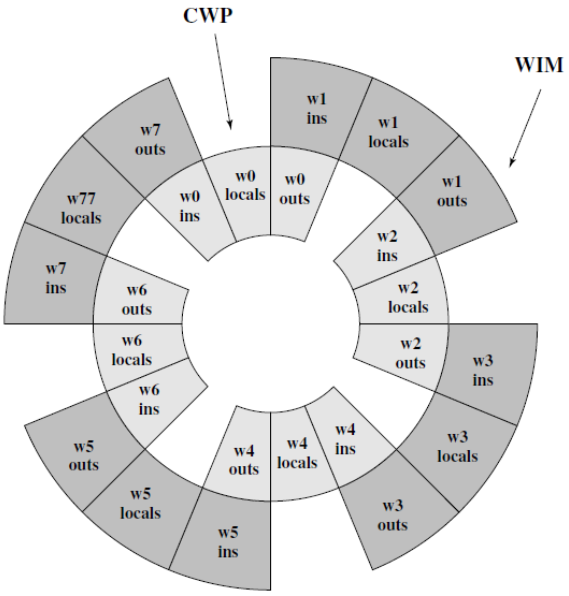


Figure 13.12 SPARC Register Window Layout with Three Procedures



معماری SPARC (ادامه)

مجموعه دستورها (جزییات برای مطالعه)

OP	Description	OP	Description
Load/Store Instructions		**Arithmetic Instructions**	
LDSB	Load signed byte	ADD	Add
LDSH	Load signed halfword	ADDCC	Add, set icc
LDUB	Load unsigned byte	ADDX	Add with carry
LDUH	Load unsigned halfword	ADDXCC	Add with carry, set icc
LD	Load word	SUB	Subtract
LDD	Load doubleword	SUBCC	Subtract, set icc
STB	Store byte	SUBX	Subtract with carry
STH	Store halfword	SUBXCC	Subtract with carry, set icc
STD	Store word	MULSCC	Multiply step, set icc
STDD	Store doubleword	**Jump/Branch Instructions**	
Shift Instructions		BCC	Branch on condition
SLL	Shift left logical	FBCC	Branch on floating-point condition
SRL	Shift right logical	CBCC	Branch on coprocessor condition
SRA	Shift right arithmetic	CALL	Call procedure
Boolean Instructions		JMPL	Jump and link
AND	AND	TCC	Trap on condition
ANDCC	AND, set icc	SAVE	Advance register window
ANDN	NAND	RESTORE	Move windows backward
ANDNCC	NAND, set icc	RETT	Return from trap
OR	OR	**Miscellaneous Instructions**	
ORCC	OR, set icc	SETHI	Set high 22 bits
ORN	NOR	UNIMP	Unimplemented instruction (trap)
ORNCC	NOR, set icc	RD	Read a special register
XOR	XOR	WR	Write a special register
XORCC	XOR, set icc	IFLUSH	Instruction cache flush
XNOR	Exclusive NOR		
XNORCC	Exclusive NOR, set icc		

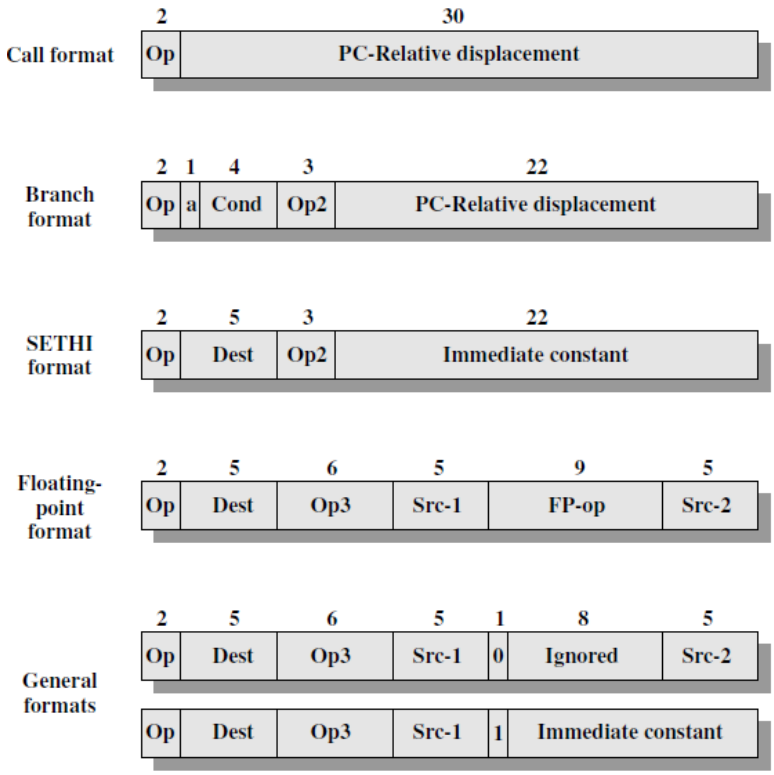
- تعداد کم و ساده بودن دستورها از مشخصه های کامپیوترهای RISC است.

معماری SPARC (ادامه)
مودهای آدرس دهی

Instruction Type	Addressing Mode	Algorithm	SPARC Equivalent
Register-to-register	Immediate	operand = A	S2
Load, store	Direct	EA = A	R_0 + S2
Register-to-register	Register	EA = R	R_{S1}, R_{S2}
Load, store	Register Indirect	EA = (R)	R_{S1} + 0
Load, store	Displacement	EA = (R) + A	R_{S1} + S2

- همانطور که در جدول بالا دیده می شود. فقط برای دستور های Load و store مودهایی برای دسترسی به حافظه قرار داده شده است. که شامل مودهای مستقیم، رجیستر غیرمستقیم و جابه جایی می شود.
- برای سایر دستورها اپرندها فقط می تواند رجیستر باشد یا صریح باشد.
- و مود آدرس دهی غیرمستقیم هم نداریم.
- اینها همه (ساده بودن مودهای آدرس دهی) خصوصیات کامپیوترهای RISC است.

معماری SPARC (ادامه) (جزییات برای مطالعه)
شکل دستورها



- اندازه برای همه انواع دستورها ثابت است.
- که این خصوصیت RISC است.
- دستورها به گروه های مختلف طبقه بندی شده است و هر یک شکل مربوط به خود را دارد.
- هم اندازه بودن دستورها، fetch و تفسیر آنها را ساده می کند. در نتیجه اجرای هر دستور سریعتر می شود.

واحد کنترل در CPU

شامل: کارکرد واحد کنترل، کنترل میکروپروگرام

بخش چهارم

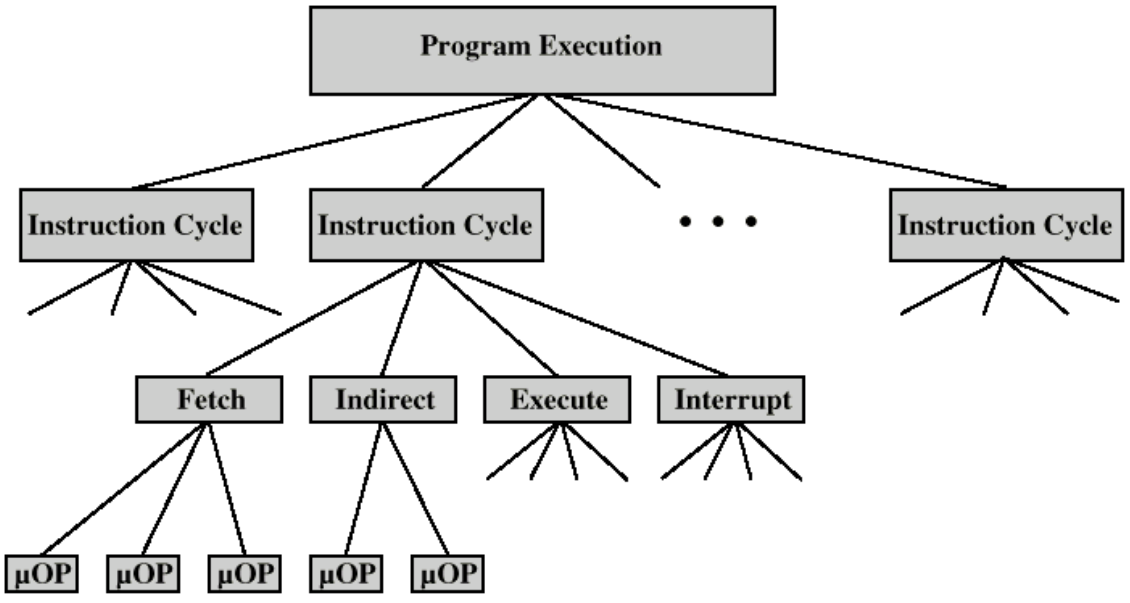
واحد کنترل چه کاری در CPU انجام می دهد؟
خلاصه پاسخ: باعث می شود که دستورها به ترتیب اجرا شوند.

- وظیفه کامپیوتر: اجرای برنامه کاربر.
- برنامه: مجموعه ی دستورهایی که به ترتیب مناسب اجرا شوند.
- دستور: وظیفه ای که CPU باید در زمان فعلی باید انجام دهد. دستور از چند سیکل (چرخه) تشکیل شده است.
- سیکل (چرخه) دستور: شامل fetch، execute، indirect، و interrupt.
- هر سیکل (چرخه) شامل اجرای چند گام است.
- به این گام ها اصطلاحاً می گویند: Micro-operation (ریز-عملکرد)
- در هر Micro-operation کارهایی کم و بسیار ساده انجام می شود: (یک یا چند کار زیر)
  - انتقال مقدار بین دو رجیستر.
  - انتقال مقدار بین یک رجیستر و باس سیستم.
  - عملکرد ALU

واحد کنترل چه کاری در CPU انجام می دهد؟ (ادامه)
خلاصه پاسخ: باعث می شود که دستورها به ترتیب اجرا شوند.

