夏季学期综合实验报告

林宝诚 无 98 2019080030

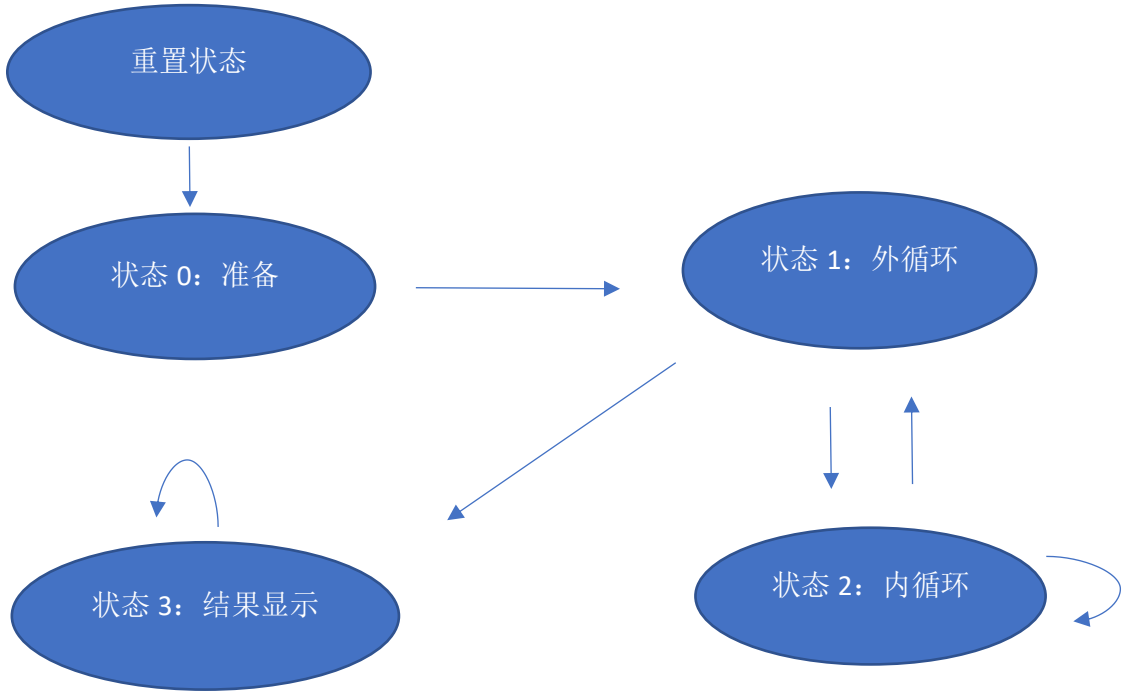
实验内容：使用数字逻辑电路求解背包问题，并和在多周期 MIPS 处理器上求解同一问题作比较。

数字逻辑电路（ASIC）设计思路：

已知背包问题的算法使用了两个嵌套循环，具体代码见图【1】。设计过程中，不能直接使用循环语句进行设计，毕竟在硬件中并没有循环的概念。为了解决此问题，一种直接的想法是将个别循环想象成状态，循环过程想象成状态转移，就能够使用有限状态机实现相应功能（见图【2】）。

```
int knapsack_dp_loop(int item_num, Item* item_list, int knapsack_capacity){
    int cache_ptr[MAX_CAPACITY + 1] = {0};
    for(int i = 0; i < item_num; ++i){
        int weight = item_list[i].weight;
        int val = item_list[i].value;
        for(int j = knapsack_capacity; j >= 0; --j){
            if(j >= weight){
                cache_ptr[j] =
                    (cache_ptr[j] > cache_ptr[j - weight] + val)?
                    cache_ptr[j]:
                    cache_ptr[j - weight] + val;
            }
        }
    }
    return cache_ptr[knapsack_capacity];
```

图【1】背包问题算法



图【2】状态转移图

状态详情

1. 重置状态
   - 初始化所有变量（内外循环变量，数码管显示）
   - 设置数据（最大背包重量，物品数量，个别物品重量和价值）
   - 设当前状态为状态 0

2. 状态 0：准备
   - 设 i=0

   状态转移：下一状态固定为状态 1

3. 状态 1：外循环
   - 设 j=knapsack capacity（给定背包最大重量）
   - 设 weight=item_list[i].weight（第 i 个物品重量）
   - 设 val=item_list[i].value（第 i 个物品价值）
   - 设 i=i+1

