رجیستر 1 بیتی : دارای قابلیت set, reset, load به صورت synchronized ,طراحی به صورت behavioral .

Register File : برای داشتن 64 رجیستر 16 بیتی ، تایپ reg_64n16b را به این صورت تعریف میکنیم که یه آرایه 64تایی از یک STD_LOGIC_VECTOR 16 بیتی باشد.

** لود رجیسترفایل به این صورت است : اگر هر دو بیت لود روشن باشند، کل 16بیت ورودی روی Rd لود میشود. ولی اگر فقط یکی روشن باشد ، 8بیت کم ارزش دیتای ورودی را روی 8بیت کم ارزش یا پرارزش Rd لود میکند. (بر اساس آنکه کدامیک از بیتهای لود روشن است)

رجیستر 16 بیتی : رجیستر 16بیتی که برای IR نیاز میشود. با قابلیت load و reset که همگی synchronized هستند.

Flags : که شامل دو رجیستر تک بیتی c و z است. که بوسیله کامپننت رجیستر 1بیتی پیاده سازی میشود.

WP : یک std_logic_vector 6بیتی برای نگهداری مقدار رجیستر و استفاده آن در هنگام WPadd .

PC : کد را از تعریف پروزه کپی میکنیم ☺ (و تصحیح 😐)

AddressLogic : کد را از تعریف پروزه کپی میکنیم☺ (و تصحیح 😐)

AddressUnit : کد را از تعریف پروزه کپی میکنیم ☺ (و تصحیح 😐)

رجیستر فایل را بوسیله registerFile-test.vhd تست میکنیم (خواندن و نوشتن و جا به جا کردن WP و local address .

Mux و TriState را بویسله when و else پیاده سازی میکنیم.

و به صورت generic هم پیاده سازی شده اند. به این دلیل که بتوانیم برای ورودی خروجی با تعداد بیتهای مختلف از همین کامپوننت استفاده کنیم یک عدد ورودی کامپوننت ها میگیریم که تعداد بیت ورودی خروجی میباشد و مدار را بر اساس آن عدد پیاده سازی میکنیم.

سینتکس برای generic به این صورت است : در تعریف کامپوننت آنجا که پورت ها را در entity مشخص میکنیم قبلش این را مینویسیم :

```
generic(
      DATA_LENGTH   : integer := 1    -- size of input and output
);
```

و برای در اتصال سیمها هم صرفا وصل میکنیم و یا در tristate به این گونه در حالت enable=0 ، z میدهیم :

```
(OTHERS => 'Z');
```

ALU :

کامپوننتهای and, or, را بویسله گیت های پیشفرض پیاده سازی میکنیم.

کامپوننت xor : ابتدا کامپوننت xorSingleGate را میسازیم که xor را برای دو تک بیتی محاسبه کند. به این صورت:

output <= (input1 and not input2) or (not input1 and input2)

و سپس در یه حلقه 16 for-generate بار از این گیت instance میگیریم و بیتهای خروجی را پر میکنیم.

کامپوننت invert : با یک حلقه بیت به بیت را not کرده و در خروجی می گذاریم.

کامپوننتهای shift را بر اساس subVector گرفتن از STD_LOGIC_VECTOR و کانکتینیشن(&) پیاده سازی میکنیم.

کامپوننت comparison را بوسیله when و else پیاده سازی میکنیم.

کامپوننت add را بویسله عملیات جمع std_logic در unsigned پیاده سازی میکنیم.و در یک vector 17بیتی میریزیم و بوسیله آن خروجی و carry را میابیم. (** unsigned کردن در محاسبه برای اعداد منفی مشکلی ایجاد نمیکند)

کامپوننت sub را هم بوسیله جمع std_logic در unsigned پیاده سازی میکنیم به این صورت که اولی را با complement دومی جمع میکنیم.

کامپوننت mul : یک variable از جنس unsigned در نظر میگیریم(16 بیتی).برای انجام ضرب یک حلقه به روی عدد اول انجام میدهیم و هر بار عدد دوم را با variable تعریف شده جمع میکنیم. (به صورت unsigned جمع میکنیم)

کامپوننت two’s complement : ابتدا بوسیله کامپوننتinvert انرا invert میکنیم و سپس با 1 جمع کرده و در خروجی میگذاریم.

