به نام خدا

معماری کامپیوتر

پروژه پایانی ـ بخش اول ـ طراحی پردازنده

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

استاد:

جناب آقای دکتر اسدی

نام و نام خانوادگی اعضای گروه:

سپهر پورقناد ـ ۹۷۱۰۱۳۵۹

سیدمحمدصادق کشاورزی ـ ۹۷۱۰۶۲۴۹

امیرمهدی نامجو ـ ۹۷۱۰۷۲۱۲

۱ مقدمه

هدف اصلی ما در این پروژه، طراحی یک پردازنده ساده براساس دستورات داده شده در مستندات آن و تست کردن آن به شکلی بوده است که حالات مختلف آن پوشش داده شود. در طراحی پردازنده داده شده، ما از اصول کلی که در درس آموخته ایم و عموما براساس پردازنده مبتنی بر MIPS بوده اند، استفاده کرده ایم. البته در این جا ساختار کلی ما از لحاظ کلی ساده تر از MIPS است. در ادامه گزارش، به توضیح نحوه طراحی بخش های مختلف این پردازنده می پردازیم.

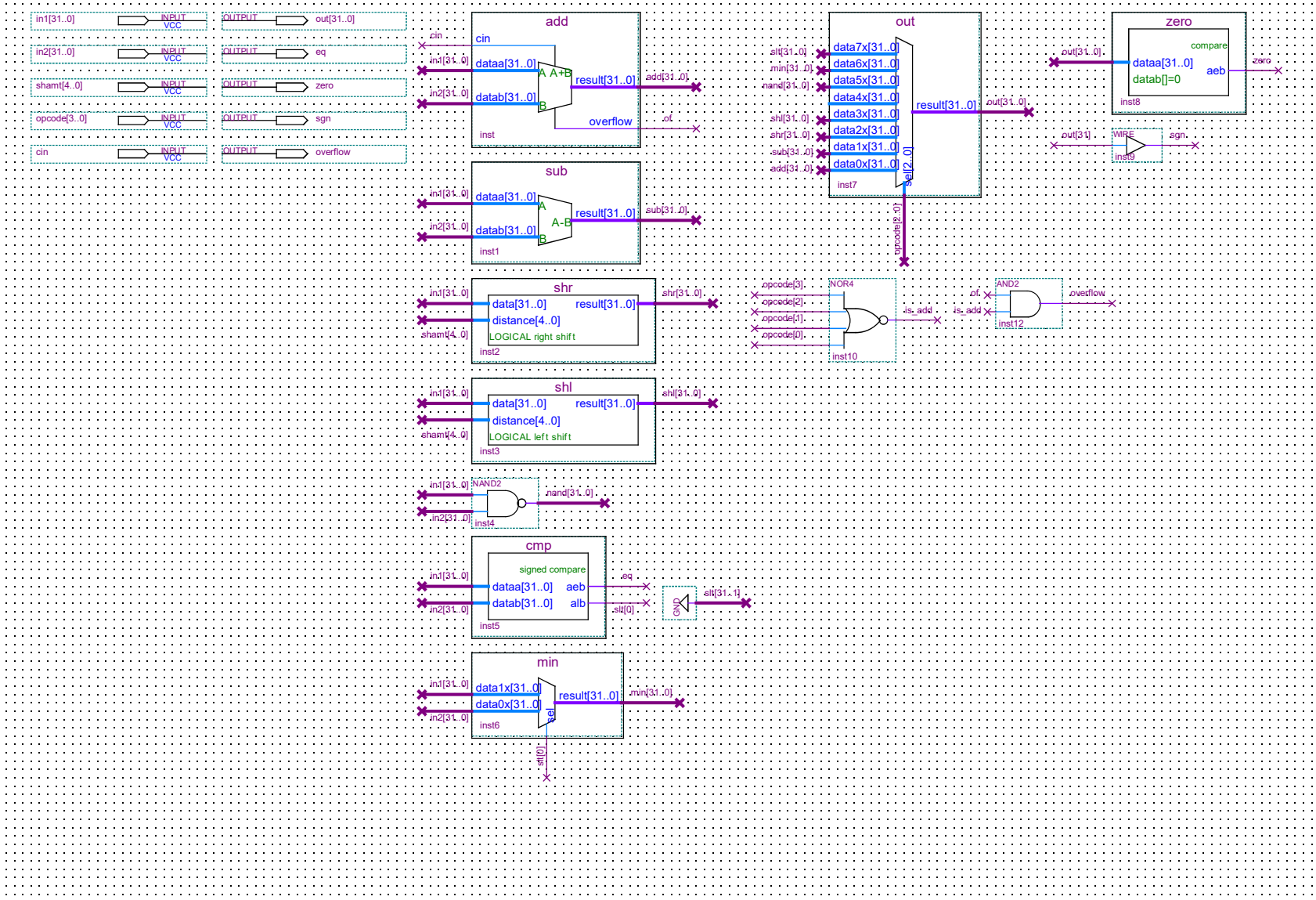
۲ طراحی ALU

بخش ALU در اصل سیستم محاسبه گر اصلی ما است و از آن جایی که بیش تر دستورات داده شده در مستندات پروژه برای اجرای به آن وابستگی دارند، از اهمیت بالایی برخوردار است. برای طراحی ALU به این شکل عمل کرده ایم پنج ورودی برای آن در نظر گرفته ایم که شامل داده ورودی ۱، داده ورودی ۲ هر دو به صورت ۳۲ بیتی، مقدار شیفت به صورت ۵ بیتی، مقدار cin یعنی carry ورودی به صورت ۱ بیتی و همچنین Opcode به صورت چهاربیتی بوده است.