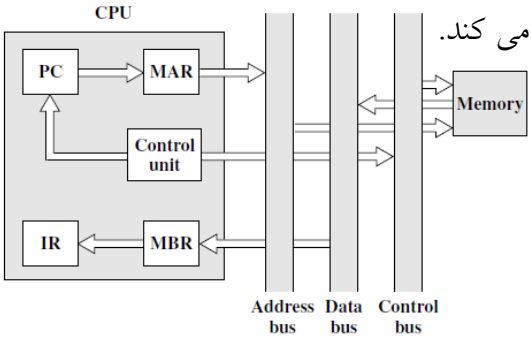
- واحد کنترل چه کار می کند؟
  1. اجرای برنامه را از طریق اجرای پشت سرهم سیکل دستورها انجام می دهد.
  2. کاری می کند تا به ازای هر سیکل دستور، Micro-operation های مناسب به ترتیب اجرا شوند.
  3. در هر Micro-operation، فرمان های مناسب را برای بخش های مختلف CPU و باس کنترل می فرستد تا آن Micro-operation به درستی اجرا شود.
- فرمان های ارسالی ورودی و خروجی های رجیسترها را به مکان های مناسب متصل می کنند. یا فرمان مناسب را به ALU می دهند.
- یک روش طراحی واحد کنترل استفاده از مدارهای ترتیبی است.
- روش دیگر روش میکرو پروگرام (microprogram) است که موضوع بخش بعدی است.
- برای صرفه جویی بعد از این Micro-operation را می نویسیم: µ-OP

اجرای هر سیکل دستور با اجرای چند µ-OP انجام می شود. (به دو اسلاید قبل مراجعه کنید.)



با اولین سیکل اجرای دستور (یعنی FETCH) شروع می کنیم. چه µ-op هایی برای اجرای این سیکل نیاز است؟

- برای اجرای fetch چهار رجیستر درگیر می شوند. (یادآوری)
  - MAR: به باس آدرس متصل است. آدرسی از حافظه که از آن باید خوانده/ نوشته شود در آن ذخیره می شود.
  - MBR: به باس داده متصل است. داده ای که خوانده شده یا باید نوشته شود را نگهداری می کند.
  - PC: آدرس دستوری را که باید برداشته شود را نگهداری می کند.
  - IR: آخرین دستوری را که برداشته شده نگهداری می کند.
- برای یادآوری مراحل fetch به اسلاید ۱۹۴ مراجعه کنید.



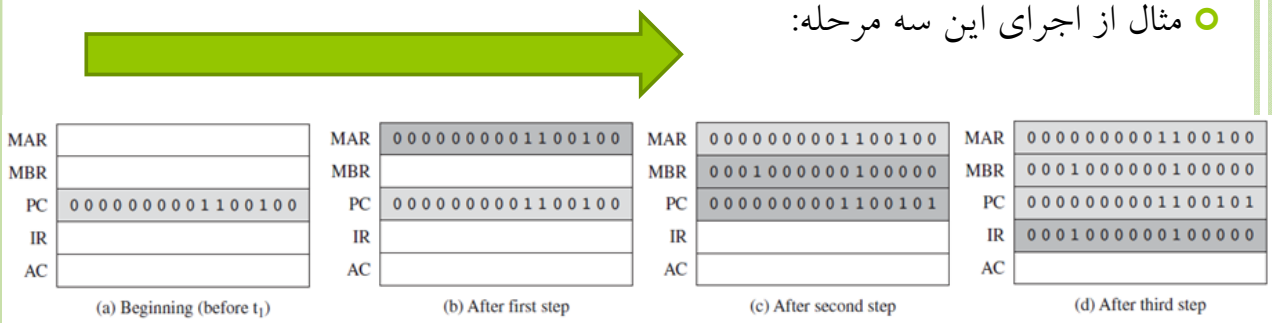
MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

مراحل اجرای FETCH را منظم و با ذکر ترتیب اجرا می نویسیم.

- سه مرحله لازم برای انجام fetch
  1. انتقال PC به MAR
  2. خواندن از حافظه و افزایش PC
  3. انتقال MBR به IR

t_1: MAR ← (PC)
t_2: MBR ← Memory
PC ← (PC) + I
t_3: IR ← (MBR)

- نکته: دقت کنید در این بخش استفاده از پرانتز به معنای indirect نیست.
- و همه انتقال ها بین رجیسترها انجام می شود. مگر اینکه لفظ حافظه نوشته شود.
- مثال از اجرای این سه مرحله:



```
t1: MAR ← (PC)
t2: MBR ← Memory
    PC ← (PC) + I
t3: IR ← (MBR)
```

نکته هایی در سه مرحله لازم برای انجام FETCH

- در زمان t2 دو μ-Opانجام داده ایم. چون این دو با یکدیگر تداخلی نداشته اند.
- هرچه بتوانیم در زمان کمتری μ-Opها را انجام بدهیم اجرای fetch سریعتر خواهد شد. پس آیا نمی شد همه را در یک زمان انجام داد؟ پاسخ: خیر زیرا قوانینی برای اجرای همزمان μ-OPها وجود دارد.
- قوانین لازم برای اجرای همزمان μ-Opها:
  - ترتیب مناسب باید رعایت شود. مثلاً نمی توان قبل از آنکه آدرس حافظه روی باس قرار گرفته باشد آن را خواند. پس نمی توان دو μ-Op اول را با هم انجام داد. یا نمی توان پیش از آنکه محتوای حافظه به به MBR انتقال یافته باشد آن را به IR انتقال داد پس های زمان 2 و 3 را نیز نمی توان در یک زمان انجام داد.
  - نباید رجیستری را همزمان خواند و نوشت. به عنوان مثال نباید عملیات روی PC را در یک زمان انجام داد.

```
t1: MAR ← (PC)
t2: MBR ← Memory
t3: PC ← (PC) + I
    IR ← (MBR)
```

- با رعایت این دو قانون fetch مقابل نیز درست است:

μ-Op های لازم برای انجام چرخه INDIRECT

- همان طور که پیش از این گفته شد، اگر دستوری دارای مود Indirect باشد، قبل از انجام مرحله execute، یک مرحله با نام Indirect انجام می شود.
- این چرخه در سه μ-Op انجام می شود:

```
t1: MAR ← (IR(Address))   بخش آدرس دستور
t2: MBR ← Memory
t3: IR(Address) ← (MBR(Address))
```

1. بخش آدرس دستور (که همان بخش operand است.) در MAR قرار می گیرد.
2. محتوای حافظه به MBR منتقل می شود.
3. بخش آدرس (یعنی آدرس operand) به بخش آدرس IR انتقال داده می شود.

- در نتیجه: پس از انجام این چرخه آدرس واقعی operand در IR قرار می گیرد.
- اگر دو Operand دارای مود Indirect باشند، این چرخه باید دو بار انجام شود.

µ-Op های لازم برای انجام چرخه INTERRUPT

- پیش از بررسی چرخه execute، چرخه Interrupt را بررسی می کنیم.
- در پایان اجرای هر دستور در صورت لزوم چرخه Interrupt انجام می شود.
- µ-Op های معمول برای انجام این چرخه عبارت است از:

t_1: MBR ← (PC)

t_2: MAR ← Save_Address ⟵ آدرس ذخیره (معمولاً برابر با SP)

PC ← Routine_Address ⟵ آدرس زیربرنامه رسیدگی به وقفه

t_3: Memory ← (MBR)

- آدرس بازگشت که همان محتوای PC است معمولاً در stack ذخیره می شود.
- آدرس مکان خالی stack در SP قرار دارد.
- بعد از اینکه مشخص شد کدام دستگاه درخواست وقفه کرده است به آدرس ابتدای زیر برنامه پرش می کنیم. (یا به عبارت دیگر آدرس را در PC قرار می دهیم.) (این آدرس ها بردارهای وقفه نام دارند. به اسلاید ۲۰۵ مراجعه کنید.)

چرخه EXECUTE برای هر دستور متفاوت است.

این چرخه را برای چند دستور نمونه بررسی می کنیم. ۱- ADD

- دستور اولی که به عنوان مثال بررسی می کنیم: ADD R1,X.
- این دستور محتوای آدرس X حافظه را با R1 جمع می کند و نتیجه را در R1 ذخیره می کند.
- µ-Op های چرخه Execute برای این دستور:

t_1: MAR ← (IR(address)) بخش آدرس دستور

t_2: MBR ← Memory

t_3: R1 ← (R1) + (MBR)

- بخش آدرس دستور (operand دستور) در MAR قرار داده می شود.
- حافظه خوانده می شود.
- MBR با R1 جمع می شود. نتیجه در R1 قرار می گیرد.
- دقت کنید که هیچ یک از µ-Op ها را نمی توان همزمان اجرا کرد.

