تکلیف اول

1. تفاوت معماری وان نیومن و هاروارد

معماری وان نیومن

- **حافظه مشترک: برنامه‌ها و داده‌ها در یک حافظه مشترک ذخیره می‌شوند.**
- **مسیر داده مشترک: یک مسیر داده برای انتقال دستورالعمل‌ها و داده‌ها استفاده می‌شود.**
- **سادگی طراحی: طراحی ساده‌تر و ارزان‌تر است.**

مزایا

- **سادگی و هزینه کمتر: به دلیل استفاده از یک حافظه و مسیر داده مشترک، طراحی و پیاده‌سازی آن ساده‌تر و ارزان‌تر است.**
- **انعطاف‌پذیری: می‌تواند برای انواع مختلف برنامه‌ها استفاده شود.**

معماری هاروارد

- **حافظه جداگانه: برنامه‌ها و داده‌ها در حافظه‌های جداگانه ذخیره می‌شوند.**
- **مسیر داده جداگانه: مسیرهای داده جداگانه برای انتقال دستورالعمل‌ها و داده‌ها وجود دارد.**
- **کارایی بالاتر: به دلیل جداسازی حافظه و مسیر داده، کارایی بالاتری دارد.**

مزایا

- **کارایی بالاتر: به دلیل جداسازی حافظه و مسیر داده، می‌تواند به طور همزمان به دستورالعمل‌ها و داده‌ها دسترسی داشته باشد که منجر به کارایی بالاتر می‌شود.**
- **امنیت بیشتر: جداسازی حافظه برنامه و داده‌ها می‌تواند امنیت بیشتری فراهم کند.**

این تفاوت‌ها و مزایا باعث می‌شوند که هر کدام از این معماری‌ها در شرایط و کاربردهای خاصی مناسب‌تر باشند.

2. تفاوت‌های بین میکروکنترلر و میکروپروسسور

تفاوت‌های اصلی بین معماری میکروکنترلر و میکروپروسسور چیست؟

- **میکروکنترلر:**
  - **یکپارچگی بالا: شامل CPU، حافظه (RAM و ROM)، و ورودی/خروجی‌ها (I/O) در یک تراشه واحد است.**
  - **طراحی برای کاربردهای خاص: معمولاً برای کنترل دستگاه‌های خاص و وظایف مشخص طراحی شده‌اند.**
  - **مصرف انرژی کم: به دلیل طراحی بهینه برای وظایف خاص، مصرف انرژی کمتری دارند.**
- **میکروپروسسور:**
  - **انعطاف‌پذیری بالا: شامل CPU است و نیاز به اجزای خارجی مانند حافظه و I/O دارد.**
  - **طراحی برای کاربردهای عمومی: معمولاً برای انجام وظایف عمومی و پردازش‌های پیچیده طراحی شده‌اند.**
  - **مصرف انرژی بیشتر: به دلیل نیاز به اجزای خارجی و پردازش‌های پیچیده، مصرف انرژی بیشتری دارند.**

کاربردهای عملی

- **میکروکنترلر:**
  - **کاربرد: کنترل دستگاه‌های خانگی مانند ماشین لباسشویی.**
  - **توضیح: میکروکنترلرها به دلیل یکپارچگی بالا و مصرف انرژی کم، برای کنترل دستگاه‌های خانگی که نیاز به وظایف مشخص و ساده دارند، مناسب‌تر هستند.**
- **میکروپروسسور:**
  - **کاربرد: کامپیوترهای شخصی.**
  - **توضیح: میکروپروسسورها به دلیل انعطاف‌پذیری بالا و توانایی انجام پردازش‌های پیچیده، برای کامپیوترهای شخصی که نیاز به انجام وظایف متنوع و پیچیده دارند، مناسب‌تر هستند.**