برای عملیات های مختلف آن، از ماژول هایی که در Quartus در اختیار ما قرار گرفته است، استفاده کرده ایم. برای جمع از ماژول جمع کننده استفاده کرده و همچنین از آن جایی که کوارتوس برای آن از سیستم محاسبه Overflow هم پشتیبانی می کند، آن را هم قرار داده ایم. برای تفریق از ماژول تفریق گر استفاده کرده ایم، البته روش های دیگری هم نظیر نقیض کردن ورودی دوم و استفاده از cin=۱ به عنوان ورودی جمع کننده هم برای این کار وجود داشت ولی برای طراحی ساده تر و امکان دیباگ راحت تر، ترجیح دادیم که از ماژول جداگانه استفاده کنیم. برای شیفت به چپ و راست، ماژول های مربوط به آن ها را قرار داده ایم. این ماژول ها مقدار ShiftAmount را با نام distance و خود داده را هم با عنوان data به عنوان ورودی می گیرند. برای NAND یک گیت NAND دو ورودی قرار داده ایم که به ورودی های آنان، با توجه به این که کوارتوس توانایی تشخیص این که از مقادیر چند بیتی استفاده بکنیم را هم دارد، مقادیر ۳۲ بیتی مربوطه را داده و مقدار ۳۲ بیتی خروجی گرفته ایم. برای عملیات set on less than از یک ماژول Comparator استفاده کرده ایم. این ماژول به عنوان ورودی دو داده ۳۲ بیتی را گرفته و در خروجی دو داده به ما می دهد؛ داده aeb در صورتی که 1 باشد، بیانگر برابر بودن دو مقدار و در غیر این صورت، بیانگر نامساوی بودن آنان است، داده alb هم در صورت یک بودن، نشانگر کوچکتر بودن a از b و در غیر این صورت نشان دهنده بزرگتر مساوی بودن b از a است. از سیگنال alb برای حالت slt استفاده کرده ایم و از آن جایی که این خروجی ۱ بیتی است ولی به عنوان خروجی ALU ما ۳۲ بیت می خواهیم، سایر بیت ها را از طریق اتصال به GND صفر کردیم. در نهایت هم ماژول min را استفاده کرده ایم که همان طور که از نامش مشخص است، کمینه دو داده را حساب می کند.