کامپوننت تولید عدد رندم : از روش استانداردی به اسم Fibonacci LFSR یا همان many-to-one استفاده میکنیم. به این صورت است که یک رجیستر 16 بیتی داریم . هر دفعه xor خانه های 10 و 12 و 13 و 15 را از سر رجیستر وارد میکنیم و رجیستر را یک شیفت میدهیم. این عمل را هر چند لحظه ی کوچک یک بار انجام میدهیم و هر موقعی که عدد رندم بخاهد بخواند مقدار رجیستر را میخواند و در لحظات مختلف مقادیر رندم متفاوتی میبیند. (روش استفاده شده : https://en.wikipedia.org/wiki/Linear-feedback_shift_register)

نمونه اعداد رندم تولید شده : 27016, 54032, 42529, 19522, 39045, 12555, 25110, 50220, 34905, 4274, 8549, ...

کامپوننت ALU : از همه کامپوننتهایی که ALU از انها استفاده میکند یک اینستنس میگیریم.

بیتهای که مشخص میکنند کدام عملیات باید انجام شود را با هم کانکت میکنیم و یک logic vector داریم. سپس روی آن with/select (selected signal assignment) میزنیم و مشخص میکنیم که خروجیه کدام کامپونتت به خروجی ALU متصل باشد. همچنین برای دو کامپوننت Add و Comparison خروجی های carry flag و zero flag را به خروجی ALU متصل میکنیم تا به کامپوننت flags برسند.

کامپوننت alu-test را برای تست کردنه کامپوننتهای alu مینویسیم و با عوض کردن سیگنال های کنترلی operatorها میختلف را بررسی میکنیم.

DataPath :

بر اساس شکل 3 و بیتهای کنترلی، entity و portهای CU را مینویسیم.

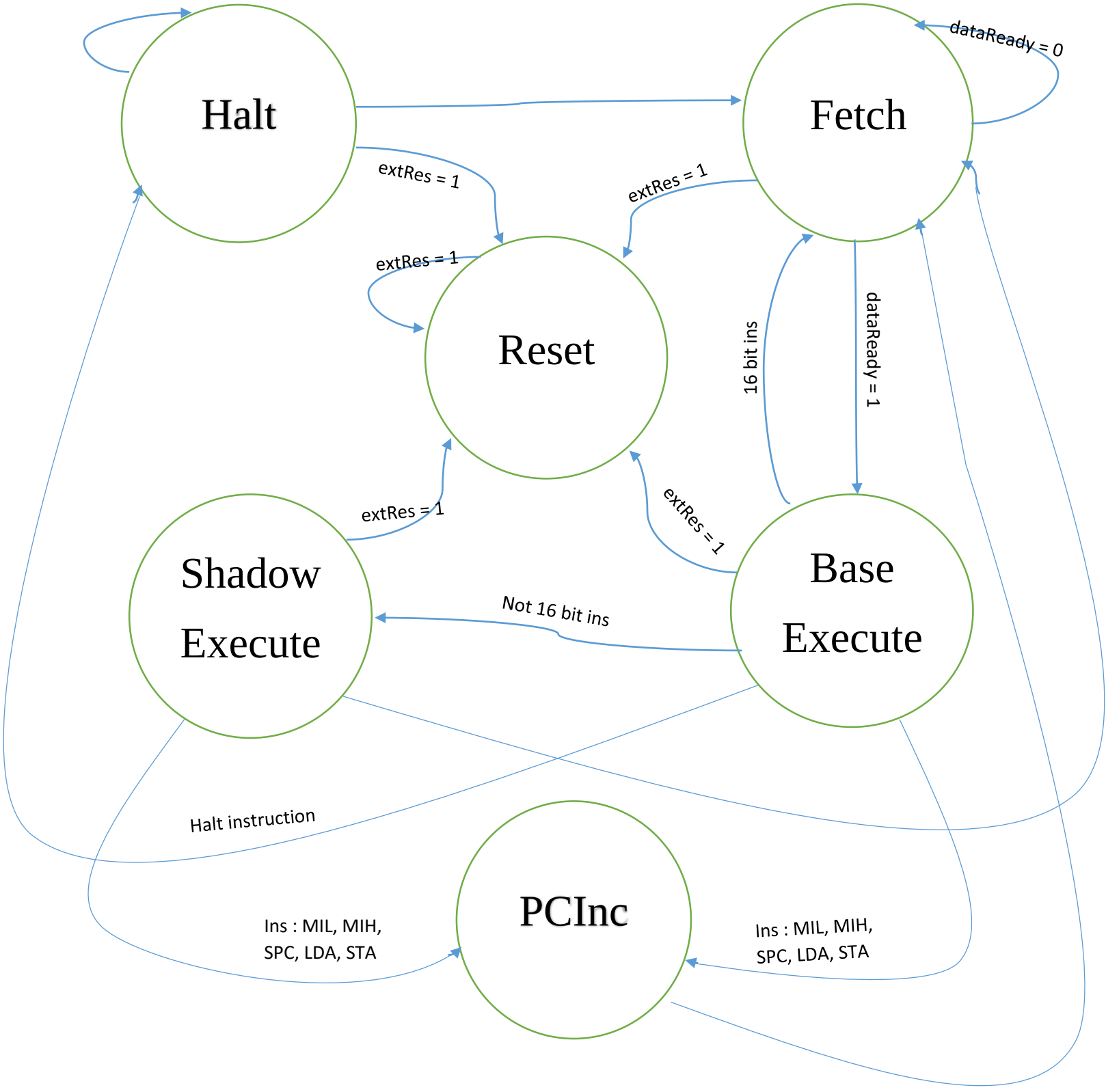
* ابتدا فقط entityه کامپوننت CU را میسازیم و سپس بویسله آن و دیگر کامپوننتهایی که داریم کامپوننت SAYEH را میسازیم و سیمهای data و سیمهای کنترلی را وصل میکنیم.