   状态转移：

   判断依据为外循环变量 i 是否大于等于 itemNum（物品数量）

结果	下一状态
是	状态 3
否	状态 2

4. 状态 2：内循环
   - 设 j=j-1
   - 根据条件判断是否更新 cache_ptr[j] （储存结果的变量）

   状态转移：

   判断依据为内循环变量 j 是否等于-1（0xffffffff）

结果	下一状态
是	状态 1
否	状态 2

虽然理论上可以使用 j 小于 0 作为判断依据，但是在 Verilog 中，0xffffffff 是大于 0 的（逐 bits 比较的结果），因此使用等于-1 作为判断依据可以保证正确性，且更便捷。

5. 状态 3：结果显示
   - 将 cache_ptr[totalWeight]（最终结果）的低 16 位显示在七段数码管上

   状态转移：下一状态固定为状态 3、

*注：Verilog 代码中使用了无阻塞赋值，因此需要注意同一状态内的每行代码是**并行执行**的。

时钟分频

实验中将 FPGA 的 100MHz 主频分为 10Hz 和 1000Hz 两个频率

10Hz：实验验收使用的时钟频率，以清楚展现状态转移的过程

1000Hz：七段数码管的刷新频率

显示

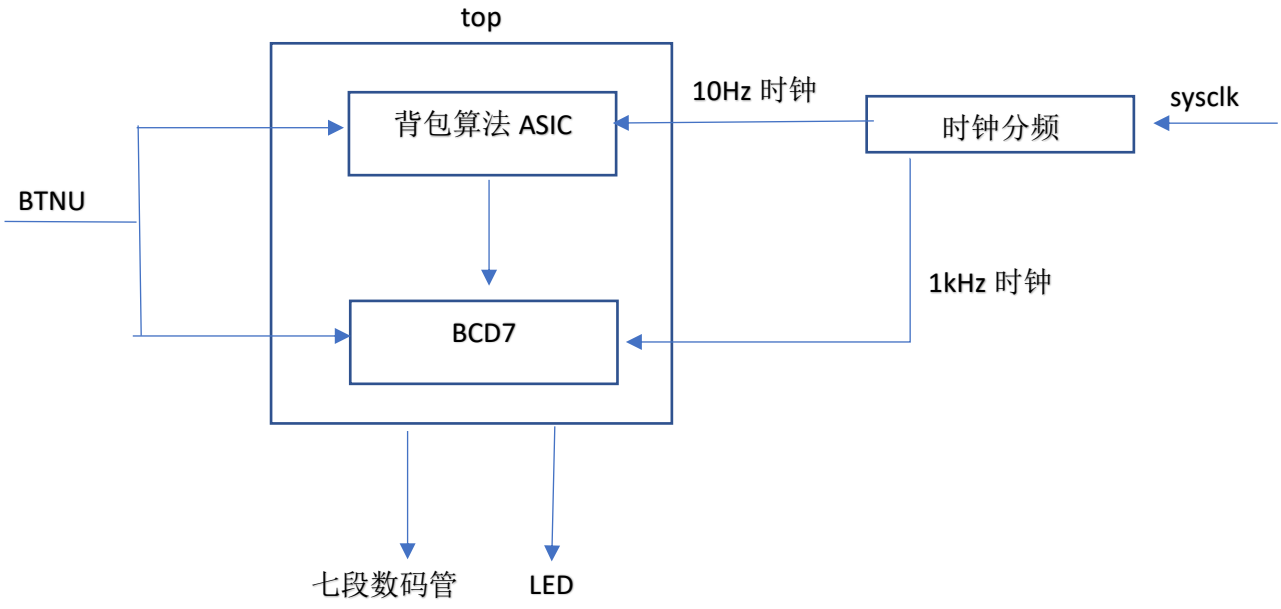
LED：显示当前状态

七段数码管：以 16 进制显示最终的计算结果

设计框图

sysclk：100MHz 系统时钟

BTNU：复位信号



图【3】设计框图

多周期处理器

由于理论课已完成 C 语言代码转换成 Mips 汇编语言的工作，因此仅需将汇编翻译成机器码（可利用 Mars 模拟器完成），并将机器码写入处理器的指令储存器即可。由于使用的算法有别，需要对原有代码进行些改动。

改动 1：添加新指令 bne

由于汇编中使用此前处理器并不支持的 bne 指令，故需要对处理器做些改动。这里引入新的控制信号如下：

控制信号 CondType	0	1
对应指令	beq	bne

根据原有的数据通路，已知 bne 在 Zero 为 0 时会进行分支，与 beq 正好相反，因此仅需要在原有的 beq 逻辑上增加一个非门和一个由上述 CondType 信号所控制的多路选择器即可完成对 bne 的支持。