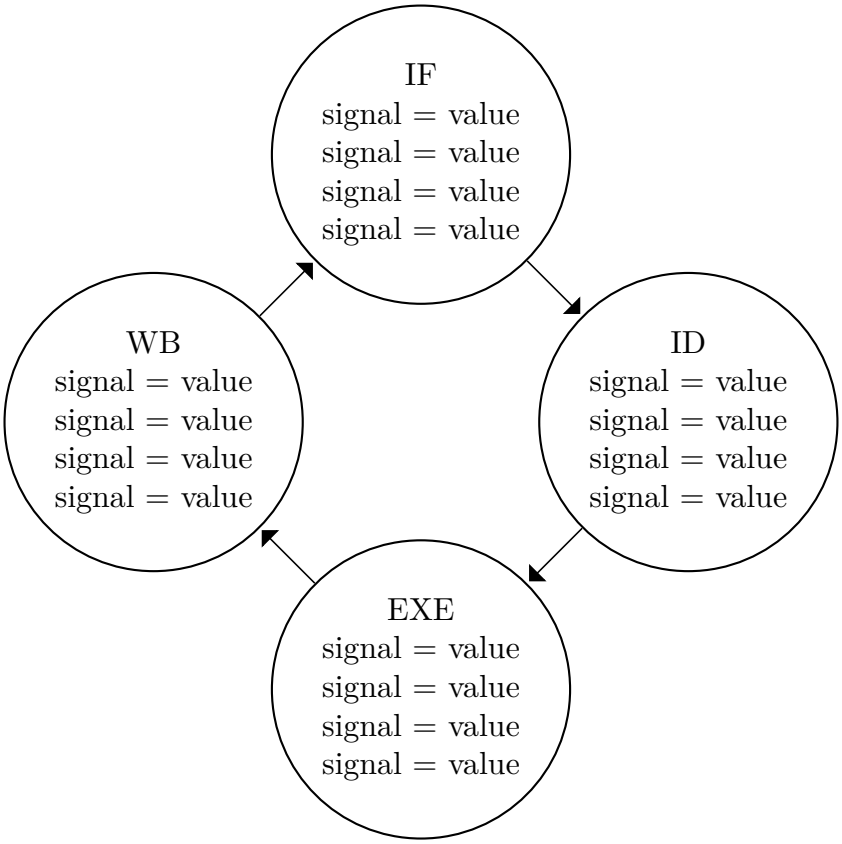
برای انتخاب این که خروجی کدام ماژول باید به عنوان خروجی نهایی داده شود، از یک مالتی پلکسر ۳۲ بیتی ۸ به ۱ استفاده کرده ایم که البته حالت 100 آن استفاده خاصی ندارد. برای خروجی sgn که نشانگر علامت خروجی است، از بیت 31 مقدار خروجی این مالتی پلکسر استفاده کرده ایم. همچنین برای خروجی eq هم از خروجی aeb ماژول Comparator استفاده کرده ایم. برای Overflow هم از آن جایی که سوال گفته فقط برای جمع باید در نظر گرفته شود، گیت هایی قرار داده ایم که براساس OpCode تنها درصورتی که در حالت جمع باشیم، مقدار Overflow ماژول جمع کننده به خروجی برود و در غیر این صورت، 0 به خروجی برود. یک ماژول به اسم zero هم قرار داده ایم که در اصل در پیاده سازی داخلی همان comparator است با این تفاوت که یکی از داده های آن به شکل درونی صفر شده است و مشخص می کند که آیا خروجی نهایی برابر صفر است یا نه.

شکل کلی ماژول ALU در صفحه بعد قرار دارد.

