计算机组织结构

CPU 设计

04021702 牛小顿

04021225 王浩森

信息科学与工程学院

东南大学

2024 年 5 月 20 日

目录

一、 实验目的

1. 设计一个简单的 CPU(Central Processing Unit)，包含基本指令集。我们将利用其指令集生成一个简单的程序。为简单起见，我们只会考虑 CPU、寄存器、内存和指令集的关系。我们只需要考虑：读/写寄存器、读/写内存和执行指令。

2. CPU 工作有五个阶段，即获取指令、译码指令、获取数据、处理数据、写入数据。至少四个部分构成一个简单的 CPU：控制单元、内部寄存器、ALU 和指令集，这是项目设计的主要研究方面。

二、 实验原理

1. 操作码：

我们的简单的 CPU 设计使用单地址指令格式。指令字包含两个部分:操作数(码)和地址部分。操作数(码)定义了函数的指令。大多数指令的地址部分包含数据的内存位置操作，称之为直接寻址。在一些指令，地址部分是操作数，叫立即寻址。简化起见，内存的大小是 256×16。指令有 16 比特。其中操作码部分 8 比特，地址段 8 比特。指令字如图 1 所示。

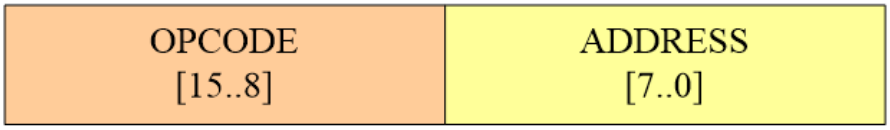

图 1 指令字

相关的操作码指令在表 1 中列出。

表 1

INSTRUTION	OPCODE	COMMENTS		
	00			
store	01	mem	← ACC	
load	02	ACC	← mem	
add	03	ACC	← ACC + mem	
sub	04	ACC	← ACC − mem	
jmpgez	05	PC	← imm8	if ACC ≥ 0
jmp	06	PC	← imm8	
halt	07			结束运行
mpy	08	{MR,ACC}	← ACC × mem	(unsigned)
div	09	ACC…MR	← ACC ÷ mem	(unsigned)
and	0A	ACC	← ACC & mem	
or	0B	ACC	← ACC | mem	
not	0C	ACC	← ~ACC	
shiftr	0D	ACC	← ACC >> 1	
shiftl	0E	ACC	← ACC << 1	
shiftar	0F	ACC	← ACC >>> 1	
disp	10	DR	← {MR,ACC}	数码管显示
	11			
load	12	ACC	← imm8	(符号位扩展)
add	13	ACC	← ACC + imm8	(符号位扩展)
sub	14	ACC	← ACC − imm8	(符号位扩展)
jmplz	15	PC	← imm8	if ACC < 0
jmpez	16	PC	← imm8	if ACC = 0
	17			
mpy	18	{MR,ACC}	← ACC × imm8	(0 扩展, unsigned)
div	19	ACC…MR	← ACC ÷ imm8	(0 扩展, unsigned)
and	1A	ACC	← ACC & imm8	(0 扩展)
or	1B	ACC	← ACC | imm8	(1 扩展)
neg	1C	ACC	← −ACC	
	1D			
	1E			
	1F			
	20			
mrs	21	mem	← MR	
	22			
	23			
	24			
jmpgz	25	PC	← imm8	if ACC > 0
jmpnez	26	PC	← imm8	if ACC ≠ 0
	27			
impy	28	{MR,ACC}	← ACC × mem	(signed)

idiv	29	ACC…MR	← ACC ÷ mem	(signed)
xor	2A	ACC	← ACC ^ mem	
	2B			
	2C			
shr	2D	ACC	← ACC >> mem	
shl	2E	ACC	← ACC << mem	
sar	2F	ACC	← ACC >>> mem	
	30			
	31			
	32			
	33			
	34			
jmplez	35	PC	← imm8	if ACC ≤ 0
	36			
	37			
impy	38	{MR,ACC}	← ACC × imm8	(符号位扩展, signed)
idiv	39	ACC…MR	← ACC ÷ imm8	(符号位扩展, signed)
xor	3A	ACC	← ACC ^ imm8	(0 扩展)
	3B			
	3C			
shr	3D	ACC	← ACC >> imm4	
shl	3E	ACC	← ACC << imm4	
sar	3F	ACC	← ACC >>> imm4	

2. 指令集(其中 00、FF、SS 指数据扩展方式)：

数据传送

store	mem	mem ← ACC
load	mem/imm8	ACC ← mem/{SS,imm8}
mrs	mem	mem ← MR
disp		DR(显示寄存器) ← {MR,ACC}

逻辑运算

not		ACC ← ~ACC
and	mem/imm8	ACC ← ACC & mem/{00,imm8}
or	mem/imm8	ACC ← ACC | mem/{FF,imm8}
xor	mem/imm8	ACC ← ACC ^ mem/{00,imm8}
shr	mem/imm4/1	ACC ← ACC >> mem/imm4/1
shl	mem/imm4/1	ACC ← ACC << mem/imm4/1

sar	mem/imm4/1	ACC ← ACC >>> mem/imm4/1

算术运算

neg		ACC ← −ACC
add	mem/imm8	ACC ← ACC + mem/{SS,imm8}
sub	mem/imm8	ACC ← ACC − mem/{SS,imm8}
mpy	mem/imm8	{MR,ACC} ← ACC × mem/{00,imm8}
div	mem/imm8	ACC…MR ← ACC ÷ mem/{00,imm8}
impy	mem/imm8	{MR,ACC} ← ACC × mem/{SS,imm8}
idiv	mem/imm8	ACC…MR ← ACC ÷ mem/{SS,imm8}

转移

jmp	imm8	PC ← imm8
jmpez	imm8	PC ← imm8 if ACC = 0
jmpnez	imm8	PC ← imm8 if ACC ≠ 0
jmpgez	imm8	PC ← imm8 if ACC ≥ 0
jmplz	imm8	PC ← imm8 if ACC < 0
jmpgz	imm8	PC ← imm8 if ACC > 0
jmplez	imm8	PC ← imm8 if ACC ≤ 0
halt		