چرخه EXECUTE برای هر دستور متفاوت است. نمونه دوم: دستور ISZ

- دستوری در اغلب کامپیوترها وجود دارد به نام ISZ این دستور همزمان جمع و پرش شرطی را انجام می دهد.
- ISZ x ابتدا محتوای مکان x را یکی افزایش می دهد. اگر این مقدار صفر شده باشد از اجرای دستور بعدی صرف نظر می کند (از روی دستور بعدی می پرد). واگر نه دستور بعدی را اجرا می کند.
- µ-Op های چرخه Execute برای این دستور:

بخش آدرس دستور

```
t1: MAR ← (IR(address))
t2: MBR ← Memory
t3: MBR ← (MBR) + 1
t4: Memory ← (MBR)
    If ((MBR) = 0) then (PC ← (PC) + I)
```

- ابتدا مقدار حافظه مورد نظر در MBR خوانده می شود.
- سپس MBR یکی افزایش می یابد.
- سپس MBR در همان آدرس ذخیره می شود (مقدار MAR تغییر نکرده است.)
- همزمان اگر MBR صفر باشد، PC یکی افزایش می یابد. (پرش از دستور بعدی)

چرخه EXECUTE برای هر دستور متفاوت است.
نمونه سوم: دستور CALL

- دستور CALL x همانند پرش است. اجرای برنامه باید به آدرس صریح x انتقال یابد. علاوه بر آن آدرس بازگشت باید (معمولاً در stack) ذخیره شود.
- µ-Op های چرخه Execute برای این دستور:

```
t1: MAR ← (SP)
    MBR ← (PC)
t2: PC ← (IR(address))
    Memory ← (MBR)
```

1. ابتدا مقدار SP در MAR ذخیره می شود.
   همزمان مقدار PC در MBR ذخیره می شود. (این آدرس برگشت است.)
2. بخش آدرس دستور در PC ذخیره می شود. (برای پرش به آدرس x)

همزمان MBR (که در اینجا آدرس را برگشت را دارد.) در آدرس MAR (که مکان خالی stack را دارد.) ذخیره می شود.

- در اینجا توانسته ایم این چهار µ-Op را در دو زمان اجرا کنیم.

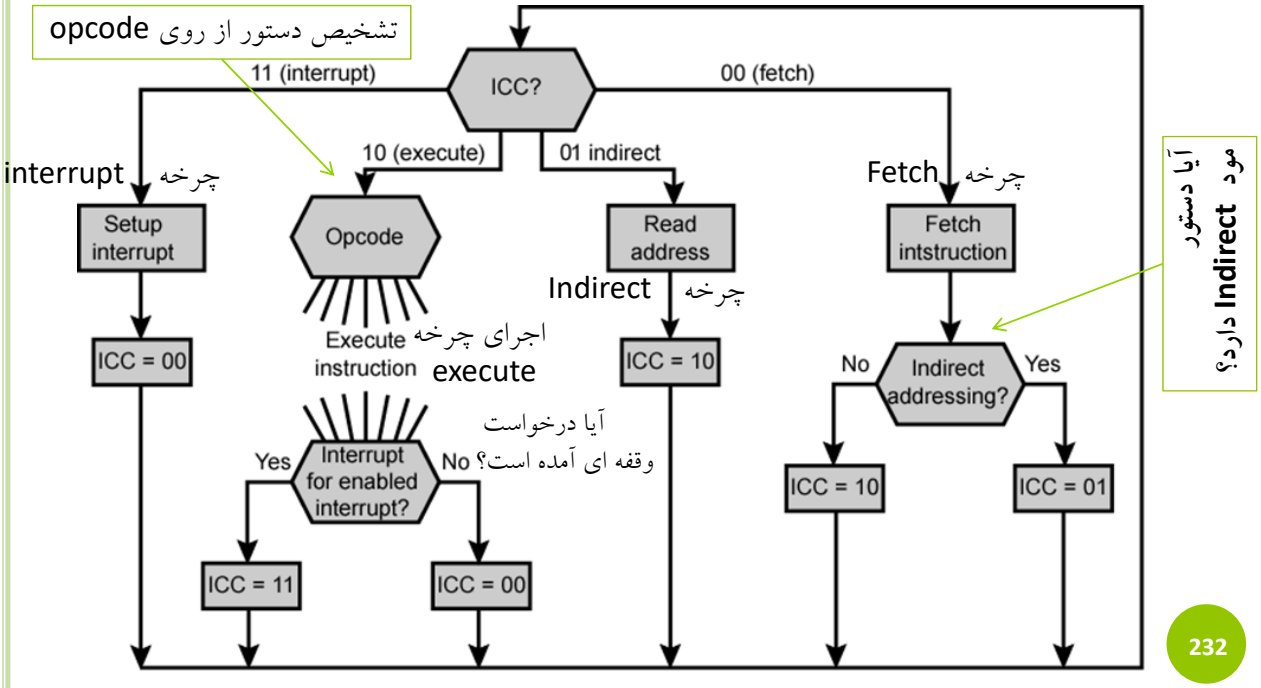
حال باید کاری کنیم که این چهار چرخه معرفی شده به ترتیب اجرا شوند. در این صورت یک دستور اجرا می شود. (چرخه دستور)

1. واحد کنترل باید کاری کند که در هر چرخه، µ-Op ها منظم اجرا شوند.
2. علاوه بر این واحد کنترل باید کاری کند که این چهار چرخه به ترتیب اجرا شوند.

- برای اینکار یک شمارنده ۲ بیتی در نظر گرفته می شود. (که از ۰ تا ۳ می شمارد.)
  - 00: Fetch
  - 01: Indirect
  - 10: Execute
  - 11: Interrupt
- این شمارنده در حال افزایش است و در انتها به مقدار صفر باز می گردد.
- چرخش این شمارنده به طور پشت سرهم باعث اجرای دستورها می شود.
- با یک چرخش کامل اجرای یک دستور کامل می شود.
- با عدد خروجی این شمارنده واحد کنترل متوجه می شود که هم اکنون در کدام چرخه اجرای دستور هستیم.

شکل زیر نشان می دهد که با تغییر مقدار شمارنده (ICC) در هر چرخه چه کاری انجام می شود. این چرخه به شکل مداری ساخته می شود.

- چرخه را با مقدار اولیه ICC=00 بررسی کنید و گام به گام جلو بروید.



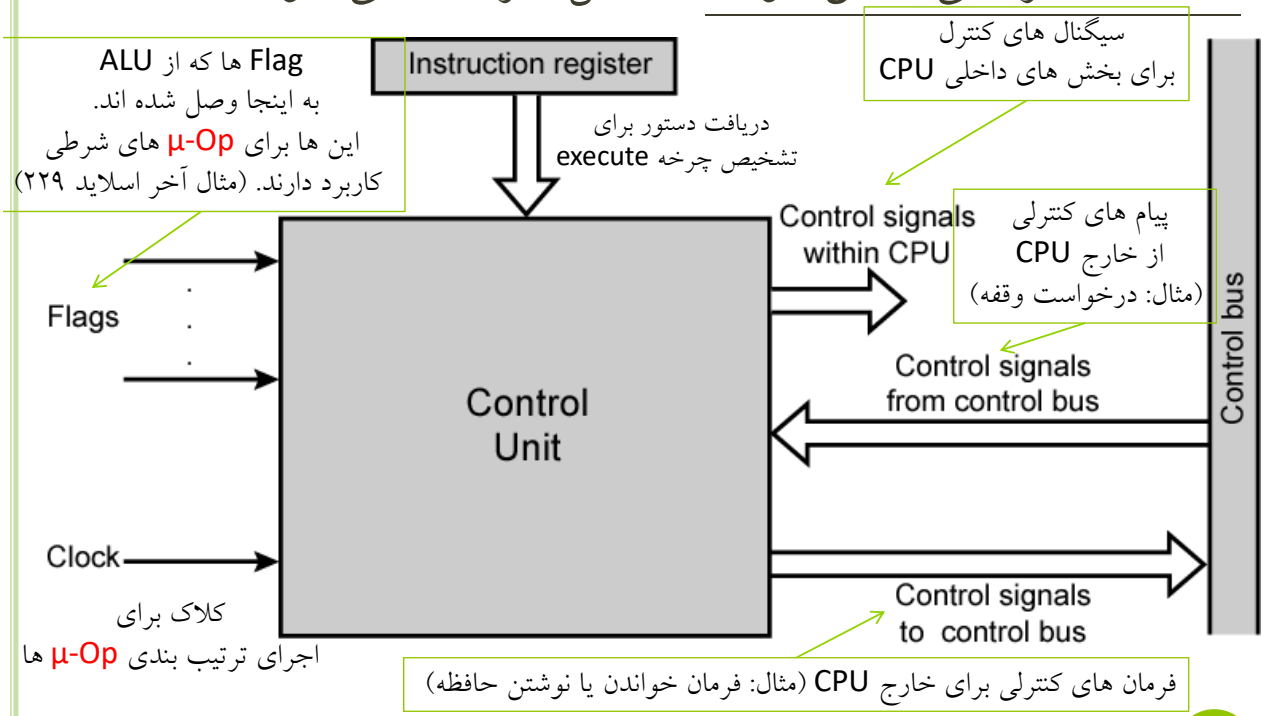
حال که با µ-Op ها آشنا شدیم باید ببینیم واحد کنترل چگونه آنها را منظم و پشت سرهم ایجاد می کند. و چگونه باعث اجرا شدن آنها می شود.

- وظیفه مداری که در واحد کنترل ساخته می شود دو کار است:
  - ترتیب بندی: کاری کند که CPU به ترتیب درست µ-Op های درست را برطبق برنامه کاربر اجرا کند.
    - این همان موضوعی است که در اسلاید قبل مطرح شد.
  - اجرای هر µ-Op: در هنگام اجرای هر µ-Op باید فرمان لازم به بخش های مختلف CPU را بفرستد.
- مثال:

t_1: MAR ← (PC)	باید خروجی PC به ورودی MAR وصل شود.
t_2: MBR ← Memory	باید باس داده به ورودی MAR وصل شود.
PC ← (PC) + I	باید PC یکی افزایش یابد.
t_3: IR ← (MBR)	باید خروجی MBR به ورودی IR وصل شود.