CU :

واحد کنترل را به صورت FSM طراحی میکنیم.

ابتدا باید یک سری استیت طراحی کنیم برای کارهای مختلفی که میخواهیم انجام دهیم :

: State Machin



یک پراسس به روی currentState ایجاد میکنیم که در آن بر اساس اینکه در چه کیسی هستیم کارهای مربوطه را انجام میدهیم. در ابتدای این پراسس تمام بیت های کنترلی را به حالت دیفالت میبریم تا اگر استیت عوض شد ، سیگنالهایی که آن استیت ارتباطی ندارد با آنها به حالت دیفالت باشند.

یک پراسس هم بر روی سیگنالهای کلاک و ریست ایجاد میکنیم. اگر ریست 1 باشد ، استیت کنونی (currentState) را برابر با reset میکنیم.

حالا به پیاده سازی استیت ماشین توسط case-when میپردازیم.

در هر case باید کار مربوطه انجام شود و همچنین nextState مشخص شود.

استیت halt :

هیچ کاری انجام نمیشود. و استیت بعدی هم خودش است. (بنابراین در این استیت خواهد ماند و کاری انجام نمیدهد تا وقتی که externalReset فعال شود و سیستم را به حالت reset ببرد.

استیت reset :

سیگنالهای مربوط به ریست شدن سیستم را فعال میکنیم :

یعنی :

CReset, ZReset, WPreset, ResetPC, EnablePC

و nextState هم برابر fetch خواهد بود. (البته توجه شود که وقتی سیگنال ریست فعال شد تا وقتی که غیرفعال نشود در پروسس اول که مرتبط با کلاک بود ، استیت به همان استیت ریست بازمیگردد و اگر سیگنال ریست غیرفعال گردد آنگاه nextState که همان fetch است برقرار میشود)

استیت fetch :

در این استیت باید از مموری به IR انتقال داد. اگر دیتا آماده باشد استیت بعدی ، استیت baseExe است برای احرای دستور. و اگر هنوز اماده نباشد انقدر در استیت fetch میماند تا اماده شود. (در این استیت readMem باید 1 باشد)

استیت PCInc :

دستور spc (save PC) به این صورت است که خروجی addressLogic میشود PC+I که بتوان آنرا از طریق دیتاباس به رجیسترفایل لود کرد. اما در ادامه آن دستور باید عمل fetch انجام شود پس باید قبل از استیت fetch ، PC را یک عدد اضافه کرد پس خروجی addressLogic باید بشود PC+1 پس نمیتوان در یک کلاک این کار را انجام داد. پس استیت PCInc را برای اینکار در نظر میگیریم تا پس از اجرای دستور spc به این استیت رفته و سپس به استیت fetch برویم.

استیت baseExe :

این استیت برای اجرای دستورات 16بیتی و یا دستورات 8 بیتی که در بخش پر ارزش IR هستند، میباشد.