3. 外部控制线：

trap	启用单步执行
pause	启用 disp 指令后暂停
step	继续执行

三、 实验内容

1. CPU 内部结构

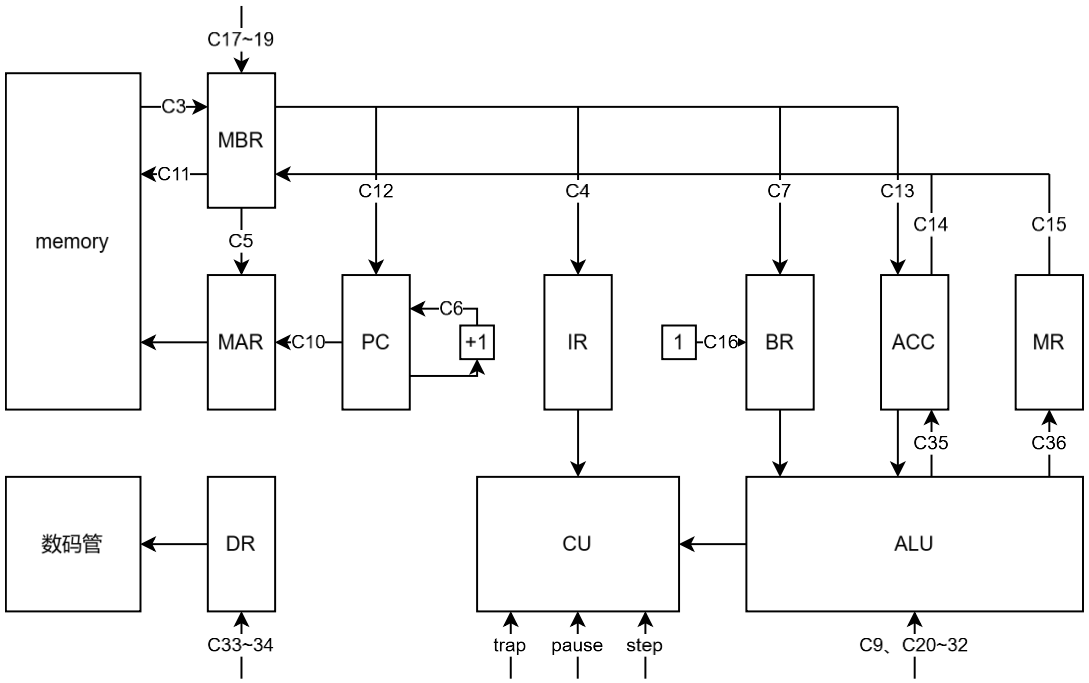

图 2 体系结构

控制信号如下表

表 2

控制信号	微操作
C0	CAR ← CAR + 1
C1	CAR ← IR<<2/…
C2	CAR ← 0
C3	MBR ← memory
C4	IR ← MBR[15:8]
C5	MAR ← MBR[7:0]
C6	PC ← PC + 1
C7	BR ← MBR
C8	ACC ← 0
C9	ACC + BR
C10	MAR ← PC
C11	memory ← MBR
C12	PC ← MBR[7:0]
C13	ACC ← MBR
C14	MBR ← ACC
C15	MBR ← MR
C16	BR ← 1

C17	MBR ← {8{MBR[7]},MBR[7:0]}
C18	MBR ← {00,MBR[7:0]}
C19	MBR ← {FF,MBR[7:0]}
C20	−ACC
C21	ACC − BR
C22	~ACC
C23	ACC & BR
C24	ACC | BR
C25	ACC ^ BR
C26	ACC >> BR
C27	ACC << BR
C28	ACC >>> BR
C29	ACC * BR (unsigned)
C30	ACC / BR (unsigned)
C31	ACC * BR (signed)
C32	ACC / BR (signed)
C33	DR ← {MR,ACC}
C34	DR ← {PC,IR,ACC}
C35	ACC ← ALU
C36	MR ← ALU

2. 内部寄存器和内存

1) MR

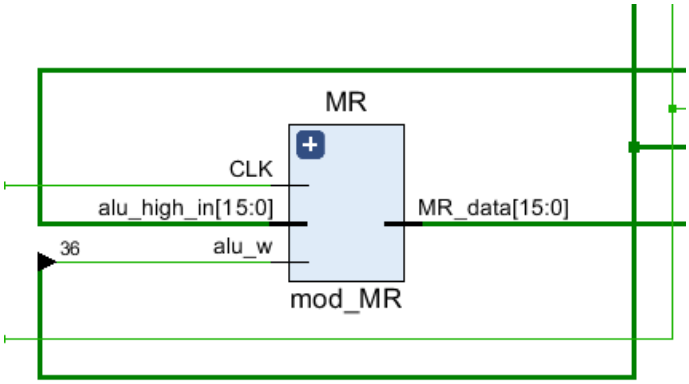

图 3 MR 寄存器

MR 寄存器为 16 位位宽，用于存储 ALU 乘法结果的高位和整数除法的余数。

2) ALU

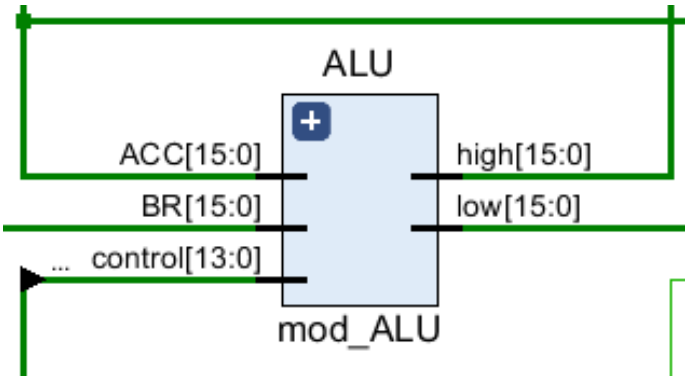

图 4 ALU 算术逻辑单元

ALU（算术逻辑单元）是完成基本算术和逻辑运算的计算单元，输入为 16 位位宽，乘法输出为 32 位位宽，其他运算输出为 16 位位宽。支持如表 3 的操作：

表 3

Operations	**Explanations**
add	ACC + BR
sub	ACC − BR
neg	−ACC
not	~ACC
and	ACC & BR
or	ACC | BR
xor	ACC ^ BR
shr	ACC >> BR
shl	ACC << BR
sar	$signed(ACC) >>> BR
mpy	ACC * BR
impy	$signed(ACC) * $signed(BR)
div	low = ACC / BR high = ACC % BR
idiv	low = $signed(ACC) / $signed(BR) high = $signed(ACC) % $signed(BR)