- فرمان هایی باید به بخش های مختلف CPU فرستاده شود: به اینها می گوییم **سیگنال های کنترل.**

اگر واحد کنترل را به شکل یک بلوک در نظر بگیریم سیگنال های ورودی و خروجی به این بلوک در شکل زیر دیده می شود.



توضیح دقیق تر شکل اسلاید قبل

- کلاک ورودی به واحد کنترل:
  - هر μ-Op (با μ-Op های همزمان) با یک کلاک اجرا می شود (می شوند).
- قسمت opcode در رجیستر IR وارد کنترل می شود.
  - با بررسی این بخش مشخص می شود که کدام چرخه execute باید اجرا شود.
  - همچنین تشخیص اینکه چرخه indirect باید اجرا شود یا نه از روی Opcode تشخیص داده می شود.
- Flag ها: نتیجه ALU را نشان می دهند. (صفر بودن و سرریز و علامت و...)
  - نتیجه عملیات قبلی است و در μ-Op های شرطی به کار می رود.
- سیگنال های دریافتی و ارسالی به باس کنترل:
  - برای فرمان دادن و دریافت پیغام ها به بخش های خارج از CPU (حافظه و ماجول های I/O)
- سیگنال های کنترل: فرمان های لازم برای اجرای هر μ-Op که به بخش های مختلف داخل CPU فرستاده می شود

به طور خلاصه واحد کنترل چه کار می کند؟

۱- اجرای منظم μ-Op ها (برمبنای برنامه کاربر)

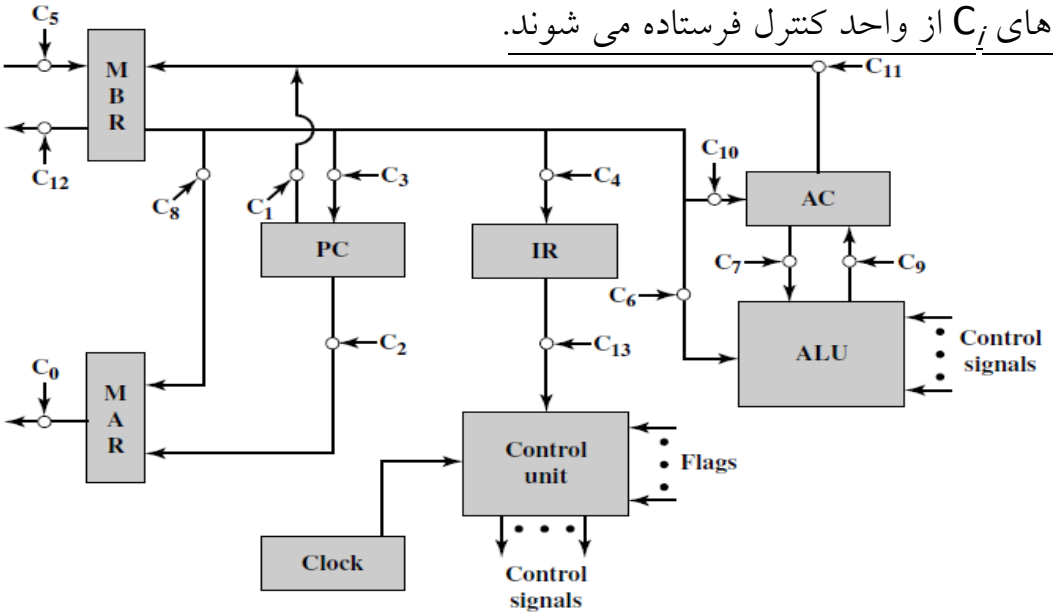
۲- اجرای درست هر μ-Op با فرستادن سیگنال کنترلی مناسب.

- همه μ-Op ها را می توان در دسته های زیر طبقه بندی کرد:

1. انتقال بین دو رجیستر:
   1. در این صورت نیاز است تا مسیر بین این دو رجیستر متصل شود.
2. انتقال بین رجیستر و مکانی خارج از CPU:
   1. این انتقال معمولاً بین MBR و باس داده انجام می شود.
   2. فرمان های خارجی لازم (مثال: نوشتن حافظه) باید همزمان انجام شود.
3. انتقال بین مکانی خارج از CPU و رجیستر:
   1. این انتقال معمولاً بین MBR و باس داده انجام می شود.
   2. فرمان های خارجی لازم (مثال: خواندن حافظه) باید همزمان انجام شود.
4. فرمان عملکرد ALU:
   1. فرمان لازم (نوع عملکرد) باید برای ALU فرستاده شود.

در اینجا با نشان دادن ساختار یک CPU فرضی و ساده روند کارکرد واحد کنترل را بررسی می کنیم.

- شکل زیر ساختار یک CPU ساده را نشان می دهد. واحد کنترل در پایین شکل است.
- هر دایره کوچک نشان دهنده دروازه ای است که قابل قطع یا وصل شدن است.
- فرمان های C_i از واحد کنترل فرستاده می شوند.



ادامه توضیح شکل اسلاید قبل

- واحد کنترل می تواند با فرستادن فرمان های مختلف بخش مختلف این CPU را به هم وصل کند یا قطع کند.
- همچنین می تواند با فرمان های مختلف کارکرد ALU را تغییر دهد (جمع، تفریق، ... یا شیفت یا عملیات منطقی دیگر).
- همچنین با فرمان های مناسب روی باس کنترل خارجی، برای حافظه ها فرمان نوشتن و خواندن در زمان مناسب می فرستد. (این فرمان ها روی شکل قبلی رسم نشده است.)
- پس مدار واحد کنترل مداری است که طبق یک روند مشخص (روند اجرای µ-Op ها) تغییر حالت می دهد و در هر حالت (هر µ-Op) خروجی مناسب (سیگنال های کنترل) را ایجاد کند.
- این دقیقا تعریف یک مدار ترتیبی است. (آنچه در درس مدارهای منطقی دیده اید.)

مثال اجرای سه چرخه FETCH، INDIRECT و INTERRUPT روی کامپیوتر دو اسلاید قبل.

- جدول زیر را همزمان با اسلاید ۲۳۷ مطالعه کنید.

	Micro-operations	Active Control Signals
Fetch:	t_1: MAR ← (PC)	C_2
	t_2: MBR ← Memory PC ← (PC) + 1	C_5, C_R
	t_3: IR ← (MBR)	C_4
Indirect:	t_1: MAR ← (IR(Address))	C_8
	t_2: MBR ← Memory	C_5, C_R
	t_3: IR(Address) ← (MBR(Address))	C_4
Interrupt:	t_1: MBR ← (PC)	C_1
	t_2: MAR ← Save-address PC ← Routine-address	
	t_3: Memory ← (MBR)	C_{12}, C_W

مسیری که باید وصل شود تا μ-Opمقابل اجرا شود.

C_R = Read control signal to system bus. فرمان خواندن حافظه که روی باس کنترل فرستاده می شود.

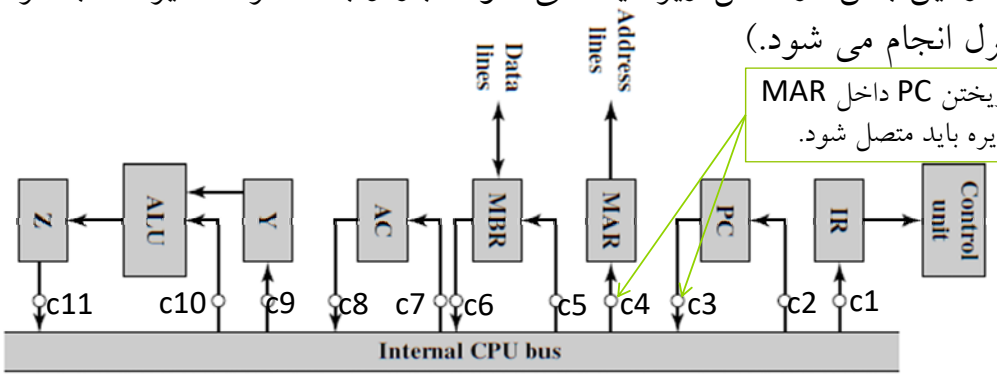
C_W = Write control signal to system bus. فرمان نوشتن حافظه که روی باس کنترل فرستاده می شود.

توضیح بیشتر برای جدول اسلاید قبل

- باید مداری بسازیم که با هر لبه کلاک بین μ-Op تغییر حالت دهد. (t1 و t2 و...)
- در جدول مرحله execute را نیاورده ایم چون این مرحله برای هر دستور متفاوت است.
- دقت کنید برای اجرای هر μ-Op باید فرمانی از واحد کنترل به بیرون فرستاده شود.
- مثلاً اگر بخواهیم PC به MAR انتقال یابد باید فرمان c2 را فعال کنیم.
- بعضی از فرمان ها روی جدول آورده نشده است. چون مدار شکل اسلاید ۲۳۷ کامل نیست و ساده است.
  - مثال: افزایش یک مقداری PC: باید یک فرمان از واحد کنترل به PC متصل باشد.
  - مثال: قسمتی که نوشته شده save address منظور همان SP است.
  - سیگنالی باید بین SP و MAR وصل باشد و کنترل اتصال آن باید سیگنالی مجزا داشته باشد.
  - جدول آدرس روتین های وقفه هم در این شکل آورده نشده است.