برای دستورات 8 بیتی استیت بعدی استیت shadowExe و برای 16بیتی ها چون باید دستور بعدی را بگیرد استیت fetch میباشد.

ابتدا برای دستوراتی که در جدول دستورات، در نظر گرفته نشده اند کدی را انتخاب میکنیم :

XOR : (xor) -> (must choose Rs and Rd) -> ??

Two’s Complement : (tcm) -> (must choose Rs and Rd) -> 0100-D-S (instead of “inp” instruction in table1)

Random : (rnd) -> (must choose just Rd) -> 0101-D-XX (instead of “oup” instruction in table1)

سپس بوسیله case-when به روی بیتهای IR دستور ر تشخیص میدهیم و در هر مورد بیتهای کنترلی مرتبط را فعال میکنیم و nextState را هم مشخص میکنیم.

** وقتی قرار است به استیت fetch برویم قبلش باید PC را یک عدد زیاد کنیم (مگر حالاتی که خود دستور PC را تغیر میدهد)

برای راحتی کار یک سری کانستنت برای دستورات تعریف میکنیم :

به این صورت که دستورات را به 4 دسته تقسیم میکنیم :

به این صورت که برای دستوراتی که با 0000 شروع میشوند ، پارت دومشان را کانستنت میگیریم(P2) (که خود اینها در واقع دو دسته 16بیتی و 8بیتی هستند) و بقیه 8 بیتی ها را پارت اولشان را(P1).
برای 4دستور 16 بیتی ای که با 1111 شروع میشوند هم بیتهای 9 و 8 را کانستنت میگیریم(P22).

دستورات مختلف :

Nop : هیچ کاری انحام نمیدهد.صرفا استیت بعدی را انتخاب میکنیم.

8بیتی

Hlt : استیت بعدی استیت halt میباشد یعنی خودش و هیچ کار دیگری انجام نمیدهیم.

8بیتی

Szf : سگنال set برای z flag را روشن میکنیم.

8بیتی

Czf : سیگنال reset برای z flag را روشن میکنیم.

8بیتی

Scf : سیگنال set را برای c flag روشن میکنیم.

8بیتی

Ccf : سیگنال reset را برای c flag روشن میکنیم.

8بیتی

Cwp : سیگنال reset برای WP را روشن میکنیم.

8بیتی

Jpr : بوسیله روشن کردن سیگنال PCplusI آن مقدار را در PC لود میکنیم.

16بیتی

Brz : اگر z flag یک باشد کار Jpr را انجام میدهد.

16بیتی

Brc : اگر c flag یک باشد کار Jpr را انجام میدهد.

16بیتی

Awp : سیگنال WPadd را روشن میکنیم. و خود به خود 6 بیت از IR را میگیرد از datapath و آنرا به WP اضافه میکند.

16بیتی

Mil : بوسیله یک کردن readMem اطلاعات خانه مورد نظر مموری را به databus انتقال میدهیم و بوسیله سیگنال لود آنرا در رجیسترفایل و در Rd لود میکنیم.

16بیتی

Mih : همان کار Mil را انجام میدهد با این تفاوت که سیگنال لود دیگر روشن میشود تا این بار در 8بیت پر ارزش عمل لود انجام بگیرد.

16بیتی

Spc : سیگنال PCplusI روشن میکنیم تا PC+I روی addressBus قرار گیرد. سپس addressBus را به dataBusوصل میکنیم (توسط Address_on_DataBus) و با روشن کردن جفت سیگنالهای لود رجیسترفایل آنرا روی رجیستر Rd ذخیره میکنم.

16بیتی

Jpa : RplusI را روشن کرده تا Rside+I در addressBus بیاید و با روشن کردن enable رجیستر PC آن مقدار را لود میکنیم در PC.

16بیتی

Mvr : بوسیله B15to0 و ALUout_on_DataBus مقدار Rs را روی dataBus قرار میدهیم. سپس با روشن کرد سیگنال لود رجیسترفایل، مقدار آن روی Rd ذخیره میشود.

8بیتی

Lda : Rs را روی addressBus قرار میدهیم . با روشن کردن Rplus0 آدرس مرتبط با مموری را وصل میکنیم که همان Rs است. سپس readMem را روشن میکنیم تا مقدار آن خانه را بخانیم و سپس با روشن کردن سیگنالهای لود آنرا روش رجیسترRd لود میکنیم.