3) BR

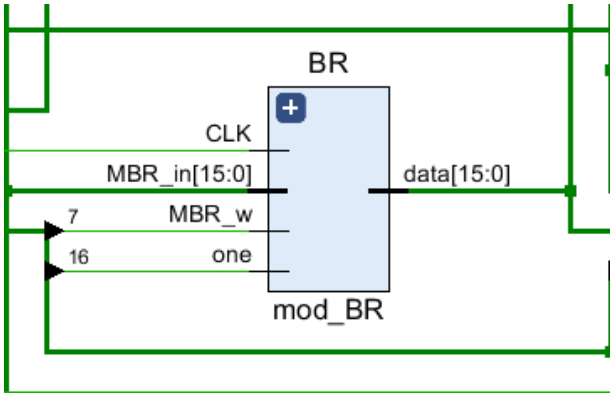

图 5 BR 寄存器

BR 寄存器存放着 ALU 的一个操作数，位宽为 16 位。

4) ACC

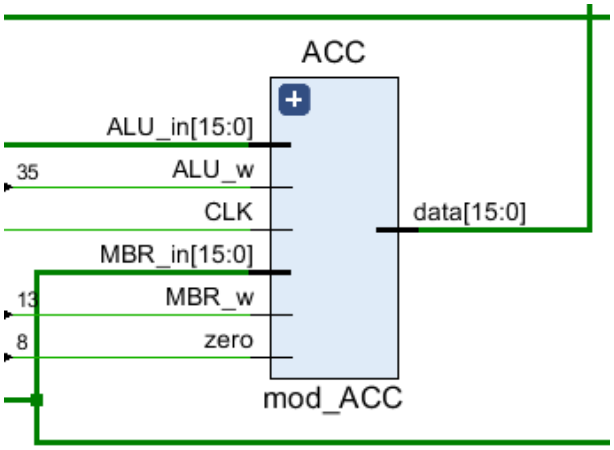

图 6 ACC

ACC 保存着 ALU 的另一个操作数，并存放 ALU 的计算结果，位宽为 16 位。

5) IR

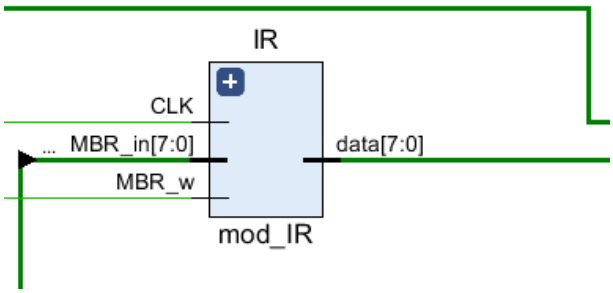

图 7 IR 寄存器

IR 存放着指令的操作码部分，位宽为 8 位。

6) MAR

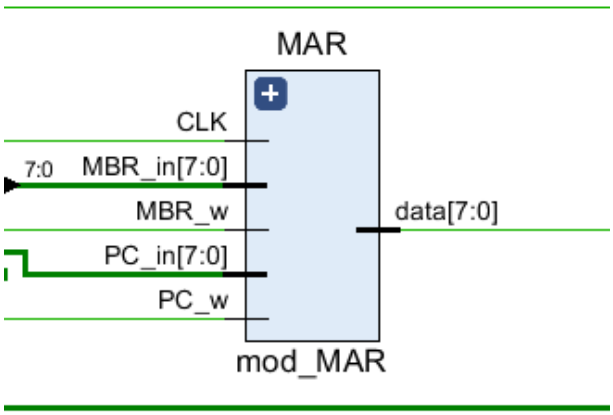

图 8 MAR

MAR 存放着要从存储器中读取或要写入存储器的存储器地址。位宽为 8 位，可以存放 256 个地址。

7) MBR

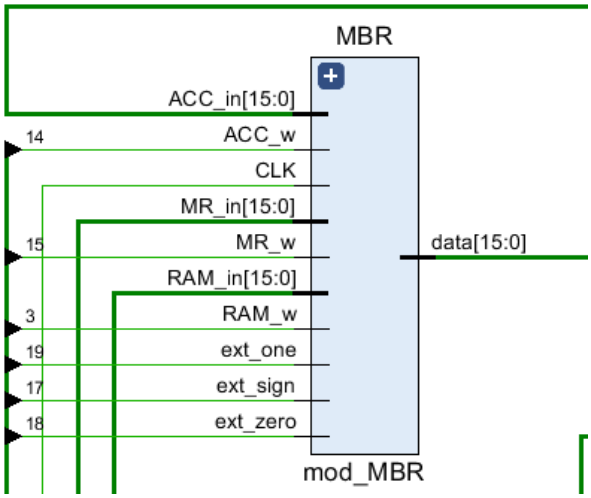

图 9 MBR

MBR 存储着将要被存入内存或者最后一次从内存中读出来的数值，位宽为 16 位。同时也承担着将 8 位立即数扩展为 16 位的工作。

8) PC

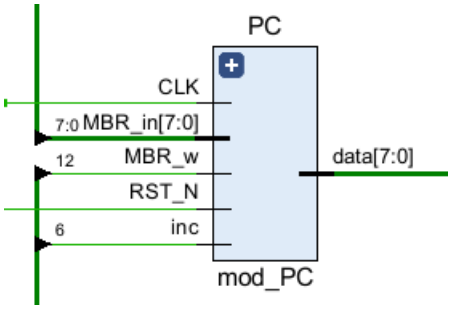

图 10 PC

PC 寄存器用来跟踪程序中将要执行的指令，位宽为 8 位。

9) DR

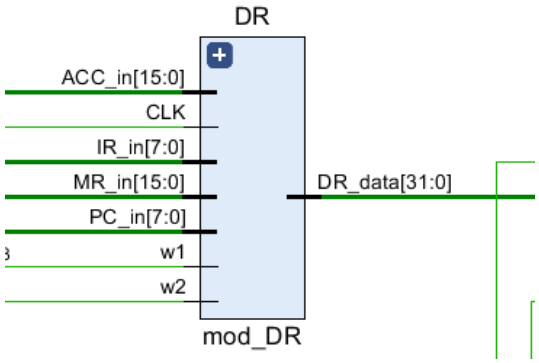

图 11 DR

DR 寄存器为显示寄存器，位宽为 32 位，由控制信号决定显示内容：当执行 disp 指令时，显示 MR 和 ACC 的值；当启用单步执行并准备取下一条指令时，显示 PC、IR 和 ACC 的值。

10) memory

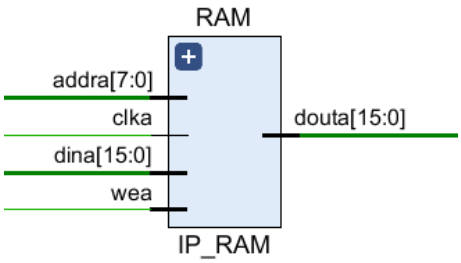

图 12 RAM

单端口 RAM IP 核，地址线为 8 位，位宽为 16 位，写优先，

always 使能，下降沿触发以降低读写延迟，用于存放程序机器码。

11) CU

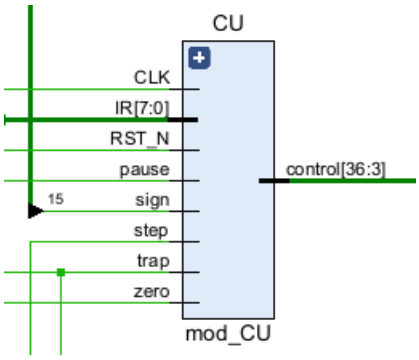

图 13 CU

根据 IR、ACC 状态和外部控制线上的信息产生控制信号。内部包含一个单端口 ROM IP 核，地址线为 8 位，位宽为 37 位，写优先，always 使能，下降沿触发以降低读写延迟，用于存放控制单元的控制信号。

3. 微程序控制单元 CU

我们已经学习了微程序控制单元的知识。在微程序控制中，微程序由一些微指令组成，微程序存储在控制存储器中，控制存储器生成正确执行指令集所需的所有控制信号。微指令包含一些同时执行的微操作。

如图 14 所示，微指令集存储在控制存储器中。控制地址寄存器包含要读取的下一条微指令的地址。当从控制存储器读取微指令时，它被传输到控制缓冲寄存器。寄存器连接到控制单元发出的控制线。因此，从控制存储器读取微指令与执行该微指令相同。图中所示的第三个元素是一个排序单元，它加载控制地址寄存器并发出读取命令。

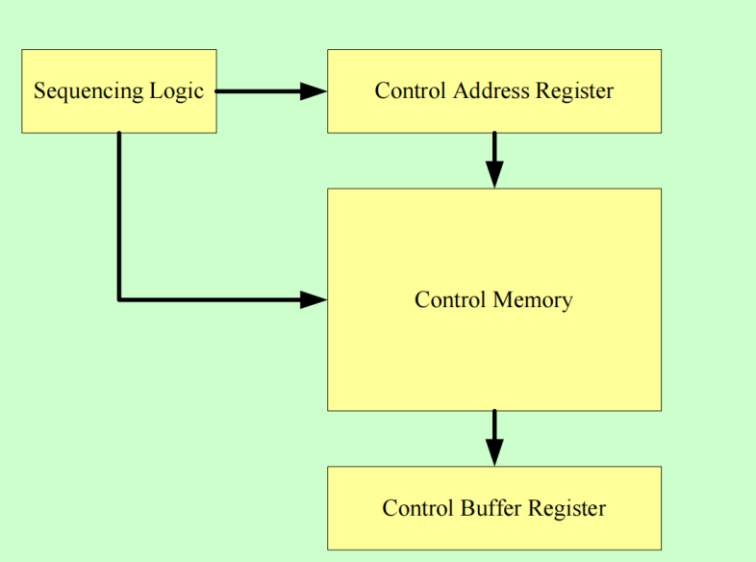

图 14 控制单元微结构

设计 CU 时，将每条指令分为 fetch 和 execute 两个阶段，fetch 阶共用相同的微程序。为简化译码考虑，每条指令的 execute 阶段微程序的起始地址设计为 4*IR，opcode 只使用低 6 位，因此共 256 条微指令。大多数指令的 execute 阶段都能在 4 周期内完成，只有 div 和 idiv 需要更多时钟周期，因此设计 div 和 idiv 指令在第 4 周期中经过一次跳转过程，使用 8 条微指令。