در برخی از معماری ها اجزای مختلف CPU روی یک باس داخلی قرار گرفته است.

- در این معماری ها، واحد کنترل ارتباط روی این باس داخلی را ایجاد می کند.
- این باس داخلی مجزا از باس خارجی CPU است.
- واحد کنترل دستور می دهد که کدام قسمت روی باس بنویسد و کدام قسمت از باس بخواند. این کار با برقراری اتصال بخش ها به باس انجام می شود.
- نمونه ای از این باس در شکل زیر دیده می شود: (باز و بسته کردن دایره ها به فرمان واحد کنترل انجام می شود.)



مثال: برای ریختن PC داخل MAR این دو دایره باید متصل شود.

مثال از معماری اسلاید قبل

در اینجا چرخه EXECUTE برای دستور ADD را می بینیم.

- این کامپیوتر به صورت یک اپرندی است.
- یعنی دستور ADD x: به معنای جمع x با AC و ذخیره آن در AC است.
- x آدرسی صریح در حافظه است.
- چرخه execute برای دستور ADD:

چون در شکل اسلاید قبل مسیر خروجی از IR رسم نشده اینجا فرمان را نیاورده ایم.

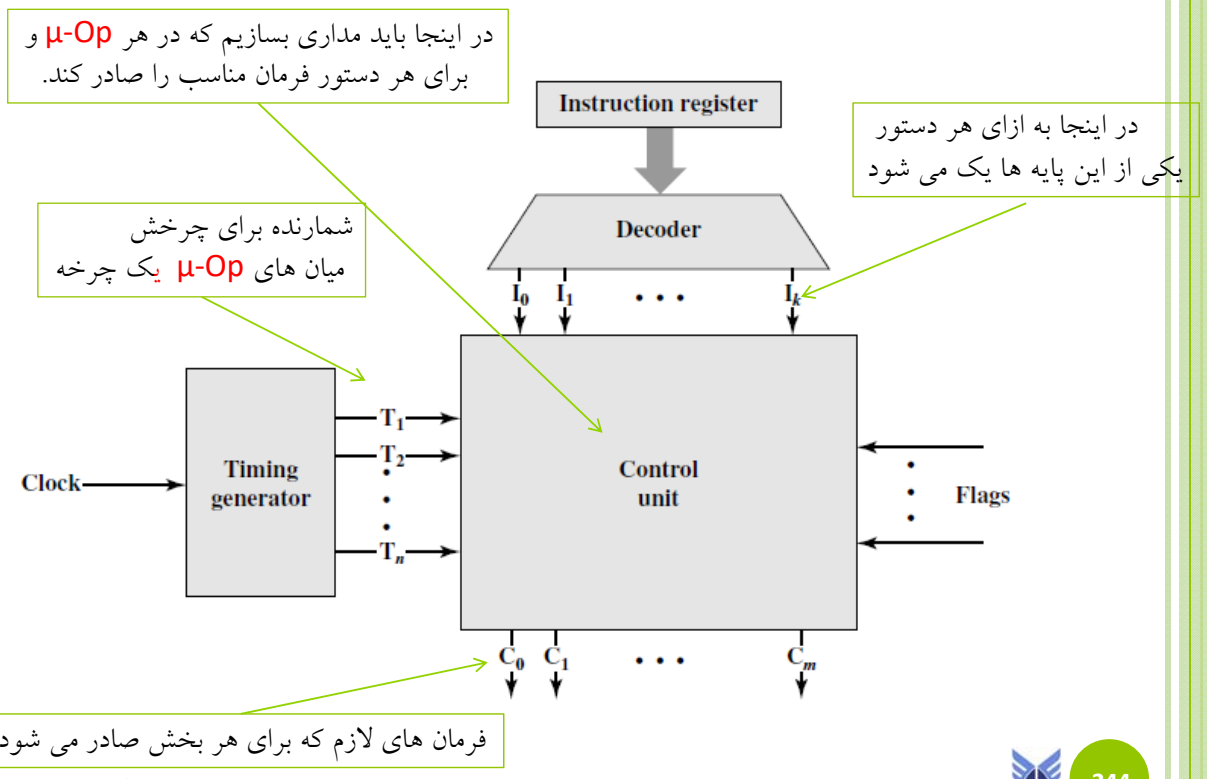
	فرمان های لازم خروجی از واحد کنترل
t_1: MAR ← (IR(address))	---
t_2: MBR ← Memory	Cr (فرمان خواندن حافظه)
t_3: Y ← (MBR)	c6, c9
t_4: Z ← (AC) + (Y)	C_{ALU} (فرمان جمع) (SUM), c10,c8
t_5: AC ← (Z)	c7, c11

اما نحوه ساخت مدار داخلی واحد کنترل چگونه است؟

دو روش وجود دارد: ۱- سیم بندی ثابت ۲- میکروپرگرام (ریزبرنامه)

- تا به حال درباره عملکرد واحد کنترل صحبت کردیم. یعنی اینکه واحد کنترل باید این کارها را انجام دهد: (۱- اجرای مرتب µ-Op ها. ۲- ارسال درست فرمان ها در هر µ-Op).
- حال می خواهیم ببینیم چگونه می توان مداری ساخت که این کارها را انجام دهد.
- یک روش برای انجام این کار سیم بندی ثابت نام دارد.
- در این روش مدار کنترل اساساً یک شمارنده است که با کلاک ساعت کار می کند.
- این شمارنده به ترتیب بین حالت های مختلف حرکت می کند.
- هر حالت نماینده یک زمان µ-Op است. (برابر با t1 و t2و ... در اسلایدهای پیشین)
- از سوی دیگر قسمت Opcode دستور (در IR) در ورودی یک دیکوردر قرار می گیرد. تا برای هر دستور یک سیگنال مجزا تولید شود.
- سپس مدار لازم برای هر فرمان خروجی با گیت های AND و OR و NOT ساخته می شود.

ساختار واحد کنترل در حالت سیم بندی ثابت در شکل زیر دیده می شود.

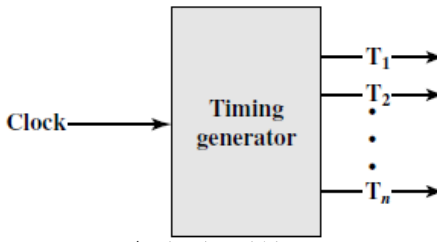


به عنوان مثال برای کامپیوتر اسلاید ۲۳۷ و برای چرخه های اسلاید ۲۳۹ مدار بخش کنترل را طراحی می کنیم.

- در اسلاید ۲۳۹ سه چرخه آورده شده است. و چرخه execute آورده نشده است. فعلاً طراحی را برای این سه چرخه انجام می دهیم.
- فرض می کنیم شمارنده ای داریم که مقدار آن چرخه در حال اجرا را نشان می دهد. (موضوع اسلاید ۲۳۱ و ۲۳۲)

PQ = 00	Fetch Cycle
PQ = 01	Indirect Cycle
PQ = 10	Execute Cycle
PQ = 11	Interrupt Cycle

- و شمارنده ای هم داریم که شماره کلاک (t1و t2و t3و....) را نشان می دهد.



ادامه مثال: طراحی واحد کنترل به روش سیم بندی ثابت

- حالا جدول ۲۳۹ را در نظر بگیرید: در چه زمانی به عنوان مثال c2 باید فعال شود؟
  - پاسخ: فقط در زمان fetch و در زمان t1.
  - پس مدار لازم برای ساخت c2 می شود: c2= P'.Q'.T1
  - یا عبارتی زمانی باید c2 فعال شود که P=0 و Q=0 و T1=1 باشد.
- باید برای همه سیگنال های کنترل به این شکل مدار طراحی کرد.
- مثال دیگر: برای c5 مدار طراحی کنید. چه زمانی c5 باید فعال شود؟
  - پاسخ: دو جا فعال می شود: (در fetch و زمان T2) یا (در indirect و زمان T2)
  - پس مدار لازم می شود: c2= P'.Q'.T2 + P'.Q.T2
  - که + به معنای گیت OR است. اگر یکی از این حالت ها اتفاق بیافتد c2 فعال می شود.
- دقت کنید که فعلاً مدار را برای سه چرخه طراحی کرده ایم. این سه چرخه برای همه دستور ها اجرا می شود و ثابت است. پس نوع دستور تا به حال تاثیری در طراحی مدار نداشته است.

حال فرض کنید بخواهیم چرخه EXECUTE را برای دستورها را هم اضافه می کنیم.

- باید ابتدا چرخه execute را برای همه دستورها نوشت.
- سپس فرمان های لازم برای هر μ-Op را پیدا کرد.
- باید ببینیم که هر فرمان در چه مواقعی فعال می شود.
- سپس با گیت های AND و ORو NOT مدار لازم را مانند قبل طراحی کرد.
- تنها تفاوت این است که عنوان دستور وارد مدار فرمان می شود.
- اگر فرض کنیم همه دستورها دارای شماره هستند (اسلاید ۲۴۴: I0و I1 و I2و ...).
- و اگر به عنوان مثال قرار است سیگنال c4 در زمان t2 چرخه execute دستور I2 فعال شود، مدار سیگنال کنترل c4 می شود: c4 = P.Q'.T2.I2.
- بدیهی است اگر در جای دیگری c4 باید فعال شود، آن جمله به شکل OR به c4 = P.Q'.T2.I2 افزوده می شود.

روش سیم بندی ثابت روشی زمان گیر و غیر قابل انعطاف است.

