Abstract:

مسیر داده و واحد کنترل‌کننده در امتحان میان‌ترم دوم طراحی شده‌اند و در ابتدا قالب دستورات پردازنده آورده شده است.
برای تست پردازنده مذکور برنامه‌ای به زبان اسمبلی نوشته شده است که مقدار و اندیس بزرگ‌ترین عضو یک آرایه 20 عنصری را پیدا می‌کند و آن‌ها را در مموری ذخیره می‌کند. همچنین برای تبدیل برنامه اسمبلی ذکر شده به زبان ماشین، یک برنامه اسمبلر به زبان C++ نوشته شده که در فولدر Utils قرار داده شده است.
برای قرار دادن مقادیر آرایه در مموری از یک فایل به نام ArrayData.txt استفاده می‌شود که در فولدر Memory قرار دارد. در این فایل مقادیر به صورت Decimal و در 20 خط متوالی نوشته می‌شوند. این مقادیر در ادامه به کمک یک اسکریپت پایتون به مقادیر باینری 16 بیتی تبدیل می‌شوند.
فایل FindMax.asm که کد اسمبلی برنامه تست پردازنده است توسط اسکریپت AssemblyGenerator.py که در فولدر Memory قرار دارد تولید می‌شود. (چون همانطور که جلوتر در بخش Test Program توضیح داده میشود، باید به ازای همه اعضای آرایه کد اسمبلی کپی شود)
در نهایت با اجرای اسکریپت GenerateMemFiles.bat که در همین فولدر Memory قرار دارد، کد اسمبلی به اسمبلر ذکر شده داده می‌شود و خروجی آن در فایل FindMax.machine ذخیره می‌شود. این فایل حاوی تعدادی دستور 16 بیتی به زبان ماشین است که هر کدام در یک خط قرار گرفته‌اند.
در نهایت هر دو فایل FindMax.machine و ArrayData.txt به اسکریپت‌های پایتون موجود در فولدر Utils داده می‌شوند که در فرمت mem. قرار گیرند. فایل نهایی data.mem ایجاد شده به وسیله اسکریپت اجرا شده در فولدر Verilog\Sim که محل ایجاد پروژه ModelSim است قرار می‌گیرد تا با اجرای شبیه‌سازی، توسط مموری خوانده شوند.
در ادامه به کمک شبیه‌سازی، درستی عملکرد پردازنده نشان داده می‌شود.

پاشا براهیمی – 810199385
میثاق محقق – 810199484

CPU Instructions:

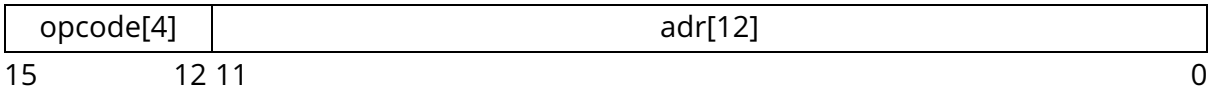
A-Type:

Adr is an immediate value. opcode:

load Adr	R0 = Mem[Adr]	0000
store Adr	Mem[Adr] = R0	0001
jump Adr	PC = Adr	0010

Assembly: *inst adr*

Machine:



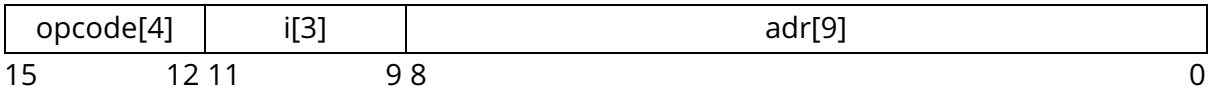
B-Type:

Adr is an immediate value. opcode:

branch Ri, Adr	if Ri == R0: PC[8:0] = Adr	0100

Assembly: *inst adr*

Machine:



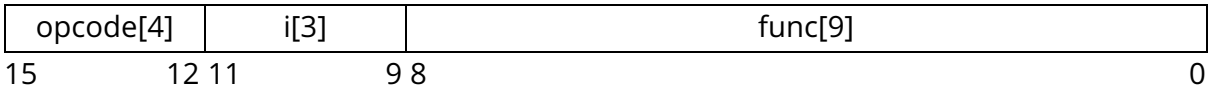
C-Type:

The opcode for all C-Type instructions is *1000* func:

add Ri	R0 = R0 + Ri	000000100
sub Ri	R0 = R0 - Ri	000001000
and Ri	R0 = R0 & Ri	000010000
or Ri	R0 = R0 | Ri	000100000
not Ri	R0 = ~R0	001000000
nop	No Operation	010000000
mvto Ri	R0 = Ri	000000001
mvfrom Ri	Ri = R0	000000010