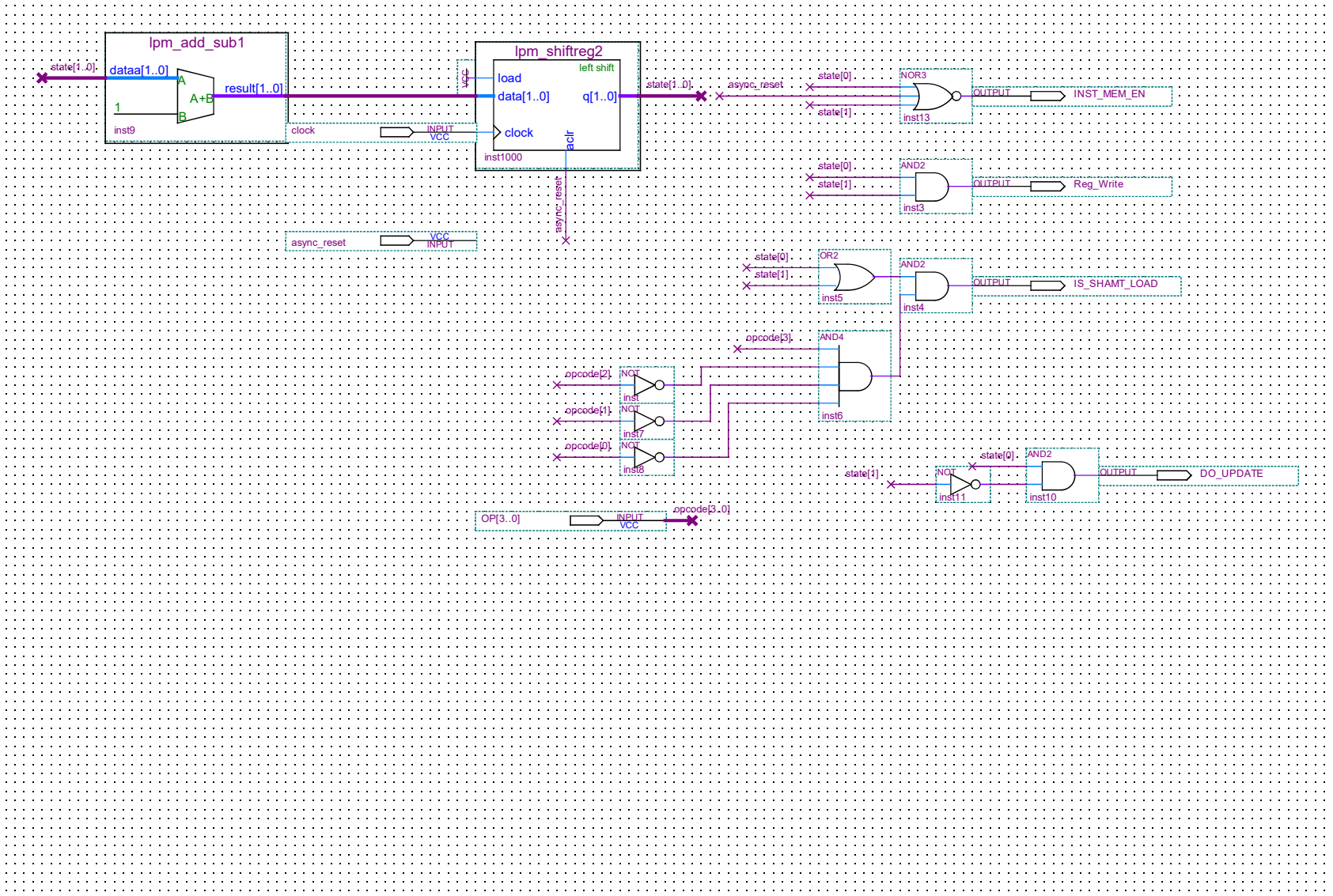

in1[31..0]
in2[31..0]
shamt[4..0]
opcode[3..0]
cin
INPUT
VCC
OUTPUT
out[31..0]
eq
zero
sgn
overflow
add
cin
dataa[31..0]
datab[31..0]
result[31..0]
add[31..0]
overflow
of
inst
sub
sub[31..0]
inst1
shr
data[31..0]
distance[4..0]
LOGICAL right shift
shr[31..0]
inst2
shl
LOGICAL left shift
shl[31..0]
inst3
NAND2
nand[31..0]
inst4
cmp
signed compare
aeb
alb
inst5
min
data1x[31..0]
data0x[31..0]
sel
inst6
out
data7x[31..0]
data6x[31..0]
data5x[31..0]
data4x[31..0]
data3x[31..0]
data2x[31..0]
data1x[31..0]
data0x[31..0]
sel[2..0]
inst7
zero
compare
dataa[31..0]
datab[]=0
aeb
inst8
WIRE
inst9
NOR4
inst10
AND2
inst12
is_add
overflow
GND

۳ طراحی واحد کنترل

برای طراحی این پردازنده، ما از ۴ کلاک استفاده کرده ایم که در شکل زیر State Machine آن قرار گرفته است.



شکل واحد کنترل در صفحه بعد قرار دارد.


lpm_add_sub1
state[1..0]
dataa[1..0]
A
A+B
result[1..0]
1
B
inst9
clock
INPUT
VCC
lpm_shiftreg2
left shift
load
data[1..0]
q[1..0]
clock
aclr
inst1000
state[1..0]
async_reset
INPUT
VCC
state[0]
state[1]
NOR3
inst13
OUTPUT
INST_MEM_EN
AND2
inst3
Reg_Write
OR2
inst5
AND2
inst4
IS_SHAMT_LOAD
opcode[3]
AND4
inst6
opcode[2]
opcode[1]
opcode[0]
NOT
inst
inst7
inst8
NOT
inst11
AND2
inst10
DO_UPDATE
OP[3..0]
opcode[3..0]

۴ مقداردهی حافظه دستورات

برای حافظه دستورات ما از بخش Mega-Plugin Wizard ارائه شده توسط Quartus و مقدار دهی ROM براساس فایل mif مشابه راهنمای قرار گرفته در مستند پروژه استفاده کرده ایم. یکی از نکات قابل توجه این جاست که در CycloneII که تا به حال طراحی های کوارتوس را با آن انجام دادیم، نمی توان رام چندان بزرگی استفاده کرد و مجبور شدیم از رامی با اندازه ۳۲۷۶۸ کلمه ۲۰ بیتی استفاده کنیم و برای تعداد کلمه بزرگ تر، در هنگام کامپایل به خطای در اختیار نبودن منبع برای سنتز چنین چیزی روی CycloneII بر می خوردیم.