微指令与对应的微操作、控制内存的关系如下表所示：

表 4

Stage	Micro-Operations		Control Signals
fetch	00	MAR←PC; DR←{PC,IR,ACC}	C10 C34 C0
	01	MBR←memory; PC←PC+1	C3 C6 C0
	02	IR←MBR	C4 C0
	03	\	C1(单步暂停/CAR←4*IR)
store	04	MAR←MBR	C5 C0
	05	MBR←ACC	C14 C0
	06	memory←MBR	C11 C2
load m	08	MAR←MBR	C5 C0
	09	MBR←memory	C3 C0
	0A	ACC←MBR	C13 C2
load i	48	MBR←{8{MBR[7],MBR[7:0]}}	C17 C0
	49	ACC←MBR	C13 C2
mrs	84	MAR←MBR	C5 C0

	85	MBR←MR	C15 C0
	86	memory←MBR	C11 C2
disp	40	DR←{MR,ACC}	C33 C1(暂停/CAR←0)
halt	1C	\	\
jmp	18	PC←MBR	C12 C2
jmpgez 等	4x	\	C1(CAR←0/CAR++)
条件跳转	4x+1	PC←MBR	C12 C2
not/neg	4x	运算	Cx C0
	4x+1	运算; ACC←ALU	Cx C35 C2
shiftr/shiftl	4x	BR←1; 运算	C16 Cx C0
/shiftar	4x+1	运算; ACC←ALU	Cx C35 C2
add m 等	4x	MAR←MBR	C5 C0
直接寻址	4x+1	MBR←memory	C3 C0
双目运算	4x+2	BR←MBR; 运算	C7 Cx C0
(乘除除外)	4x+3	运算; ACC←ALU	Cx C35 C2
mpy/impy m	4x	MAR←MBR	C5 C0
	4x+1	MBR←memory	C3 C0
	4x+2	BR←MBR; 运算	C7 Cx C0
	4x+3	运算; ACC←ALU; MR←ALU	Cx C35 C36 C2
div m	24	MAR←MBR	C5 C0
	25	MBR←memory	C3 C0
	26	BR←MBR; div	C7 C30 C0
	27	div	C30 C1(CAR←AC)
	AC	div	C30 C0
	AD	div	C30 C0
	AE	div	C30 C0
	AF	div; ACC←ALU; MR←ALU	C30 C35 C36 C2
idiv m	A4	MAR←MBR	C5 C0
	A5	MBR←memory	C3 C0
	A6	BR←MBR; idiv	C7 C30 C0
	A7	idiv	C32 C1(CAR←B0)
	B0	idiv	C32 C0
	B1	idiv	C32 C0
	B2	idiv	C32 C0
	B3	idiv; ACC←ALU; MR←ALU	C30 C35 C36 C2
add/sub/	4x	MBR←{8{MBR[7],MBR[7:0]}}	C17 C0
impy i	4x+1	BR←MBR; 运算	C7 Cx C0
	4x+2	运算; ACC←ALU	Cx C35 C2
impy i	E0	MBR←{8{MBR[7],MBR[7:0]}}	C17 C0
	E1	BR←MBR; impy	C7 C31 C0
	E2	impy; ACC←ALU; MR←ALU	C31 C35 C36 C2
shr/shl/sar	4x+1	MBR←{00,MBR[7:0]}}	C18 C0
/and/xor i	4x+2	BR←MBR; 运算	C7 Cx C0

	4x+3	运算; ACC←ALU	Cx C35 C2
mpy i	60	MBR←{00,MBR[7:0]}}	C18 C0
	61	BR←MBR; mpy	C7 C29 C0
	62	mpy; ACC←ALU; MR←ALU	C29 C35 C2
or i	6C	MBR←{FF,MBR[7:0]}}	C19 C0
	6D	BR←MBR; or	C7 Cx C0
	6E	or; ACC←ALU	Cx C35 C2
div i	64	MBR←{00,MBR[7:0]}}	C18 C0
	65	BR←MBR; div	C7 C30 C0
	66	div	C30 C0
	67	div	C30 C1(CAR←AC)
	AC	div	C30 C0
	AD	div	C30 C0
	AE	div	C30 C0
	AF	div; ACC←ALU; MR←ALU	C30 C35 C36 C2
idiv i	E4	MBR←{8{MBR[7],MBR[7:0]}}	C17 C0
	E5	BR←MBR; idiv	C7 C32 C0
	E6	idiv	C32 C0
	E7	idiv	C32 C1(CAR←B0)
	B0	idiv	C32 C0
	B1	idiv	C32 C0
	B2	idiv	C32 C0
	B3	idiv; ACC←ALU; MR←ALU	C32 C35 C36 C2

四、 设计亮点

1. 微指令执行频率为 100MHz，不同指令时钟周期从 5~12 不等，最大化提升执行效率；
2. 所有指令共用 fetch 阶段微指令，execute 阶段微指令起始地址为 4*IR，寻址简单，ROM 利用率高，且可将 execute 阶段扩展至 8 条及以上；
3. 调试功能丰富，可在 disp 显示时暂停，可单步调试，能显示 PC、IR 的值，方便定位程序位置；
4. 扩展了原始指令集，为所有运算增加了立即数寻址方式，根据具体

运算对 8 位立即数进行 0 扩展、1 扩展或符号位扩展，增加了多种条件跳转指令。

五、 仿真测试

1. 对实验指导书中的 1+2+..+100 进行仿真验证。

高级语言	汇编语言
sum = 0 i = 100 next: sum = sum + i i = i - 1 if (temp1 >= 0) goto next disp(sum)	.code load i next: add sum disp store sum load i sub one store i jmpgez next halt .data 128 i 100 sum 0 one 1