- ساخت مدار به این روش پیچیده و زمان گیر است.
- مخصوصاً اگر تعداد دستورها زیاد شود.
- خطا یابی، و تصحیح خطای طراحی بسیار مشکل می شود.
- اگر بخواهیم دستوری جدید اضافه کنیم کل مدار را باید از نو طراحی کنیم.
- تغییر در طراحی و بهینه سازی مستلزم طراحی دوباره است.
- روش مناسب تر طراحی واحد کنترل روشی است با نام میکروپرگرام که موضوع بخش بعدی است.

طراحی واحد کنترل به روش MICROPROGRAM (میکروپروگرام)
(خلاصه آنچه تاکنون گفتیم.)

- همان طور که دیدیم وظیفه واحد کنترل اجرای دستورها به ترتیب مناسب است.
- هر دستور تشکیل شده از چهار سیکل است که هر سیکل نیز از چند µ-Op تشکیل شده است.
- پس برای اینکه هر دستور اجرا شود چندین µ-Op به ترتیب باید اجرا شود.
- واحد کنترل اجرای هر µ-Op را با فرستادن سیگنال های کنترل مناسب به بخش های مختلف CPU و باس کنترل خارج از CPU انجام می دهد.
- بنابرین می توان گفت هر µ-Op معادل با مجموعه ای از سیگنال های کنترل است.
- برای مثال به اسلاید ۲۳۹ و ۲۴۲ مراجعه کنید. در مقابل هر µ-Op مجموعه ای سیگنالهای کنترلی معادل نوشته شده است.

طراحی واحد کنترل به روش MICROPROGRAM (میکروپروگرام)
ایده میکروپروگرام چیست؟

- اگر همه سیگنال های کنترلی خارجی و داخلی CPU را اسم گذاری کنیم مانند زیر:

C1	C2	C3	C4	C5	C6	....	Cn

- می توان هر µ-Op را با توجه به سیگنال های کنترلی آن معرفی کرد:
- به عنوان مثال µ-Op زیر دارای سه سیگنال کنترلی c1، c2، و c3 است:
- یعنی هنگام اجرای آن سه فرمان زیر باید فعال شوند و بقیه غیرفعال هستند.

C1	C2	C3	C4	C5	C6	....	Cn
1	1	1	0	0	0	....	0

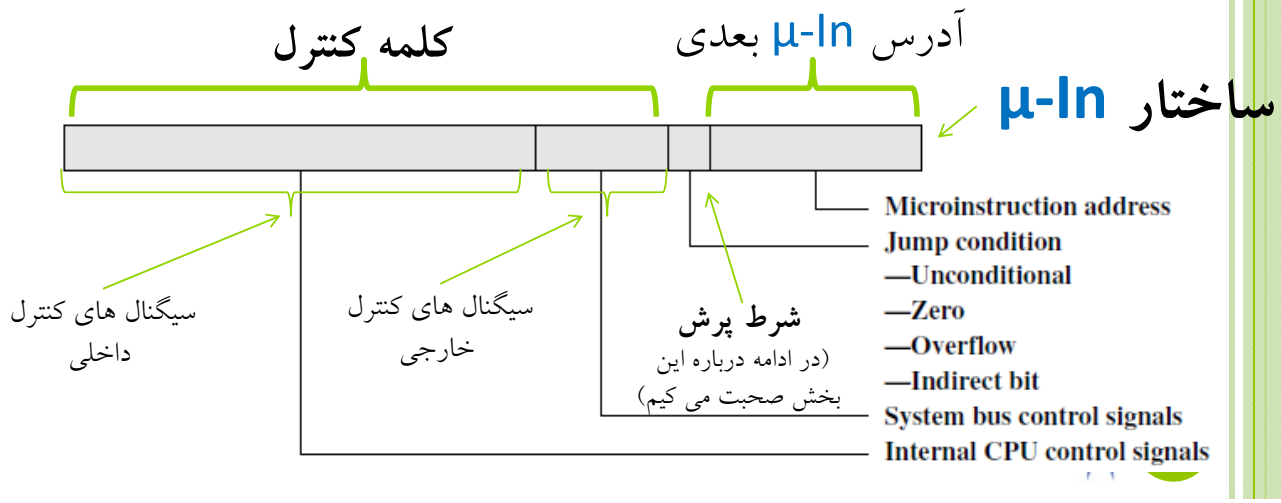
- به این مجموعه این بیت ها می گوییم کلمه کنترل (Control word)
- از این پس هر µ-Op را با کلمه کنترل آن می شناسیم.
- به ازای همه µ-Op های موجود (در سه سیکل مشترک و سیکل execute) این کلمه کنترل را تشکیل می دهیم.

ایده طراحی به روش میکروپروگرام چیست؟

- همه کلمه های کنترل را در حافظه ای نگه دارای می کنیم و به هریک آدرسی یکتا تخصیص می دهیم.
- این حافظه مکانی جدید در واحد کنترل است. (جدا از همه حافظه هایی است که تاکنون مطالعه کردیم.) و آن را با حافظه میکرو پروگرام می شناسیم.
- اگر بتوانیم به ترتیب مناسب محتوای این حافظه را فراخوانی کرده به عنوان سیگنال های کنترل به کار ببریم عملاً µ-Op ها را به ترتیب مناسب اجرا کرده ایم.
- پس به طور خلاصه می توان گفت:
  - در روش میکروپروگرام، کلمه های کنترل به ترتیب از حافظه میکروپروگرام فراخوانی می شوند. این ترتیب منطبق بر اجرای µ-Op ها است.
- این ساختار خیلی شبیه اجرای دستورها در حافظه اصلی است. (fetch و Execute)
- بنابرین اسم این روش را گذاشته اند: micro-programing (یا firmware)
- اسم هر دستور این ریزبرنامه (که همان کلمه های کنترل است.) را هم گذاشته اند ریز–دستور (micro-instruction).
- دقت کنید که ریز–دستور (µ-In) را با µ-Op اشتباه نکنید.

ریزدستور (µ-In) معمولاً از دو بخش تشکیل می شود
۱– بیت های کلمه کنترل ۲– اطلاعاتی برای تعیین آدرس µ-In بعدی

- ریز دستوری که از حافظه میکروپروگرام خوانده می شود دارای دو بخش است.
- بخش اول همان کلمه کنترل است. این بیت ها شامل سیگنال های کنترل داخلی و خارجی است.
- بخش دوم دارای اطلاعاتی است که واحد کنترل از آنها متوجه می شود که µ-In بعدی را باید از کدام آدرس در حافظه میکروپروگرام بردارد.



روند اجرای هر µ-In

1. پس از خواندن از حافظه میکروپروگرام بخش کلمه کنترل عیناً به سیگنال های فرمان خروجی از واحد کنترل وصل می شود.
2. یک بخش شرط وجود دارد: در این بخش ابتدا یک شرط بررسی می شود (که معمولاً flag ها هستند و یا شماره سیکلی که در آن هستیم.)
3. اگر این شرط برقرار باشد، µ-In بعدی از آدرسی که در ادامه آورده شده (آدرس µ-In بعدی) برداشته خواهد شد.
4. اگر شرط برقرار نباشد، µ-In از آدرس بعدی برداشته می شود. (مشابه اینکه دستور را از خط بعدی اجرا می کردیم.)

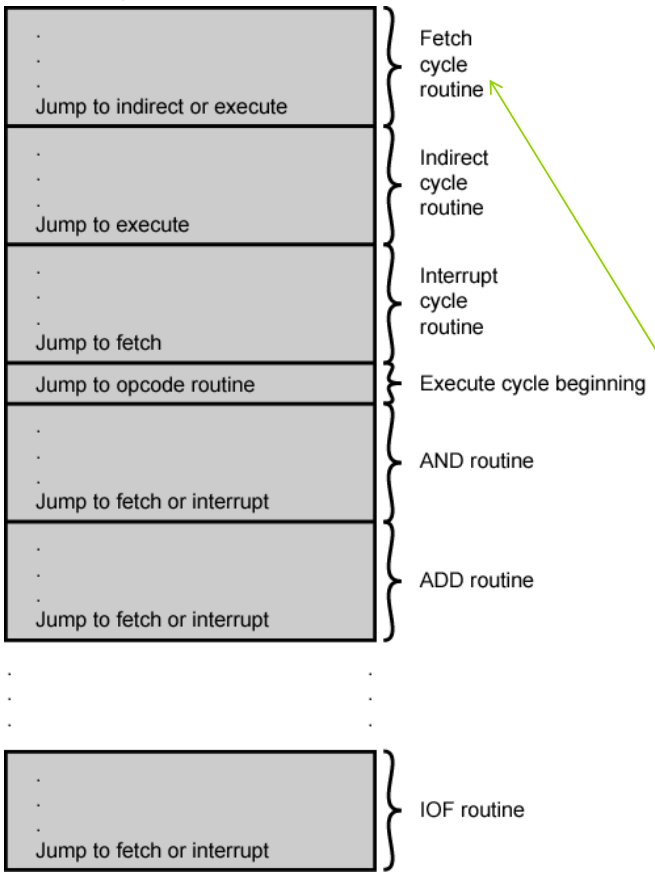
- می دانیم که هر سیکل (fetch، ...) از چند µ-Op تشکیل شده است.
- در واحد کنترل برای اجرای هر µ-Op یک µ-In تعریف شده است.
- این µ-In ها به ترتیب در حافظه میکروپروگرام ذخیره شده است.