Assembly: *inst Ri*

Machine:



پاشا براهیمی – 810199385
میثاق محقق – 810199484

D-Type:

Num is an immediate value. opcode:

addi Num	R0 = R0 + Num	1100
subi Num	R0 = R0 - Num	1101
andi Num	R0 = R0 & Num	1110
ori Num	R0 = R0 | Num	1111

Assembly: *inst imm*

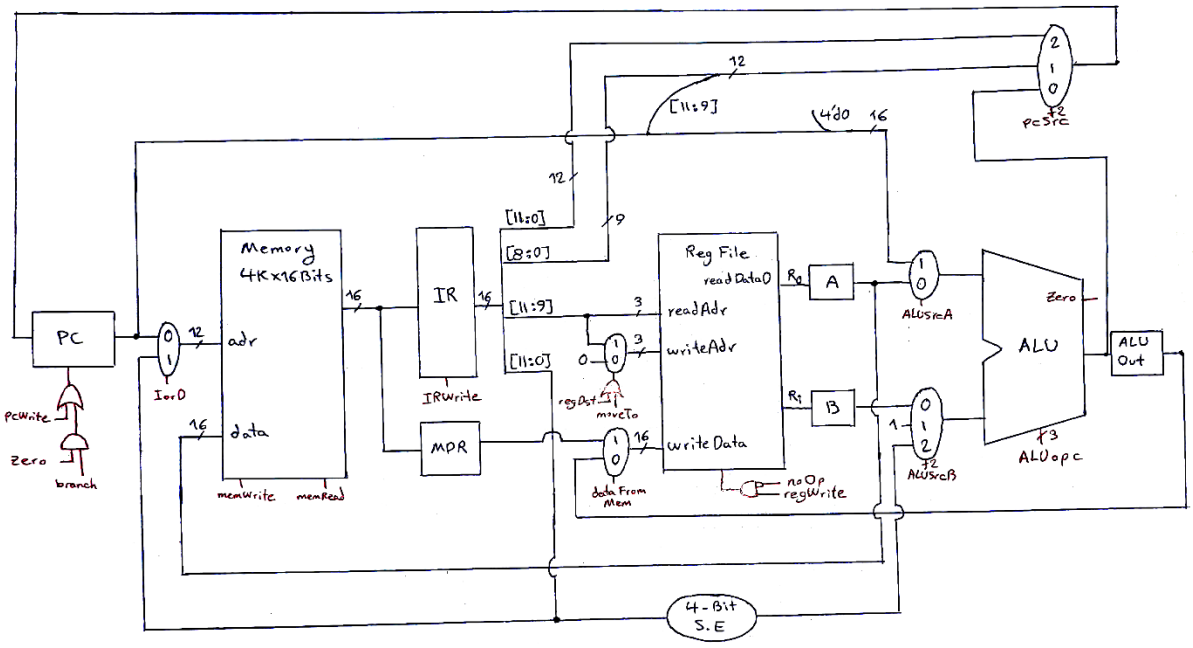
Machine:

opcode[4]	imm[12]