مصرف انرژی و هزینه

- **مصرف انرژی و هزینه کمتر: میکروکنترلرها به دلیل طراحی بهینه برای وظایف خاص، مصرف انرژی و هزینه کمتری دارند.**
- **کاربردهای مهم: این تفاوت‌ها در کاربردهایی که نیاز به مصرف انرژی کم و هزینه پایین دارند، مانند دستگاه‌های قابل حمل و سیستم‌های تعبیه‌شده، اهمیت پیدا می‌کنند.**

3. جدول تبدیل اعداد

Hexadecimal	Binary	Decimal
4FD	10011111101	1277
111100110001	111100110001	3873
8A5D	1000101001011101	35421
2024	10000000100100	8228

4. محاسبه عبارت‌ها

I. 0b1111101 + 0b110010101

جمع دودویی:

ابتدا اعداد را زیر هم قرار می‌دهیم:

```
  110010101
+   1111101
-----------
```

از سمت راست به چپ جمع می‌کنیم:

```
1 + 1 = 10 (و 1 به ستون بعدی 0)
```

```
0 + 0 + 1 = 1
1 + 1 = 10 (و 1 به ستون بعدی 0)
1 + 0 + 1 = 10 (و 1 به ستون بعدی 0)
1 + 1 = 10 (و 1 به ستون بعدی 0)
1 + 0 + 1 = 10 (و 1 به ستون بعدی 0)
1 + 1 = 10 (و 1 به ستون بعدی 0)
1 + 1 = 10 (و 1 به ستون بعدی 0)
```

نتیجه نهایی:

```
  1000010010
```

بنابراین، 0b1000010010 = 0b110010101 + 0b1111101

II. 0x4B4 + 0x5FD

جمع هگزادسیمال:

ابتدا اعداد را زیر هم قرار می‌دهیم:

```
  0x4B4
+ 0x5FD
--------
```

از سمت راست به چپ جمع می‌کنیم:

```
D + 4 = 11 + 4 = 15 (F)
F + B = 15 + 11 = 26 (1A، 1 به ستون بعدی)
5 + 4 + 1 = 10 (A)
```

نتیجه نهایی:

```
  0xAB1
```

بنابراین، 0x4B4 + 0x5FD = 0xAB1

III. 0x2B7 − 0x59E

تفریق هگزادسیمال:

ابتدا اعداد را زیر هم قرار می‌دهیم:

```
  0x2B7
- 0x59E
--------
```

از سمت راست به چپ تفریق می‌کنیم:

```
7 - E (نیاز به قرض گرفتن داریم)
17 - E = 9
B - 9 = 2
2 - 5 (نیاز به قرض گرفتن داریم)
12 - 5 = 7
```

نتیجه نهایی:

```
  -0x2E7
```

بنابراین، 0x2B7 – 0x59E = –0x2E7

IV. 0b1011000011 – 0xF1 (به روش مکمل دو)

تبدیل 0 xF1 به مکمل دو:

ابتدا عدد هگزادسیمال را به دودویی تبدیل می‌کنیم:

```
0xF1 = 0b11110001
```

سپس عدد دودویی را به مکمل دو تبدیل می‌کنیم:

1. معکوس کردن بیت‌ها:

```
0b11110001 -> 0b00001110
```

2. اضافه کردن 1 به نتیجه:

```
0b00001110 + 1 = 0b00001111
```

بنابراین، مکمل دو عدد 0xF1 برابر است با 0b00001111.

انجام عملیات تفریق:

1. انجام عملیات جمع با مکمل دو:

```
  0b1011000011
+ 0b00001111
-------------
```

2. جمع از سمت راست به چپ:

```
 1 + 1 = 10 (و 1 به ستون بعدی 0)
1 + 1 = 10 (و 1 به ستون بعدی 0)
0 + 1 + 1 = 10 (و 1 به ستون بعدی 0)
0 + 1 + 1 = 10 (و 1 به ستون بعدی 0)
0 + 0 + 1 = 1
1 + 0 = 1
0 + 0 = 0
1 + 0 = 1
1 + 0 = 1
```

نتیجه نهایی:

```
  0b110000010
```

بنابراین، 0b1011000011 – 0xF1 = 0b110000010

V. 0x4B8 – 0x1A3 (به روش دلخواه)

تفریق هگزادسیمال:

ابتدا اعداد را زیر هم قرار می‌دهیم:

```
  0x4B8
- 0x1A3
--------
```

از سمت راست به چپ تفریق می‌کنیم:

```
 8 - 3 = 5
B - A = 1
4 - 1 = 3
```

نتیجه نهایی:

```
  0x315
```

بنابراین، 0x4B8 – 0x1A3 = 0x315

5. مراحل طراحی

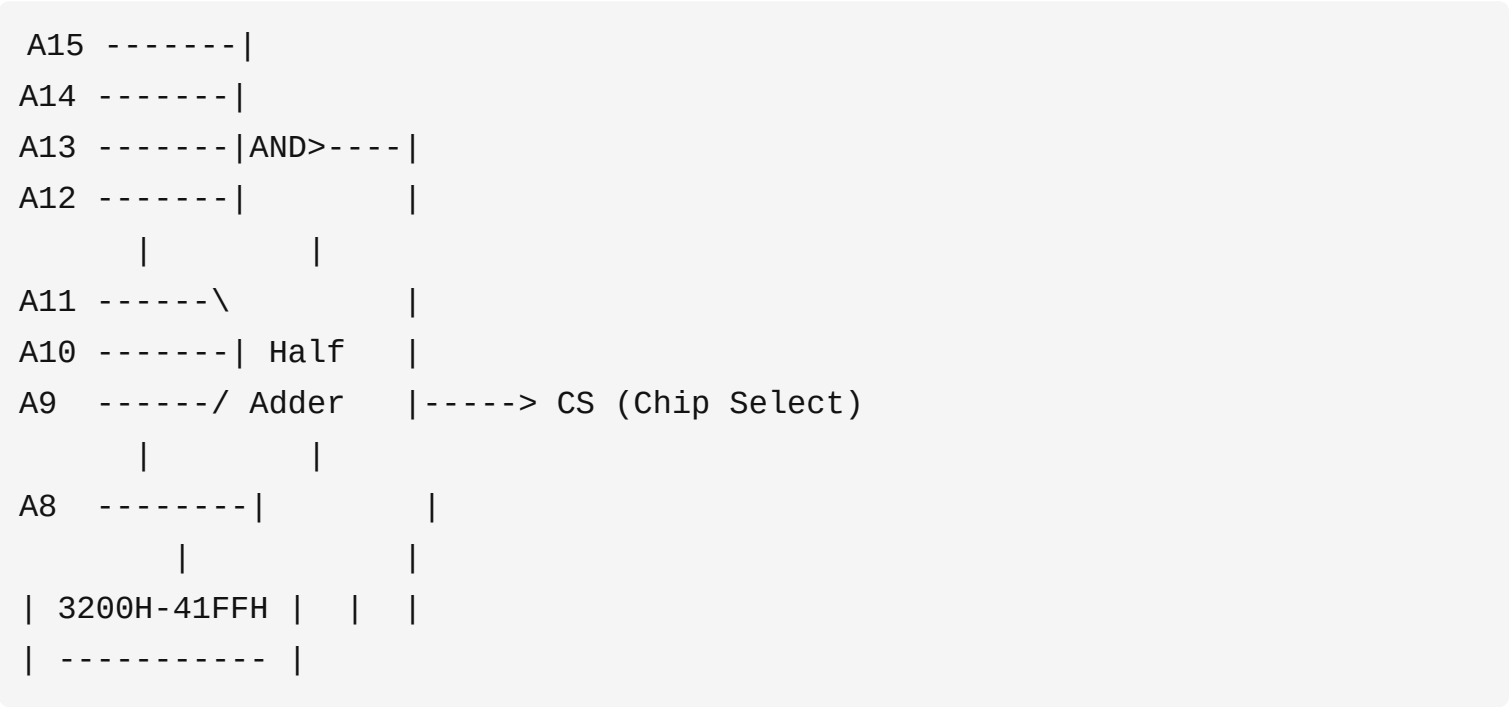
تحلیل محدوده آدرس:

- **محدوده آدرس: 3200H تا 41FFH**
- **ظرفیت حافظه: 4096 = 4K بایت = 1000H بایت**
- **بیت‌های آدرس مورد نیاز: 12 بیت (برای 4K)**

محاسبه خطوط آدرس:

- **آدرس شروع: 0000 0010 0000 0011 = 3200H**
- **آدرس پایان: 1111 1111 0001 0100 = 41FFH**

طراحی مدار:



توضیحات

- **نیاز به بررسی بیت‌های بالای آدرس (A12–A15) برای تشخیص محدوده**
- **استفاده از Half-Adder برای مقایسه بیت‌های میانی**
- **خروجی CS فقط در محدوده آدرس مورد نظر فعال می‌شود**
- **بیت‌های پایین (A0–A11) مستقیماً به حافظه متصل می‌شوند**

6. مقایسه معماری RISC و CISC

معماری RISC (Reduced Instruction Set Computer)

- **تعداد دستورات کمتر: مجموعه دستورات کوچک و ساده.**
- **اجرای سریع‌تر: هر دستورالعمل معمولاً در یک سیکل ساعت اجرا می‌شود.**
- **طراحی ساده‌تر: طراحی ساده‌تر و بهینه‌تر برای پردازش سریع.**
- **استفاده بیشتر از حافظه: به دلیل تعداد دستورات کمتر، برنامه‌ها ممکن است طولانی‌تر باشند و حافظه بیشتری مصرف کنند.**

- مثال‌ها: ARM، MIPS

معماری CISC (Complex Instruction Set Computer)

- تعداد دستورات بیشتر: مجموعه دستورات بزرگ و پیچیده.
- اجرای کندتر: برخی از دستورالعمل‌ها ممکن است چندین سیکل ساعت برای اجرا نیاز داشته باشند.
- طراحی پیچیده‌تر: طراحی پیچیده‌تر و نیاز به مدارهای کنترلی بیشتر.
- استفاده کمتر از حافظه: به دلیل تعداد دستورات بیشتر، برنامه‌ها معمولاً کوتاه‌تر هستند و حافظه کمتری مصرف می‌کنند.
- مثال‌ها: x86، VAX

مقایسه و انتخاب

- کارایی: معماری RISC به دلیل اجرای سریع‌تر دستورالعمل‌ها، در کاربردهایی که نیاز به پردازش سریع دارند، مناسب‌تر است.
- پیچیدگی: معماری CISC به دلیل پیچیدگی بیشتر، در کاربردهایی که نیاز به دستورات پیچیده و متنوع دارند، مناسب‌تر است.
- مصرف انرژی: معماری RISC به دلیل طراحی ساده‌تر و اجرای سریع‌تر، مصرف انرژی کمتری دارد و برای دستگاه‌های قابل حمل مناسب‌تر است.
- انعطاف‌پذیری: معماری CISC به دلیل مجموعه دستورات بزرگ‌تر، انعطاف‌پذیری بیشتری دارد و برای کاربردهای عمومی مناسب‌تر است.

نتیجه‌گیری

انتخاب بین RISC و CISC بستگی به نیازهای خاص کاربرد دارد. اگر نیاز به پردازش سریع و مصرف انرژی کم دارید، معماری RISC مناسب‌تر است. اگر نیاز به انعطاف‌پذیری و دستورات پیچیده دارید، معماری CISC مناسب‌تر است.