改动 2：更改储存器中的指令和添加输入数据

最终在储存器中的指令机器码和输入数据如图【5】所示，数据的输入形式以图【6】为准。图【4】为翻译成机器码前的 Mips 指令。需要注意的是首两个 addi 指令的 Immediate 分别为 256 与 424。其中前者是因为处理器中预留了 64 个 32 位空间（0 至 63）作为指令的储存空间，从序号 64 的空间开始储存输入数据。由于设计中的输入地址是以字节为单位，故需要乘以 4，即 $64 \times 4 = 256$。值得一提的是，设计中预留了 42 个 32 位空间（64 至 105）作为输入数据储存空间。这时就能根据同样方式计算 s1（代表 cache_ptr 首地址）的值（$106 \times 4 = 424$）。

```
        addi $s0,$zero,256
        addi $s1,$zero,424
        lw $t0,0($s0)           # $t0 is knapsack capacity
        lw $t1,4($s0)           # $t1 is item_num
        addi $s0,$s0,8          # shift to address of the first weight
        add $t2,$zero,$zero     # $t2 is i

oloop:  slt $t3,$t2,$t1
        beq $t3,$zero,oexit
        lw $t4,0($s0)           # set $t4 as weight
        lw $t5,4($s0)           # set $t5 as value
        addi $s0,$s0,8          # shift to the next weight address
        addi $t2,$t2,1

        # begin inner loop
        add $t6,$t0,$zero       # $t6 is j

iloop:  slt $t3,$t6,$zero
        bne $t3,$zero,iexit
        slt $t3,$t6,$t4
        bne $t3,$zero,skpchk

        # find cache_ptr[j] and save in $t7
        sll $t9,$t6,2
        add $s1,$s1,$t9
        lw $t7,0($s1)
        sub $s1,$s1,$t9
```

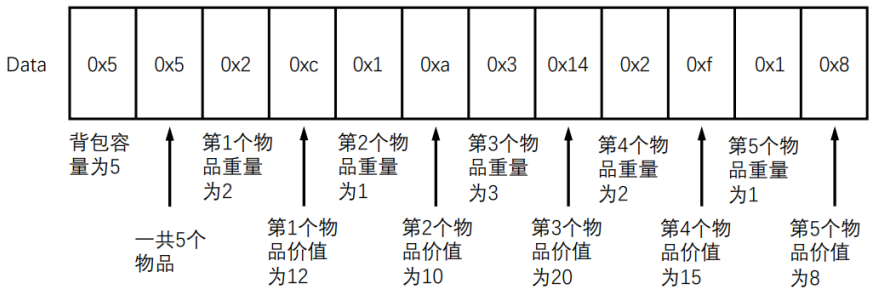
```
        # update $t8 = cache_ptr[j-weight]+val
        add $t8,$t8,$t5

        # if(cache_ptr[j]>cache_ptr[j-weight]+val)
        slt $t3,$t8,$t7
        bne $t3,$zero,skpchk

        # else update cache_ptr[j] in memory
        sll $t9,$t6,2
        add $s1,$s1,$t9
        sw  $t8,0($s1)
        sub $s1,$s1,$t9

skpchk: addi $t6,$t6,-1
        j iloop

iexit:  j oloop

oexit:  sll $t9,$t0,2
        add $s1,$s1,$t9
        lw $v0,0($s1)           # final result save in $v0

infinite: beq $zero, $zero, infinite
```