8بیتی

Sta : مانند lda با این تفاوت که writeMem را روشن میکنیم و مقدار Rs را روی dataBus میریزیم.

8بیتی

برای تمام دستورات مربوط به ALU فقط کافیست بیت کنترلی مربوط به آن عمل را روشن کنیم و خروجی ALU را هم به dataBus وصل کنیم (توسط ALUout_on_DataBus) (البته به جز دستور cmp که خروجیalu ندارد)

همچنین برای دستورات ریاضی و دستور cmp باید که SRload را روشن کنیم تا c و z را بتوانند تغیر دهند.

8بیتی

استیت shadowExe:

این استیت هم مانند استیت baseExe برای decode کردن دستور و اجرای آن میباشد. با این تفاوت که در این استیت دیگر نمیتواند دستور 16بیتی باشد . و تفاوت دیگر در آن است که در استیت baseExe اگر دستور 8بیتی میبود ، استیت بعدیش میشد shadowExe ولی در اینجا استیت بعدی همه آنها fetch میباشد.

البته همان ابتدای این استیت

```
shadow <= 1
```

را قرار میدهیم تا بوسیله آن بفهمیم در حال خواندن دستوری در قسمت shadow هستیم یا خیر.

استیت PCInc :

دستورات MIL, MIH, SPC, LDA, STA به این صورت میباشند که با addressBus ارتباط دارند و مقداری در addressBus موجود است . بنابراین هم زمان با اجرای این دستورات نمیتوان PC را یکدانه اضافه کنیم پس ابتدا دستور را اجرا میکنیم و سپس با رفتن به استیت PCInc در آنجا PC را یکی زیاد کرده و سپس به استیت بعدی که همان fetch است میرویم.

کنترلر CU Plus :

با دستکاری کردن برخی کدهای دستورات از پیش تعیین شده میتوانیم دستورات بیشتری جای دهیم . تغیراتی که در دستورات از پیش تعین شده ایجاد شد :

CZF = 0000-11-00

JPR = 0000-11-01-I

و دستور رندم را هم به این صورت تعریف میکنیم :

RND = 0000-D-11

دلیل این کار و تغیر کد دستورهای از قبل تعین شده این است که با این تغیرات یک حالت منحصر به فرد ایجاد میشود : حالتی که 4بیت پر ارزش 0 هستند و دو بیت آخر (در دستور 8 بیتی) 1 باشند. و با یک شرط میبینیم اگر اینگونه است پس دستور رندم است وگرنه باید وارد case بندی ها بشود.

پس دستورات در CU_Plus به این صورت هستند :

Instruction Mnemonic and Definition		Bits 15:0
nop	No operation	0000-00-00
hlt	Halt	0000-00-01
szf	Set zero flag	0000-00-10
czf	Clr zero flag	0000-11-00
scf	Set carry flag	0000-01-00
ccf	Clr carry flag	0000-01-01
cwp	Clr Window pointer	0000-01-10
mvr	Move Register	0001-D-S
lda	Load Addressed	0010-D-S
sta	Store Addressed	0011-D-S
tcm	Two's Complemet	0100-D-S
xor	XOR Registers	0101-D-S
and	AND Registers	0110-D-S
orr	OR Registers	0111-D-S
not	NOT Register	1000-D-S
shl	Shift Left	1001-D-S
shr	Shift Right	1010-D-S
add	Add Registers	1011-D-S
sub	Subtract Registers	1100-D-S
mul	Multiply Registers	1101-D-S
cmp	Compare	1110-D-S
mil	Move Immd Low	1111-D-00-I
mih	Move Immd High	1111-D-01-I
spc	Save PC	1111-D-10-I
jpa	Jump Addressed	1111-D-11-I
jpr	Jump Relative	0000-11-01-I
brz	Branch if Zero	0000-10-00-I
brc	Branch if Carry	0000-10-01-I
awp	Add Win pntr	0000-10-10-I
rnd	Generate Random number	0000-D-11

؟ : چرا بیتهای 2 و 3 (از آخر) را برای منحصر به فرد گرفن در نظر نگرفتیم ، با اینکه در دستورات پیش فرض حتی، یک حالت خاص میبود؟

زیرا برای دستور رندم ایندو بیت نشاندهنده D هستند پس نمیتوانند ثابت باشند و برای تشخیص دستور استفاده شوند.