ساختار حافظه میکروپروگرام

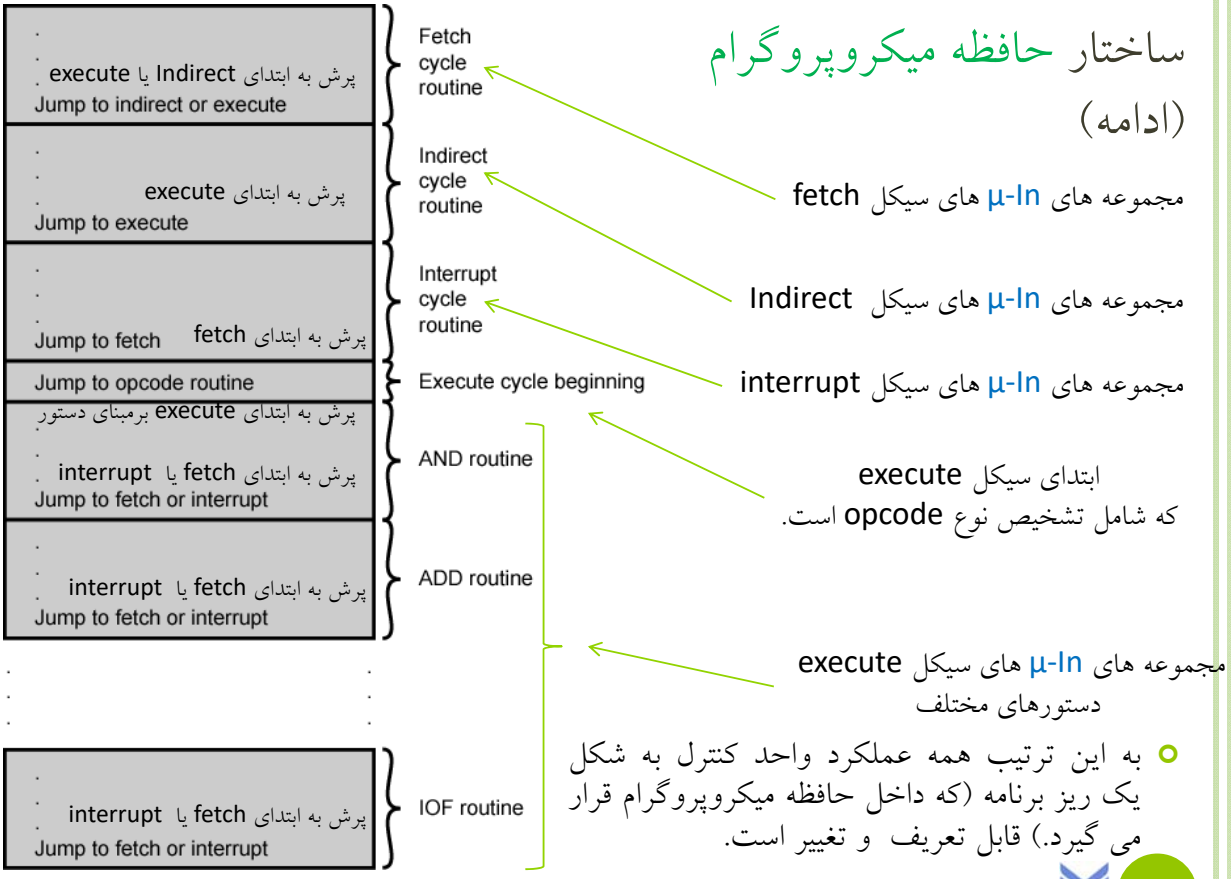
این شکل نحوه قرار گیری µ-Inها را در حافظه میکروپرگرام نشان می دهد.

µ-In ها به شکل منظم در حافظه میکروپروگرام ذخیره می شوند.

- در این حافظه µ-In های مرتبط که همیشه پشت سرهم انجام می شوند در آدرس های پشت سرهم قرار می گیرند.
- به عنوان مثال µ-In های fetch پشت سرهم قرار داده شده ند.
- در آخرین µ-In هر بخش یک پرش قرار داده شده است.
- دقت کنید این پرش در حوزه µ-In قرار دارد و پرش به آدرسی دیگر در حافظه میکروپروگرام انجام می شود.



ساختار حافظه میکروپروگرام (ادامه)



- به این ترتیب همه عملکرد واحد کنترل به شکل یک ریز برنامه (که داخل حافظه میکروپروگرام قرار می گیرد.) قابل تعریف و تغییر است.

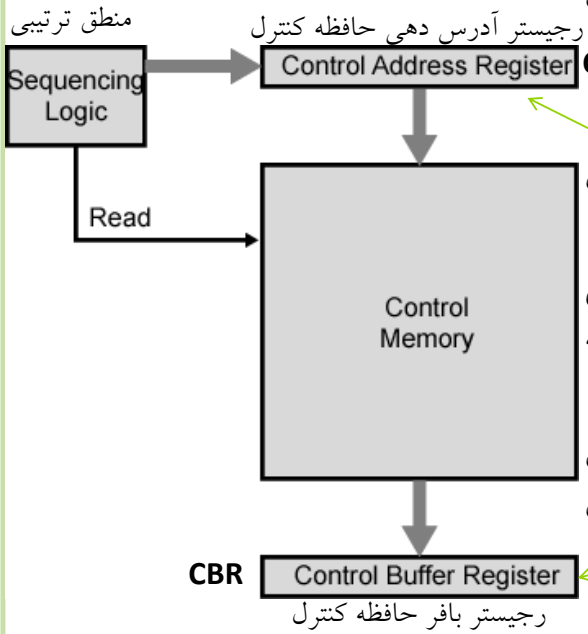
مشکلات موجود بر سر راه ساخت واحد کنترل به روش میکروپروگرام (و برتری های آن)

- در پردازنده های امروزی
  - بخش های داخلی CPU و اتصالات آنها بسیار زیاد شده اند. ← کلمه کنترل بسیار بزرگ است.
  - تعداد دستورها زیاد شده است. ← تعداد µ-In ها بسیار زیاد می شود.
- در نتیجه حافظه میکروپروگرام بزرگ می شود.
  - این حافظه هم تعداد زیادی کلمه بزرگ خواهد داشت.
  - البته اندازه این حافظه خیلی کوچکتر از حافظه اصلی و کوچکتر از حافظه cache خواهد بود.
- چون اینجا هم خواندن از حافظه و اجرای ریز برنامه وجود دارد سرعت اجرای برنامه کمی نسبت به حالت سیم بندی ثابت کندتر می شود.
- اما برتری میکروپروگرام نسبت به سیم بندی ثابت چیست؟
  - طراحی بسیار ساده تر و کم خطاتر می شود: کل طراحی مشابه برنامه نویسی می شود.
  - بسیار قابل انعطاف تر می شود: مثال: دستور جدید به معنای اضافه کردن چند خط برنامه است.
  - تغییر واحد کنترل بسیار ساده می شود: تغییر برنامه معادل با تغییر طراحی مدار است.

اصول میکروپروگرام را دیدیم.

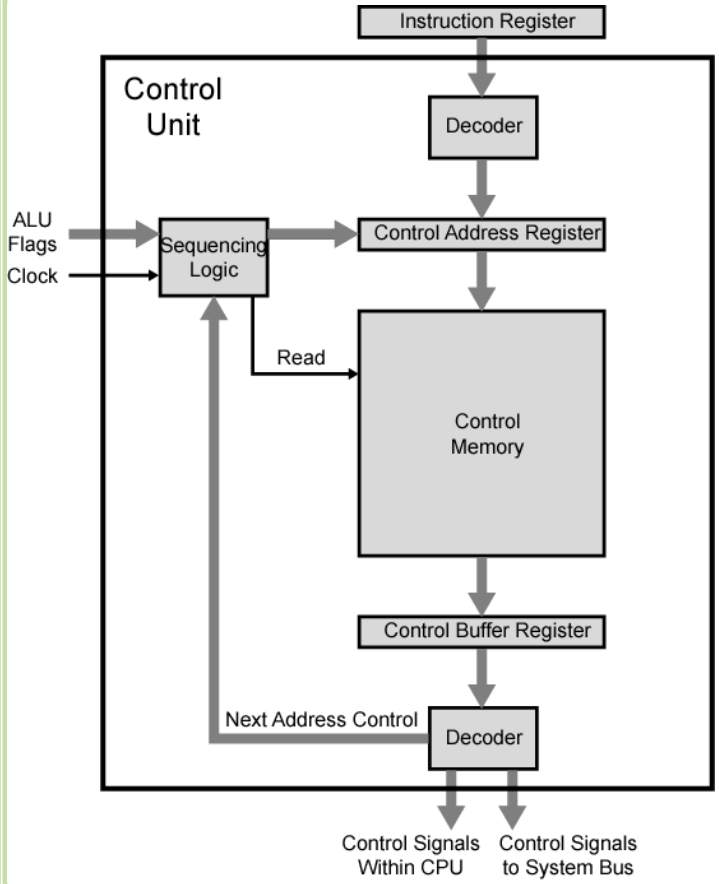
حال به نحوه ساخت مدار کنترل به روش میکروپروگرام می پردازیم.

- شکل زیر مدار واحد کنترل را در ساده ترین حالت نمایش داده است.
- با حافظه کنترل (میکروپروگرام) آشنا هستیم.
- برای اجرای یک دستور به ترتیب باید از این حافظه µ-In ها را بخوانیم.
- برای اینکار یک رجیستر (با نام CAR) برای آدرس دهی این حافظه نیاز داریم (مشابه MAR برای حافظه اصلی)
- و یک رجیستر (با نام CBR) برای دریافت داده از این حافظه مورد نیاز است (مشابهMBR برای حافظه اصلی).