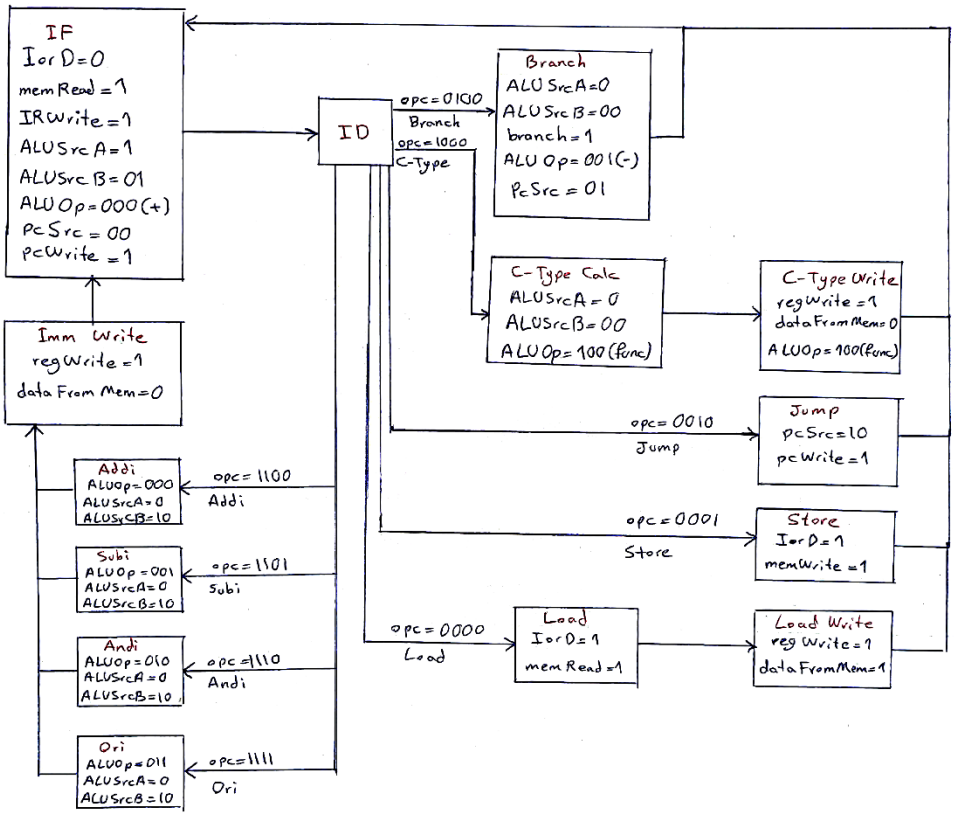
15 12 11 0

CPU:

Datapath:



Controller:



جدول زیر مقادیر سیگنال های خروجی واحد کنترل‌کننده را در هر state نشان می‌دهد:

	memWrite	memRead	IorD	IRWrite	regDst	dataFromMem	regWrite	ALUSrcA	ALUSrcB	ALUop	PcSrc	PcWrite	branch
Instruction Fetch (IF)	0	1	0	1	—	—	0	1 (PC)	01 ('d1)	000 (+)	00 (ALU)	1	0
Instruction Decode (ID)	0	0	—	0	—	—	0	—	—	—	—	0	0
Branch	0	0	—	0	—	—	0	0 (R0)	00 (Ri)	001 (-)	01 (Branch)	—	1
C-Type Calc	0	0	—	0	—	—	0	0 (R0)	00 (Ri)	100 (func)	—	0	0
C-Type Write	0	0	—	0	0 (R0)	0	1	—	—	100 (func)	—	0	0
Jump	0	0	—	0	—	—	0	—	—	—	10 (Jump)	1	0
Store	1	0	1	0	—	—	0	—	—	—	—	0	0
Load	0	1	1	0	—	—	0	—	—	—	—	0	0
Load Write	0	0	—	0	0 (R0)	1	1	—	—	—	—	0	0
Addi	0	0	—	0	—	—	0	0 (R0)	10 (imm)	000 (+)	—	0	0
Subi	0	0	—	0	—	—	0	0 (R0)	10 (imm)	001 (-)	—	0	0
Andi	0	0	—	0	—	—	0	0 (R0)	10 (imm)	010 (&)	—	0	0
Ori	0	0	—	0	—	—	0	0 (R0)	10 (imm)	011 (|)	—	0	0
Imm Write	0	0	—	0	0 (R0)	0	1	—	—	—	—	0	0

جدول طراحی ALU:

ALU Operations	Opcode	Output
+	000	A + B
-	001	A - B
AND	010	A & B
OR	011	A | B
NOT	100	~B
PassInput1	101	A
PassInput2	110	B

جدول خروجی کنترلر ALU (به ALU):

ALU Controller Function	ALUopc	noOp	moveTo
000000100	000 (+)	0	0
000001000	001 (-)	0	0
000010000	010 (&)	0	0
000100000	011 (|)	0	0
001000000	100 (~)	0	0
000000001	101 (Pass1 R0)	0	1
000000010	110 (Pass2 Ri)	0	0
010000000	101 (Pass)	1	0

جدول ورودی کنترلر ALU (از کنترلر اصلی):

CPU To ALU Controller	ALUop
+	000
-	001
AND	010
OR	011
Function	100

Test Program:

Assembly:

از آنجا که این CPU دستوری برای دسترسی به حافظه با مقدار رجیستر ندارد و فقط مقادیر ثابت را قبول می‌کند، نمیتوانیم حلقه for بزنیم و باید کد اسمبلی برای چرخه را 20 بار (با تغییر آدرس حافظه) تکرار کنیم.
این کار تعداد دستور ها را افزایش می‌دهد و از 200 دستور بیشتر خواهد شد. پس شروع عناصر آرایه را به خانه 300 و مقدار بزرگترین عنصر و اندیس آن را به ترتیب در خانه‌های 400 و 404 حافظه قرار می‌دهیم.
کد اسمبلی برای پیدا کردن بزرگ‌ترین عضو آرایه‌ای 20 عنصری:

```
andi 0           # R0 = 0
mvto R7          # R7 = 0 (const)
addi 1024        # R0 = 1024
add R0           # R0 = 2048
add R0           # R0 = 4096
add R0           # R0 = 8192
add R0           # R0 = 16384
add R0           # R0 = 32768
mvto R6          # R6 = 0x8000 (const)

load 300         # R0 = M[300]
mvto R1          # R1 = M[300] (max)
andi 0           # R0 = 0
mvto R2          # R2 = 0 (maxIdx)

# start loop

load 301         # R0 = M[301]
mvto R4          # R4 = M[301] (temp)
sub R1           # R0 = temp - max
and R6           # R0 & 0x8000 (keep MSB)
branch R7, gt01  # R0 == 0
jump end01

gt01:
    mvfrom R4    # R0 = temp
    mvto R1      # max = temp
    andi 0       # R0 = 0
    addi 01      # R0 = 1
    mvto R2      # R2 = 1 (maxIdx)

end01:

# end loop

... # repeat (start-end) 19 times


mvfrom R1        # R0 = max
store 400        # M[400] = max
mvfrom R2        # R0 = maxIdx
store 404        # M[404] = maxIdx
```

توضیح مقایسه کردن دو مقدار در اسمبلی:
برای اینکه دو عنصر را مقایسه کنیم، از آنجا که دستوری برای این کار نداریم، ابتدا دو عنصر را از هم کم می‌کنیم، و سپس از MSB برای تشخیص منفی بودن حاصل تفریق استفاده می‌کنیم. از اینجا متوجه می‌شویم که عدد بزرگ‌تر است یا خیر.
در ابتدای اسمبلی عدد 32768 که در باینری معادل 1000000000000000 است (1 با 15 تا 0) را ذخیره می‌کنیم. از این عدد برای mask کردن MSB استفاده می‌کنیم.
برای مقایسه، ابتدا دو عدد را از هم کم می‌کنیم (a-b)، سپس با عدد بالا and می‌کنیم، و در آخر با عدد 0 مقایسه می‌کنیم (این کار با دستور branch ممکن است). اگر عدد 0 بود، یعنی a-b مثبت بوده و در نتیجه a > b است.

Machine Code:

با توجه به معادل‌های اسمبلی و کدهای 16 بیتی قابل اجرا در بخش Instructions، یک برنامه assembler نوشته شد که بخشی از خروجی آن برای کد بالا در تصویر زیر مشخص شده است:

```
1111000000000000
1000111000000001
1100010000000000
1000000000000100
1000000000000100
1000000000000100
1000000000000100
1000000000000100
1000110000000001
0000000100101100
1000001000000001
1110000000000000
1000010000000001
0000000100101101
1000100000000001
1000001000001000
1000110000010000
0100111000010011
0010000000011000
1000100000000010
1000001000000001
1110000000000000
1100000000000001
1000010000000001
0000000100101110
1000100000000001
1000001000001000
```

Simulation Results:

Memory > ArrayData.txt

```
1   -34
2   -10
3   37
4   46
5   98
6   2
7   131
8   -982
9   143
10  8
11  56
12  36
13  -28
14  98
15  17
16  -7
17  0
18  2
19  5
20  91
```

Memory > data.mem

```
1   // Array Data
2
3   @12c
4   1111111111011110
5   1111111111110110
6   0000000000100101
7   0000000000101110
8   0000000001100010
9   0000000000000010
10  0000000010000011
11  1111110000101010
12  0000000010001111
13  0000000000001000
14  0000000000111000
15  0000000000100100
16  1111111111100100
17  0000000001100010
18  0000000000010001
19  1111111111111001
20  0000000000000000
21  0000000000000010
22  0000000000000101
23  0000000001011011
24
25
26  // Instructions
27
28  @000
29  1110000000000000
30  1000111000000001
31  1100010000000000
```

این عناصر خانه 300 تا 319 حافظه را اشغال می‌کنند.
برای خواندن این مقادیر از فرمت mem. استفاده شده است.
ابتدا مقادیر آرایه از خانه 300 نوشته شده‌اند و سپس دستورها از خانه 0 شروع می‌شوند.

نتیجه شبیه‌سازی:

Max Element خانه 400 مموری است که مقدار 143 را نشان می‌دهد.
همچنین Max Index خانه 404 مموری است که مقدار 8 را نشان می‌دهد.
با توجه به مقادیر Decimal آرایه که بالاتر نشان داده شد، عملکرد پردازنده صحیح است.
در طی محاسبات، R1 عنصر بیشینه و R2 هم اندیس آن را نگه می‌دارد.