仿真过程如下：

如图为循环加法的仿真，每加一次存储的 i 值减 1，ACC 低位将每次的 i 值累加

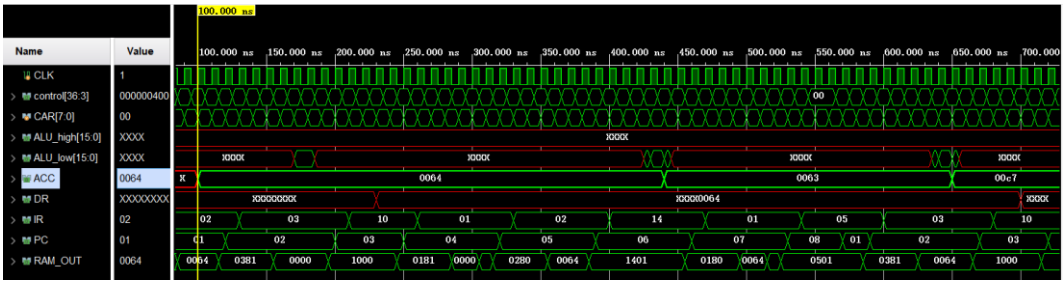

图 13

最终结果如下，ACC 内的值为 5050 的 16 进制 13ba，显示寄存器 DR

内值也为 13ba。

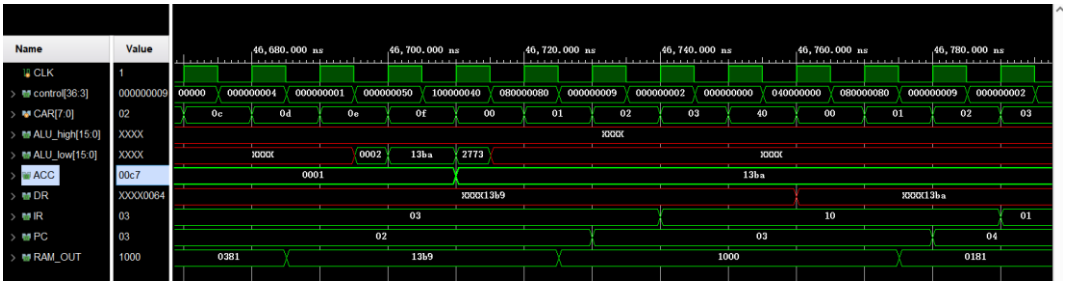

图 14

2. 对(27964)*（-31279）or(1984) SHR2(算术右移)进行仿真验证

高级语言	汇编语言
a = 27964 b = -31279 c = 1984 r = a * b r = r | c r = r >>> 2 disp(r)	.code load a impy b ;CBDD 59FC or c ;CBDD 5FFC shiftar shiftar ;CBDD 17FF disp halt .data 128 a 27964 b -31279 c 1984

仿真过程如下：

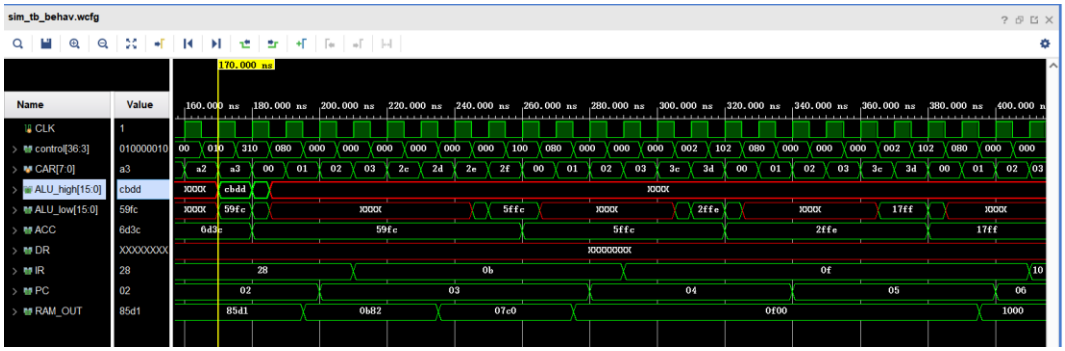

图 15

乘法高位在 ALU_high 为 CBDD，后续 or 和算术右移均是低 16 位进行操作，为 17FF，与计算器验证结果一致

3. 对(31431)/(-1925）进行仿真验证

高级语言	汇编语言
a = 31431 b = -1925 r = a / b disp(r)	.code load a ; 7AC7 idiv b ; F87B disp ;0277 FFF0 halt .data 128 a 31431 b -1925

仿真过程如下：

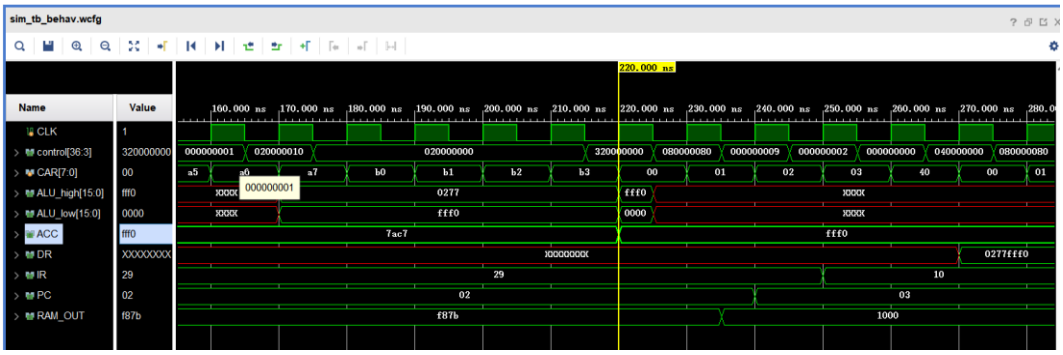

图 16

除法余数在 ALU_high，为 0277，商为 fff0 在 ALU_low，与计算器验证结果一致。

4. 对 NOT((19+21+23+…+199)*(-15)) SHR(2)*(-5)验证。

高级语言	汇编语言
sum = 0 i = 199 next: sum = sum + i i = i - 2 if (i - 19 >= 0) goto next r = sum * -15 r = ~r r = r >> 2 r = r * -5 disp(r)	.code next: load i add sum disp store sum load i sub A0 store i sub n19 jmpgez next

```
    load    sum
    disp
    impy    A1
    disp
    not
    shiftr
    disp
    shiftr
    disp
    impy    A2
    disp
    halt
.data 128
    sum     0
    i       199
    n19     19
    A0      2
    A1      -15
    A2      -5
```

仿真过程如下：

首先是循环加法，加法结果为 26BF。

然后是与-15 的有符号乘法，为 FFFD BACF，之后高位存储在 MR 寄存器中，低 16 位取反为 4530，右移两位为 114C，与-5 相乘结果为 A984，与计算器验证结果一致。

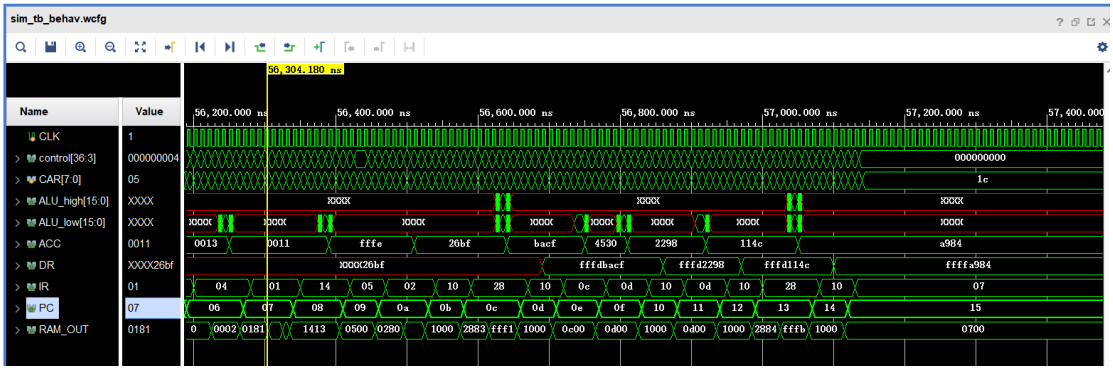

图 17

5. 对部分跳转指令测试。

汇编语言如下：

```
.code
    jmp     LB
LA:
    load    a
    disp
    jmp     LC
LB:
    load    b
    disp
    jmp     LA
LC:             ;A2F9
                ;35D2
    load    a
    jmpgez  B1
    load    a
    jmp     F1
B1:
    load    b
F1:
    disp        ;A2F9
    load    b
    jmpgez  B2
    load    a
    jmp     F2
B2:
    load    b
F2:
    disp        ;35D2
    halt

.data 192
    a       35D2H
    b       A2F9H
```