نحوه ساخت مدار کنترل به روش میکروپروگرام (ادامه)

- منطق ترتیبی وظیفه دارد تا آدرس µ-In بعدی را ایجاد کند.
- سپس این منطق در زمان مناسب فرمان خواندن را برای حافظه کنترل می فرستد.
- محتوای خوانده شده از حافظه (که همان µ-In است.) به CBR منتقل می شود.
- µ-In خوانده شده که در CBR است دارای دو بخش است. (اسلاید ۲۵۲)
  1. بخش کلمه کنترل آن به عنوان سیگنال های کنترل به کار گرفته می شود تا µ-Op مورد نظر اجرا شود.
  2. بخش آدرس و شرط هم تحویل منطق ترتیبی خواهد شد تا آدرس بعدی µ-In را تشخیص دهد.
- منطق ترتیبی چگونه آدرس بعدی را تشخیص می دهد؟ سه حالت اتفاق می افتد.
  - یا از آدرس بعدی باید برداشته شود که در این صورت CAR یکی افزایش می یابد.
  - یا بخش آدرس CBR در CAR قرار می گیرد.
  - یا برمبنای نوع دستور پرش انجام می شود. (این بخشها در ص ۲۶۳ بیشتر بررسی می شود.)
- این سه حالت برمبنای بررسی شرط µ-In فعلی انجام می شود.

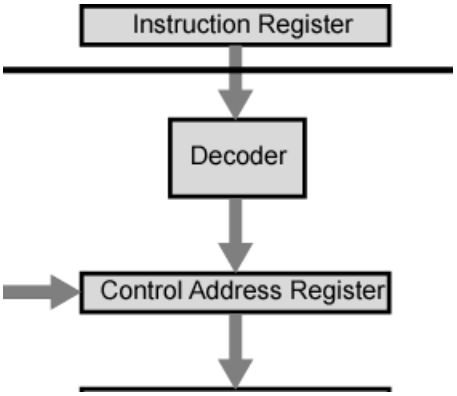


با توجه به مطالب گفته شده مدار کامل تر واحد کنترل به روش میکروپروگرام را بررسی می کنیم.

- بخش های جدید روی این شکل:
- دو decoder.
- دقت کنید که این decoder ها به معنای decoder عادی نیستند و کاربرد خاصی دارند.
- کارکرد این دو را در دو اسلاید بعدی توضیح می دهیم.

به ازای هر دستور باید بخشی از حافظه میکروپروگرام اجرا شود.
DECODER بالایی آدرس ابتدای این بخش را می سازد.

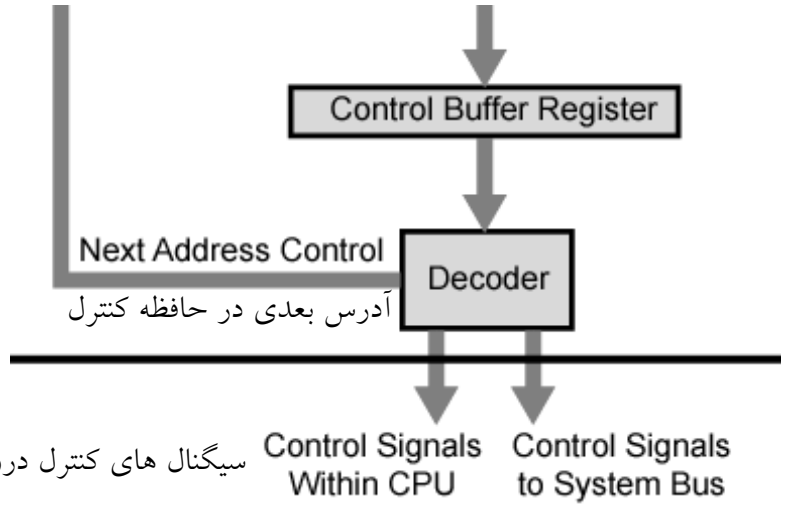
- وظیفه decoder بالایی:
- به ازای هر دستوری که از قسمت Opcode در رجیستر IR تشخیص داده می شود آدرس ابتدای بخش مربوط به آن دستور در حافظه میکروپروگرام انتخاب می شود. (اسلاید ۲۵۵ را دوباره ببینید. برای هر دستور بخشی در حافظه کنترل قرار داده شده است. وظیفه این دیکودر ساختن این آدرس به ازای هر دستور است.)



- در ابتدای سیکل execute این آدرس درون CAR قرار می گیرد.
- ساخت این آدرس بر مبنای جدولی از پیش تعیین شده انجام می شود.

DECODER پایینی بخش های مختلف را جدا می کند و به قسمت های مختلف واحد کنترل و CPU و بیرون CPU می فرستد.

- Decoder پایین کلمه کنترل را استخراج می کند و به وسیله آن سیگنال های کنترلی را می سازد. همچنین بخش آدرس بعدی را نیز جدا می کند.



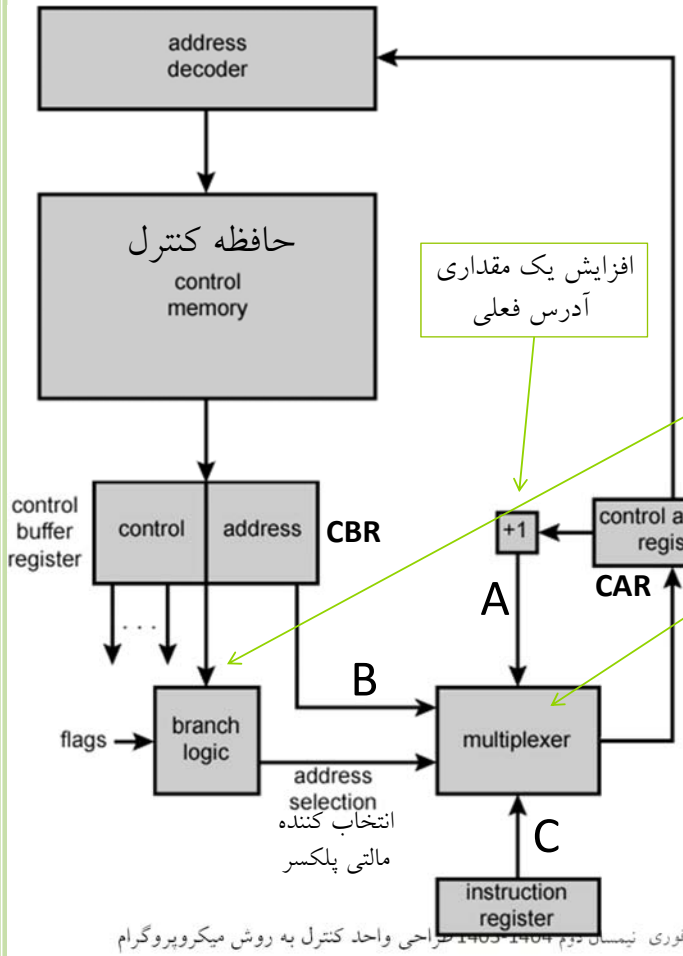
واحد کنترل میکروپروگرام چه وظیفه هایی دارد؟

1. ترتیب بندی درست µ-In ها: اجرای درست و به ترتیب.
   - واحد کنترل میکروپروگرام باید بتواند به ازای هر دستور µ-In های مناسب را اجرا کند.
   - یعنی ابتدا µ-In های fetch را پشت سرهم اجرا کند (که در آدرس های پشت سرهم قرار گرفته اند.)
   - سپس µ-In های indirect را در صورت لزوم اجرا کند. (که در آدرس های پشت سرهم قرار گرفته اند.) اگر نیازی نبود باید بتواند به آدرس ابتدای execute پرش کند.
   - در ابتدای execute باید براساس opcode موجود در IR به ابتدای سکیل execute هر دستور پرش انجام دهد. (µ-In های هر دستور در آدرس پشت سرهم قرار گرفته اند.)
   - و در انتهای execute باید برحسب نیاز پرش به مکان مناسب انجام شود.
2. اجرای هر µ-In: که به معنای ساختن سیگنال های کنترلی است. و آماده کردن آدرس µ-In بعدی است.

ترتیب بندی درست µ-In ها به معنای تشخیص درست آدرس µ-In بعدی است. پس مدار باید این کار را انجام دهد.

- پس از اجرای هر µ-In آدرس µ-In بعدی به یکی از این سه شکل تعیین می شود.
  A. یکی بیشتر از آدرس فعلی.
  B. پرش به آدرس داده شده درون µ-In فعلی.
  C. پرش به آدرس دیکود شده از opcode (اسلاید ۲۶۰)
- پس باید مداری داشته باشیم که در شرایط مناسب یکی از سه حالت بالا را انتخاب کند و به عنوان آدرس بعدی در باس آدرس حافظه کنترل قرار دهد.
- حالت A همیشه در زمانی غیر از کلاک آخر هر سیکل اتفاق می افتد.
- حالت B همیشه در زمان انتهای هر سیکل اتفاق می افتد. (با توجه به شرایط)
- و حالت C تنها در اولین سیکل چرخه execute اتفاق می افتد.
- پس به راحتی می توان مداری ساخت که این سه حالت را تشخیص دهد.

مدار تعیین آدرس µ-In بعدی



- مدار باید یکی از سه حالت اسلاید قبل ر انتخاب کند.
- این انتخاب توسط این مدار انجام می شود.
- سپس توسط این مدار یکی از این سه حالت انتخاب می شود.
- این مدار یک مالتی پلکسر است.
- مداری برای افزایش یک مقداری آدرس فعلی قرار داده شده است.
- در قسمت C این شکل decoder ای باید قرار می گرفت که رسم نشده است.