محتویات فایل mif قرار گرفته بدین صورت است. در جلوی دستورات به صورت کامنت، کاری که دستور انجام می دهد هم مشخص شده است:

```
WIDTH=20;
DEPTH=32768;
ADDRESS_RADIX = UNS;          -- The radix for address values is unsigned
DATA_RADIX = BIN;             -- The radix for data values is Binary
CONTENT                       -- start of (address : data pairs)
BEGIN
0 : 10000000000000100001; -- load shamt r1 <-1
1 : 10000000000000100010; -- load shamt r2 <-1
2 : 10000000000001000011; -- load shamt r3 <-10
3 : 10000000000010000100; -- load shamt r4 <-100
4 : 00110000011111000001; -- r1 = r1 << 30
5 : 00000000000000100001; -- r1 = r0 + r1 (0)
6 : 00000000000001000010; -- r2 = r0 + r2 (0)
7 : 00000000000001100011; -- r3 = r0 + r3 (0)
8 : 00000000000010000100; -- r4 = r0 + r4 (0)
9 : 00000000100001000101; -- r5 = r2 + r2
10 : 00000000010000100110; -- r6 = r1 + r1  -> must overflow
11 : 00100001000000100100; -- r4 = r4 >> 1
12 : 01010001000001100111; -- r7 = r4 ~& r3
13 : 01110000000010001000; -- r8 = set on less than (r0 , r4)
14 : 01100000100010001001; -- r9 = min(r2 , r4)
15 : 10000000000000101010; -- load shamt r10 <-1
16 : 00001000000101001011; -- r11 = r0 - r10 (answer must be -1)
17 : 00000000000000100001; -- r1 = r0 + r1 (0)
18 : 00000000000001000010; -- r2 = r0 + r2 (0)
19 : 00000000000001100011; -- r3 = r0 + r3 (0)
20 : 00000000000001000010; -- r4 = r0 + r4 (0)
21 : 00000000000010100101; -- r5 = r0 + r5 (0)
22 : 00000000000011000110; -- r6 = r0 + r6 (0)
```

```
23 : 00000000000011100111; -- r7 = r0 + r7 (0)
24 : 00000000000100001000; -- r8 = r0 + r8 (0)
25 : 00000000000100101001; -- r9 = r0 + r9 (0)
26 : 00001000000101001010; -- r10 = r0 + r10 + cin(1)
27 : 00000000000101101011; -- r11 = r0 + r11
28:  00000000000101001010; -- r10 = r0 + r10
[29..32767] : 0000000000000000000;
END;
```

نکته مهم در مورد دستورات آخر که اکثرا جمع با صفر هستند، این است که به کمک این دستورات و Output های قرار داده شده، مقادیر هر کدام از رجیسترها را مشاهده بکنیم که ببینیم آیا در نهایت مقادیر قرار گرفته در آنان درست است یا نه.

خروجی های نهایی رجیسترها، باید به شکل زیر باشد (یعنی در خط های آخر، باید این اعداد را مشاهده کنیم):

```
R1 = 1073741824
R2 = 1
R3 = 2
R4 = 1
R5 = 2
R6 = -2147483648 -- this is caused by overflow on address 10
R7 = -3
R8 = 1
R9 = 1
R10 = on line 26 it is 1 and then on address 28 it is 2
R11 = 2
```

ضمنا باید توجه کنیم که این ROM ها به صورت Word-Addressable هستند و از طرفی PC ما چهار واحد چهار واحد جلو می رود و همچنین با توجه به فضای داده شده، برای یافتن آدرس درست تنها به ۱۵ بیت نیاز دارند، ما از بیت 2 تا 16 PC را به آن به عنوان ورودی آدرس داده ایم. دو بیت اول در اصل همیشه 0 هستند و با توجه به Word-Addressable بودن رام نیازی به آن نیست.