仿真过程如下：

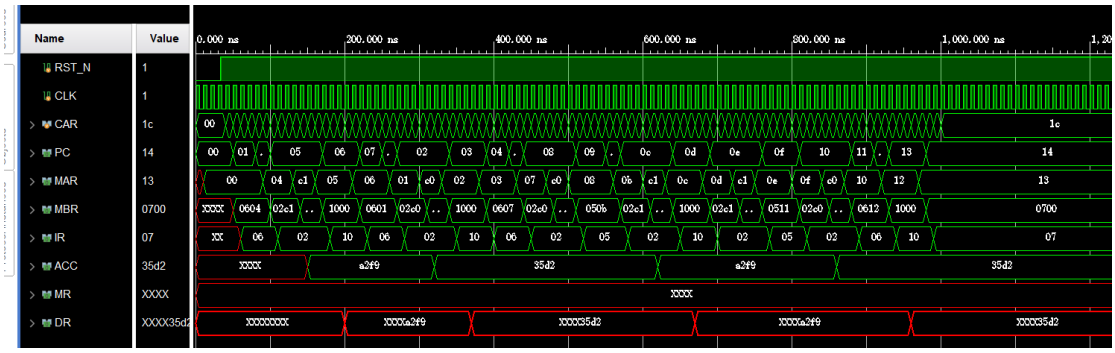

图 18

从 DR 波形可以看到，每次调用 disp 展示的值与程序中的预期相符，PC 的变化也符合跳转规则。

六、 实物验证

外部控制线：外部控制线共有 3 条，pause、trap 和 step。pause 控制线连接最右侧开关 SW[0], 闭合时每当执行完 disp 指令后暂停运行；trap 控制线连接次右侧开关 SW[1]，闭合时在执行每条指令前暂停运行；step 控制线连接中央按钮，经过消抖，按下后从暂停中恢复。

数码管和 LED：每当执行 disp 指令后，数码管高 4 位显示 MR 的值，低 4 位显示 ACC 的值，同时最右侧 LED[0]点亮；当启用单步调试时，在执行每条指令前，数码管高 2 位显示 PC 的值，次高 2 位显示 IR 的值，低 4 位显示 ACC 的值，同时次右侧 LED[1]点亮。