7. برنامه به زبان اسمبلی برای تبدیل مقدار 0xfd به دسیمال و ذخیره ارقام در آدرس‌های مشخص شده

```
ORG 0x1000
LDA 0x315
MOV B, A
```

- `ORG 0x1000` : تعیین آدرس شروع برنامه در حافظه.
- `LDA 0x315` : بارگذاری مقدار موجود در آدرس 0x315 به داخل رجیستر A.
- `MOV B, A` : کپی کردن مقدار رجیستر A به رجیستر B.

```
MVI C, 0x0A
CALL DIVIDE
STA 0x322
```

- `MVI C, 0x0A` : مقدار 10 را به رجیستر C منتقل می‌کند.
- `CALL DIVIDE` : فراخوانی زیرروال تقسیم برای تقسیم مقدار در رجیستر A بر 10.
- `STA 0x322` : ذخیره نتیجه تقسیم در آدرس 0x322.

```
MOV A, B
```

```
  CALL DIVIDE
  STA 0x323
```

- `MOV A, B` : **کپی کردن مقدار رجیستر B به رجیستر A.**
- `CALL DIVIDE` : **فراخوانی زیرروال تقسیم برای تقسیم مقدار در رجیستر A بر 10.**
- `STA 0x323` : **ذخیره نتیجه تقسیم در آدرس 0x323.**

```
  MOV A, B
  STA 0x324
```

- `MOV A, B` : **کپی کردن مقدار رجیستر B به رجیستر A.**
- `STA 0x324` : **ذخیره مقدار رجیستر A در آدرس 0x324.**

پایان برنامه:

```
  HLT
```

- `HLT` : **توقف اجرای برنامه.**

زیرروال تقسیم:

```
DIVIDE:
  MOV D, A
  MVI A, 0x00
DIV_LOOP:
  CMP C
  JC DIV_END
  SUB C
  INR A
  JMP DIV_LOOP
DIV_END:
  MOV B, D
  RET
```

- `DIVIDE:` : **شروع زیرروال تقسیم.**
- `MOV D, A` : **کپی کردن مقدار رجیستر A به رجیستر D.**
- `MVI A, 0x00` : **مقداردهی رجیستر A به صفر.**
- `DIV_LOOP:` : **شروع حلقه تقسیم.**
- `CMP C` : **مقایسه مقدار رجیستر A با رجیستر C.**
- `JC DIV_END` : **اگر مقدار A کمتر از C باشد، به** `DIV_END` **بپر.**
- `SUB C` : **کم کردن مقدار C از A.**
- `INR A` : **افزایش مقدار A به اندازه 1.**

- `JMP DIV_LOOP` : بازگشت به ابتدای حلقه.
- `DIV_END:` : پایان حلقه تقسیم.
- `MOV B, D` : بازگرداندن مقدار اصلی **A** به رجیستر **B**.
- `RET` : بازگشت از زیرروال.

این برنامه مقدار 0xfd را از آدرس 0x315 بارگذاری کرده و آن را به دسیمال تبدیل می‌کند. سپس ارقام دسیمال را در آدرس‌های ،0x322 0x323 و 0x324 ذخیره می‌کند. رقم کم ارزش در آدرس 0x322، رقم میانی در آدرس 0x323 و رقم با ارزش بالا در آدرس 0x324 ذخیره می‌شود.

8. برنامه به زبان اسمبلی برای محاسبه فاکتوریل یک عدد صحیح بین 0 تا 5

```
 ORG 0x1000
LDA 0x400
CALL FACTORIAL
STA 0x401
HLT
```

- `ORG 0x1000` : شروع برنامه از آدرس **0x1000**.
- `LDA 0x400` : بارگذاری مقدار از آدرس **0x400** به رجیستر **A**.
- `CALL FACTORIAL` : فراخوانی زیرروال محاسبه فاکتوریل.
- `STA 0x401` : ذخیره نتیجه در آدرس **0x401**.
- `HLT` : توقف اجرای برنامه.

```
FACTORIAL:
  MOV B, A
  MVI A, 0x01
  MOV C, A
```

- `MOV B, A` : کپی مقدار رجیستر **A** به رجیستر **B**.
- `MVI A, 0x01` : مقدار **1** را به رجیستر **A** منتقل می‌کند.
- `MOV C, A` : کپی مقدار رجیستر **A** به رجیستر **C**.

```
CHECK_ZERO:
  CMP B
  JZ RETURN_ONE
```