当处于正常模式开关均未闭合，直接显示运行结果，如下图显示仿真 1 循环加法的结果，与仿真结果相符合。

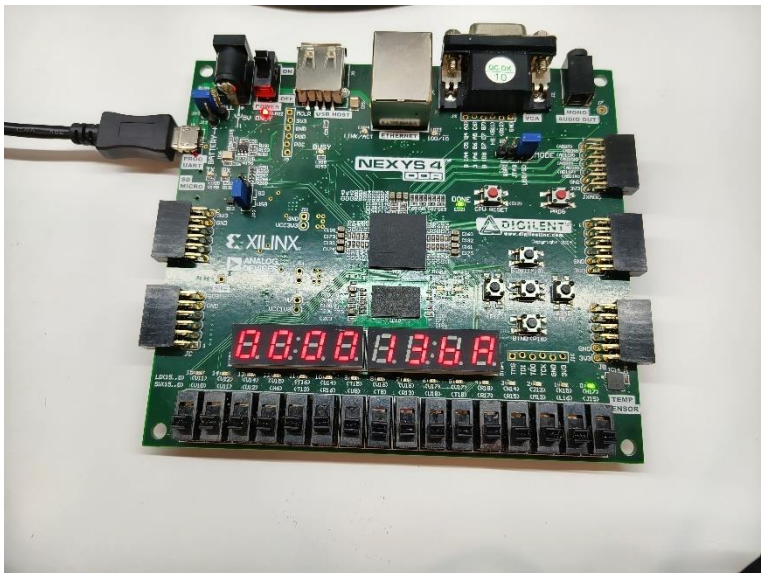

图 19

第二种模式为显示模式，将最右侧开关上划，每按一次中间按键，显示一次 disp 时寄存器的值。

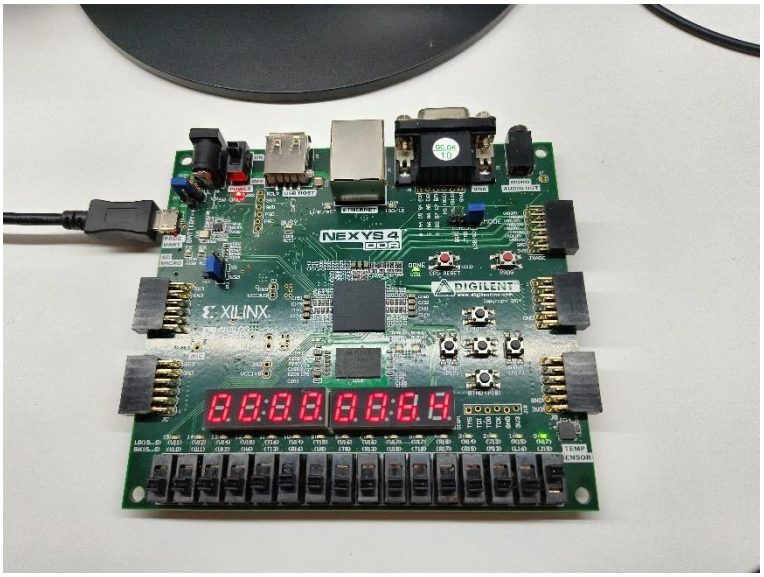

图 20

第三种模式为单步调试模式，将次右侧开关划至上侧，右侧第二个数码管亮起时代表当前高 4 位显示的是 PC 和 IR 的值，便于调试。

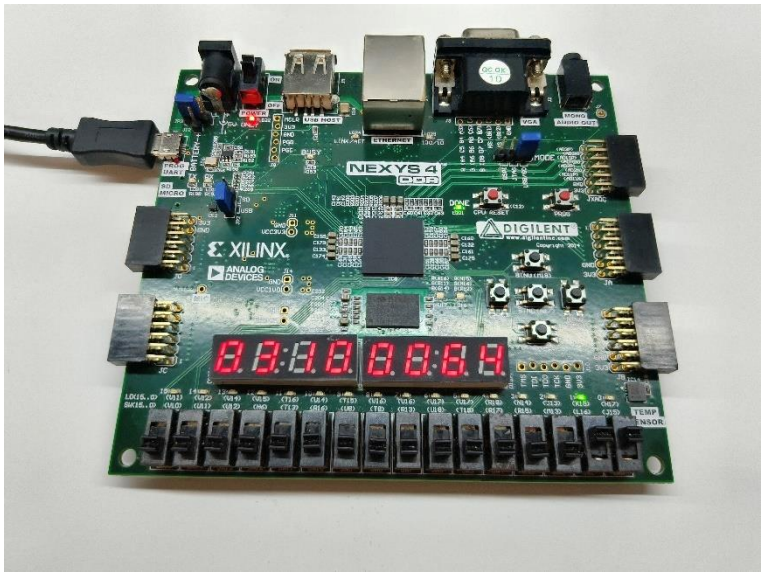

图 21

我们对全部 42 条指令进行了上板测试，均与仿真结果相符。

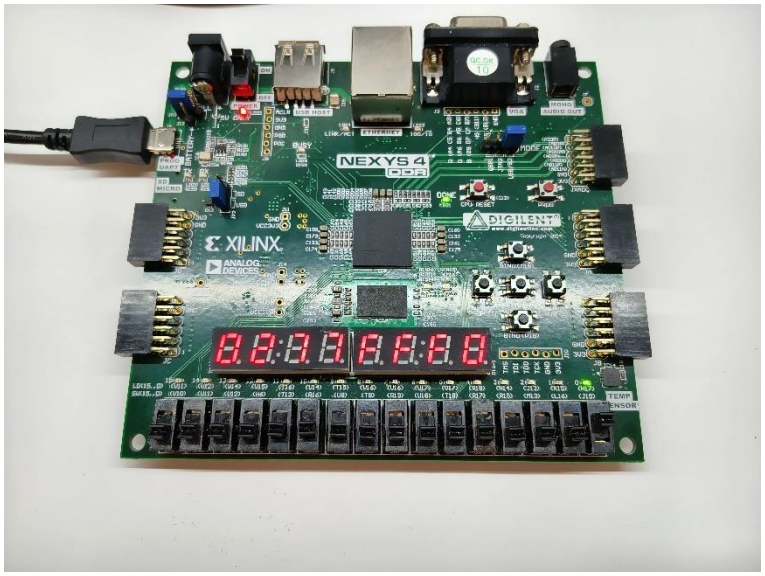

图 22

如图是除法结果的上板测试图，最终结果商为 FFF0，余数为 0277，与仿真结果相同。

七、 实验总结

1. 牛小顿

该 CPU 实验的难点在于微程序 CU 的设计，而 CU 设计中最棘手的是其中延迟问题的处理。下面以 jmp 指令为例展示设计过程中遇

到的问题和解决办法。

jmp 指令包含以下微程序：

fetch	00 01 02 03	MAR←PC MBR←memory; PC←PC+1 IR←MBR \	C10 C0 C3 C6 C0 C4 C0 C1(CAR←4*IR)
jmp	18	PC←MBR	C12 C2

将时序列入下表，每列代表该模块当前正在执行微指令地址：

CAR	ROM	registers	memory
00			
	00		
01		00	
02	01		00
	02	01	
03		02	
	03		
18			
	18		
00		18	
	00		
01		00	

可以看到，其中必须插入许多空操作才能保证微程序执行。MBR←memory 需要等待 memory 数据输出，CAR 的跳转也要等待 ROM 读取出微操作才能完成，同时根据 IR 的跳转也需要等待 IR 寄存器数据准备好。整个 fetch 需要 7 周期来完成，浪费了运行效率和 ROM 空间，下面通过改变时钟的触发沿来优化时序表：

CAR	ROM	registers	memory
00			
01	00	00	
02	01	01	00
03	02	02	
18	03		
00	18	18	
01	00	00	

通过将 ROM 和 memory 设置为下降沿触发，错位半周期，减小了 memory 和 registers、registers 和 ROM 以及 ROM 和 CAR 之间的延时，恰好解决了上述 3 个问题所产生的空操作。调整后，整个 ROM 不再需要任何空操作，极大优化了运行效率和 ROM 利用率。

2. 王浩森

该 CPU 实验我负责的部分是内部寄存器如 MBR、MAR 的编写，负责 ALU 算数部分，完成各部分的整合，进行仿真验证，并撰写实验报告。在实验中遇到的问题主要是将寄存器模块化时输入时钟的问题，为此将 RAM 和 ROM 的时钟改为下降沿触发，其他寄存器均为上升沿触发。在调试除法的过程中，由于延时较大，系统时钟为 100M 时还未计算完全，在分频后计算结果正确，为此将除法扩展为了 8 个周期。

附录(代码):

```
module mod_top (
    input             RST_N,
    input             CLK,
    input             TRAP,
    input             PAUSE,
    input             STEP,
    output wire       CA,
    output wire       CB,
    output wire       CC,
    output wire       CD,
    output wire       CE,
    output wire       CF,
    output wire       CG,
    output wire [7:0] AN,
    output reg  [1:0] LED
);
    wire [ 7:0] MAR_out;
    wire [15:0] MBR_out;
    wire [ 7:0] PC_out;
    wire [ 7:0] IR_out;
    wire [15:0] ACC_out;
    wire [15:0] BR_out;
    wire [15:0] MR_out;
    wire [31:0] DR_out;

    wire [36:3] control;
    wire [15:0] RAM_out;
    wire [15:0] ALU_high;
    wire [15:0] ALU_low;

    mod_MAR MAR (
        CLK,
        control[5],
        MBR_out[7:0],
        control[10],
        PC_out,
        MAR_out
    );
    mod_MBR MBR (
        CLK,
```

```
    control[3],
    RAM_out,
    control[14],
    ACC_out,
    control[15],
    MR_out,
    control[17],
    control[18],
    control[19],
    MBR_out
);
mod_PC PC (
    RST_N,
    CLK,
    control[6],
    control[12],
    MBR_out[7:0],
    PC_out
);
mod_IR IR (
    CLK,
    control[4],
    MBR_out[15:8],
    IR_out
);
mod_ACC ACC (
    CLK,
    control[8],
    control[13],
    MBR_out,
    control[35],
    ALU_low,
    ACC_out
);
mod_BR BR (
    CLK,
    control[7],
    MBR_out,
    control[16],
    BR_out
);
mod_MR MR (
```

```
    CLK,
    control[36],
    ALU_high,
    MR_out
);
mod_DR DR (
    CLK,
    control[33],
    control[34] && TRAP,
    MR_out,
    ACC_out,
    PC_out,
    IR_out,
    DR_out
);

wire step;
mod_CU CU (
    RST_N,
    CLK,
    IR_out,
    ACC_out[15],
    ~|ACC_out,
    TRAP,
    PAUSE,
    step,
    control
);
IP_RAM RAM (
    !CLK,
    control[11],
    MAR_out,
    MBR_out,
    RAM_out
);
mod_ALU ALU (
    {control[32:20], control[9]},
    ACC_out,
    BR_out,
    ALU_high,
    ALU_low
);
```

```
    mod_disp disp (
        CLK,
        DR_out,
        AN,
        {CA, CB, CC, CD, CE, CF, CG}
    );
    always @(posedge CLK) begin
        if (control[33]) LED <= 'b01;
        if (control[34] && TRAP) LED <= 'b10;
    end
    mod_button button (
        CLK,
        STEP,
        step
    );
endmodule

module mod_CU (
    input             RST_N,
    input             CLK,
    input      [ 7:0] IR,
    input             sign,    // ACC 符号位
    input             zero,    // ACC 为零
    input             trap,
    input             pause,
    input             step,
    output wire [36:3] control
);
    reg  [ 7:0] CAR;
    wire [36:0] ROM_out;
    IP_ROM ROM (
        !CLK,
        CAR,
        ROM_out
    );
    always @(posedge CLK, negedge RST_N)
        if (!RST_N) CAR <= 0;
        else
            case (ROM_out[2:0])
                'b000: CAR <= CAR;
                'b001: CAR <= CAR + 1;
```

```
                'b100: CAR <= 0;
                'b010:
                case (CAR)
                    3:  // fetch
                    if (!trap || step) CAR <= IR << 2;
                    'h10 * 4:  // disp
                    if (!pause || step) CAR <= 0;
                    'h05 * 4:  // jmpgez
                    if (!sign) CAR <= CAR + 1;
                    else CAR <= 0;
                    'h15 * 4:  // jmplz
                    if (sign) CAR <= CAR + 1;
                    else CAR <= 0;
                    'h16 * 4:  // jmpez
                    if (zero) CAR <= CAR + 1;
                    else CAR <= 0;
                    'h25 * 4:  // jmpgz
                    if (!sign && !zero) CAR <= CAR + 1;
                    else CAR <= 0;
                    'h26 * 4:  // jmpnez
                    if (!zero) CAR <= CAR + 1;
                    else CAR <= 0;
                    'h35 * 4:  // jmplez
                    if (sign || zero) CAR <= CAR + 1;
                    else CAR <= 0;
                    'h09 * 4 + 3:  // div m
                    CAR <= 'h2B * 4;
                    'h19 * 4 + 3:  // div i
                    CAR <= 'h2B * 4;
                    'h29 * 4 + 3:  // idiv m
                    CAR <= 'h2C * 4;
                    'h39 * 4 + 3:  // idiv i
                    CAR <= 'h2C * 4;
                    default: CAR <= 'dX;
                endcase
                default: CAR <= 'dX;
            endcase
    assign control = ROM_out[36:3];
endmodule

module mod_ALU (
    input      [13:0] control,
```

```
    input      [15:0] ACC,
    input      [15:0] BR,
    output reg [15:0] high,
    output reg [15:0] low
);
    always @(*)
        case (control)
            'b00000000000001: begin
                low  = ACC + BR;
                high = 'dX;
            end
            'b00000000000010: begin
                low  = -ACC;
                high = 'dX;
            end
            'b00000000000100: begin
                low  = ACC - BR;
                high = 'dX;
            end
            'b00000000001000: begin
                low  = ~ACC;
                high = 'dX;
            end
            'b00000000010000: begin
                low  = ACC & BR;
                high = 'dX;
            end
            'b00000000100000: begin
                low  = ACC | BR;
                high = 'dX;
            end
            'b00000001000000: begin
                low  = ACC ^ BR;
                high = 'dX;
            end
            'b00000010000000: begin
                low  = ACC >> BR;
                high = 'dX;
            end
            'b00000100000000: begin
                low  = ACC << BR;
                high = 'dX;
```

```
            end
            'b000010000000000: begin
                low  = $signed(ACC) >>> BR;
                high = 'dX;
            end
            'b000100000000000: {high, low} = ACC * BR;
            'b001000000000000: begin
                low  = ACC / BR;
                high = ACC % BR;
            end
            'b010000000000000: {high, low} = $signed(ACC) *
$signed(BR);
            'b100000000000000: begin
                low  = $signed(ACC) / $signed(BR);
                high = $signed(ACC) % $signed(BR);
            end
            default: {high, low} = 'dX;
        endcase
endmodule

module mod_ACC (
    input            CLK,
    input            zero,
    input            MBR_w,
    input     [15:0] MBR_in,
    input            ALU_w,
    input     [15:0] ALU_in,
    output reg [15:0] data
);
    always @(posedge CLK) begin
        if (zero) data <= 0;
        if (MBR_w) data <= MBR_in;
        if (ALU_w) data <= ALU_in;
    end
endmodule

module mod_MBR (
    input            CLK,
    input            RAM_w,
    input     [15:0] RAM_in,
    input            ACC_w,
    input     [15:0] ACC_in,
```

```
    input            MR_w,
    input      [15:0] MR_in,
    input            ext_sign,
    input            ext_zero,
    input            ext_one,
    output reg [15:0] data
);
    always @(posedge CLK) begin
        if (RAM_w) data <= RAM_in;
        if (ACC_w) data <= ACC_in;
        if (MR_w) data <= MR_in;
        if (ext_sign) data[15:8] <= {8{data[7]}};
        if (ext_zero) data[15:8] <= 'h00;
        if (ext_one) data[15:8] <= 'hFF;
    end
endmodule

module mod_MAR (
    input            CLK,
    input            MBR_w,
    input      [7:0] MBR_in,
    input            PC_w,
    input      [7:0] PC_in,
    output reg [7:0] data
);
    always @(posedge CLK) begin
        if (MBR_w) data <= MBR_in;
        if (PC_w) data <= PC_in;
    end
endmodule

module mod_PC (
    input            RST_N,
    input            CLK,
    input            inc,
    input            MBR_w,
    input      [7:0] MBR_in,
    output reg [7:0] data
);
    always @(posedge CLK, negedge RST_N)
        if (!RST_N) data <= 0;
        else begin
```

```
            if (inc) data <= data + 1;
            if (MBR_w) data <= MBR_in;
        end
endmodule

module mod_IR (
    input           CLK,
    input           MBR_w,
    input      [7:0] MBR_in,
    output reg [7:0] data
);
    always @(posedge CLK) begin
        if (MBR_w) data <= MBR_in;
    end
endmodule

module mod_MR (
    input           CLK,
    input           ALU_w,
    input      [15:0] ALU_in,
    output reg [15:0] data
);
    always @(posedge CLK) begin
        if (ALU_w) data <= ALU_in;
    end
endmodule

module mod_BR (
    input           CLK,
    input           MBR_w,
    input      [15:0] MBR_in,
    input           one,
    output reg [15:0] data
);
    always @(posedge CLK) begin
        if (MBR_w) data <= MBR_in;
        if (one) data <= 1;
    end
endmodule

module mod_DR (
    input           CLK,
```

```
    input            disp,
    input            trap,
    input      [15:0] MR_in,
    input      [15:0] ACC_in,
    input      [ 7:0] PC_in,
    input      [ 7:0] IR_in,
    output reg [31:0] data
);
    always @(posedge CLK) begin
        if (disp) data <= {MR_in, ACC_in};
        if (trap) data <= {PC_in, IR_in, ACC_in};
    end
endmodule

module mod_disp (
    input            CLK,
    input      [31:0] DR,
    output     [ 7:0] anode,   // 低电平驱动
    output reg [ 6:0] segment  // 低电平点亮，实际 wire
);

    reg [16:0] div;  // 扫描，763Hz
    always @(posedge CLK) div <= div + 1;

    wire [2:0] current = div[16:14];

    assign anode = ~(1 << current);

    always @(*) begin
        case (DR[current*4+:4])
            'h0: segment = 7'b0000001;
            'h1: segment = 7'b1001111;
            'h2: segment = 7'b0010010;
            'h3: segment = 7'b0000110;
            'h4: segment = 7'b1001100;
            'h5: segment = 7'b0100100;
            'h6: segment = 7'b0100000;
            'h7: segment = 7'b0001111;
            'h8: segment = 7'b0000000;
            'h9: segment = 7'b0000100;
            'hA: segment = 7'b0001000;
            'hB: segment = 7'b1100000;
```

```
            'hC: segment = 7'b0110001;
            'hD: segment = 7'b1000010;
            'hE: segment = 7'b0110000;
            'hF: segment = 7'b0111000;
        endcase
    end
endmodule

module mod_button (
    input  CLK,
    input  in,
    output out
);
    reg [20:0] div;
    reg state;
    always @(posedge CLK)
        if (in == state) div <= 0;
        else begin
            div <= div + 1;
            if (&div) state <= in;
        end
    reg shift;
    always @(posedge CLK) shift <= state;
    assign out = state && !shift;
endmodule
```