- `CMP B` : مقایسه مقدار رجیستر **A** با رجیستر **B**.
- `JZ RETURN_ONE` : اگر مقدار **B** صفر باشد، به برچسب **RETURN_ONE** پرش می‌کند.

```
CALC_FACTORIAL:
  CMP B
  JZ END_FACTORIAL
```

```
  MOV D, B
  DEC D
  MOV B, D
  MOV D, C
  MOV A, D
  MOV E, B
  CALL MULTIPLY
  MOV C, A
  JMP CALC_FACTORIAL
```

- `CMP B` : **مقایسه مقدار رجیستر A با رجیستر B.**
- `JZ END_FACTORIAL` : **اگر مقدار B صفر باشد، به برچسب END_FACTORIAL پرش می‌کند.**
- `MOV D, B` : **کپی مقدار رجیستر B به رجیستر D.**
- `DEC D` : **کاهش مقدار رجیستر D به اندازه 1.**
- `MOV B, D` : **کپی مقدار رجیستر D به رجیستر B.**
- `MOV D, C` : **کپی مقدار رجیستر C به رجیستر D.**
- `MOV A, D` : **کپی مقدار رجیستر D به رجیستر A.**
- `MOV E, B` : **کپی مقدار رجیستر B به رجیستر E.**
- `CALL MULTIPLY` : **فراخوانی زیرروال ضرب.**
- `MOV C, A` : **کپی مقدار رجیستر A به رجیستر C.**
- `JMP CALC_FACTORIAL` : **پرش به برچسب CALC_FACTORIAL برای تکرار محاسبات.**

```
END_FACTORIAL:
  MOV A, C
  RET
```

- `MOV A, C` : **کپی مقدار رجیستر C به رجیستر A.**
- `RET` : **بازگشت از زیرروال.**

```
RETURN_ONE:
  MVI A, 0x01
  RET
```

- `MVI A, 0x01` : **مقدار 1 را به رجیستر A منتقل می‌کند.**
- `RET` : **بازگشت از زیرروال.**

```
MULTIPLY:
  MVI H, 0x00
  MVI L, 0x00
  MOV D, A
  MOV E, B
  CALL MUL_LOOP
```

```
  MOV A, L
  RET
```

- `MVI H, 0x00` : **مقدار 0 را به رجیستر H منتقل می‌کند.**
- `MVI L, 0x00` : **مقدار 0 را به رجیستر L منتقل می‌کند.**
- `MOV D, A` : **کپی مقدار رجیستر A به رجیستر D.**
- `MOV E, B` : **کپی مقدار رجیستر B به رجیستر E.**
- `CALL MUL_LOOP` : **فراخوانی زیرروال حلقه ضرب.**
- `MOV A, L` : **کپی مقدار رجیستر L به رجیستر A.**
- `RET` : **بازگشت از زیرروال.**

```
MUL_LOOP:
  MOV A, D
  ADD L
  MOV L, A
  MOV A, H
  ADC H
  MOV H, A
  DEC E
  JNZ MUL_LOOP
  RET
```

- `MOV A, D` : **کپی مقدار رجیستر D به رجیستر A.**
- `ADD L` : **جمع مقدار رجیستر L با رجیستر A.**
- `MOV L, A` : **کپی مقدار رجیستر A به رجیستر L.**
- `MOV A, H` : **کپی مقدار رجیستر H به رجیستر A.**
- `ADC H` : **جمع مقدار رجیستر H با رجیستر A به همراه بیت حمل.**
- `MOV H, A` : **کپی مقدار رجیستر A به رجیستر H.**
- `DEC E` : **کاهش مقدار رجیستر E به اندازه 1.**
- `JNZ MUL_LOOP` : **اگر مقدار E صفر نباشد، به برچسب MUL_LOOP پرش می‌کند.**
- `RET` : **بازگشت از زیرروال.**

این برنامه مقدار ورودی را از آدرس 0x400 می‌خواند و فاکتوریل آن را محاسبه می‌کند. نتیجه در آدرس 0x401 ذخیره می‌شود. اگر عدد ورودی صفر باشد، تابع مقدار 1 را برمی‌گرداند.