图【4】背包问题 Mips 指令

```
RAM_data[9'd0] <= 32'h2010_0100;    //addi
RAM_data[9'd1] <= 32'h2011_01a8;    //addi
RAM_data[9'd2] <= 32'h8e08_0000;    //lw
RAM_data[9'd3] <= 32'h8e09_0004;    //lw
RAM_data[9'd4] <= 32'h2210_0008;    //addi
RAM_data[9'd5] <= 32'h0000_5020;    //add

//outer loop
RAM_data[9'd6] <= 32'h0149_582a;    //slt
RAM_data[9'd7] <= 32'h1160_001c;    //beq
RAM_data[9'd8] <= 32'h8e0c_0000;    //lw
RAM_data[9'd9] <= 32'h8e0d_0004;    //lw
RAM_data[9'd10] <= 32'h2210_0008;   //addi
RAM_data[9'd11] <= 32'h214a_0001;   //addi
RAM_data[9'd12] <= 32'h0100_7020;   //add

//inner loop
RAM_data[9'd13] <= 32'h01c0_582a;   //slt
RAM_data[9'd14] <= 32'h1560_0014;   //bne
RAM_data[9'd15] <= 32'h01cc_582a;   //slt
RAM_data[9'd16] <= 32'h1560_0010;   //bne
RAM_data[9'd17] <= 32'h000e_c880;   //sll
RAM_data[9'd18] <= 32'h0239_8820;   //add
RAM_data[9'd19] <= 32'h8e2f_0000;   //lw
RAM_data[9'd20] <= 32'h0239_8822;   //sub
RAM_data[9'd21] <= 32'h01cc_5822;   //sub

RAM_data[9'd22] <= 32'h000b_c880;   //sll
RAM_data[9'd23] <= 32'h0239_8820;   //add
RAM_data[9'd24] <= 32'h8e38_0000;   //lw
RAM_data[9'd25] <= 32'h0239_8822;   //sub

RAM_data[9'd26] <= 32'h030d_c020;   //add
RAM_data[9'd27] <= 32'h030f_582a;   //slt
RAM_data[9'd28] <= 32'h1560_0004;   //bne
RAM_data[9'd29] <= 32'h000e_c880;   //sll
RAM_data[9'd30] <= 32'h0239_8820;   //add
RAM_data[9'd31] <= 32'hae38_0000;   //sw
RAM_data[9'd32] <= 32'h0239_8822;   //sub
RAM_data[9'd33] <= 32'h21ce_ffff;   //addi
RAM_data[9'd34] <= {6'h02, 26'd13}; //j
RAM_data[9'd35] <= {6'h02, 26'd6};  //j

RAM_data[9'd36] <= 32'h0008_c880;   //sll
RAM_data[9'd37] <= 32'h0239_8820;   //add
RAM_data[9'd38] <= 32'h8e22_0000;   //lw

//infinite loop
RAM_data[9'd39] <= 32'h1000_ffff;   //beq
```

```
//knapsack test set
RAM_data[9'd64]<=32'd20;
RAM_data[9'd65]<=32'd20;

RAM_data[9'd66]<=32'd2;
RAM_data[9'd68]<=32'd5;
RAM_data[9'd70]<=32'd10;
RAM_data[9'd72]<=32'd9;
RAM_data[9'd74]<=32'd3;
RAM_data[9'd76]<=32'd6;
RAM_data[9'd78]<=32'd2;
RAM_data[9'd80]<=32'd2;
RAM_data[9'd82]<=32'd6;
RAM_data[9'd84]<=32'd8;
RAM_data[9'd86]<=32'd2;
RAM_data[9'd88]<=32'd3;
RAM_data[9'd90]<=32'd3;
RAM_data[9'd92]<=32'd2;
RAM_data[9'd94]<=32'd9;
RAM_data[9'd96]<=32'd8;
RAM_data[9'd98]<=32'd2;
RAM_data[9'd100]<=32'd10;
RAM_data[9'd102]<=32'd8;
RAM_data[9'd104]<=32'd6;

RAM_data[9'd67]<=32'd8;
RAM_data[9'd69]<=32'd1;
RAM_data[9'd71]<=32'd5;
RAM_data[9'd73]<=32'd9;
RAM_data[9'd75]<=32'd5;
RAM_data[9'd77]<=32'd6;
RAM_data[9'd79]<=32'd8;
RAM_data[9'd81]<=32'd2;
RAM_data[9'd83]<=32'd3;
RAM_data[9'd85]<=32'd7;
RAM_data[9'd87]<=32'd5;
RAM_data[9'd89]<=32'd4;
RAM_data[9'd91]<=32'd3;
RAM_data[9'd93]<=32'd7;
RAM_data[9'd95]<=32'd6;
RAM_data[9'd97]<=32'd7;
RAM_data[9'd99]<=32'd9;
RAM_data[9'd101]<=32'd3;
RAM_data[9'd103]<=32'd10;
RAM_data[9'd105]<=32'd5;
```

图【5】机器码与输入数据



图【6】输入数据格式

仿真分析

测试数据：

序号	重量	价值
1	2	8
2	5	1
3	10	5
4	9	9
5	3	5
6	6	6
7	2	8
8	2	2
9	6	3
10	8	7
11	2	5
12	3	4
13	3	3
14	2	7
15	9	6
16	8	7
17	2	9
18	10	3
19	8	10
20	6	5

最大背包重量：20

物品数量：20

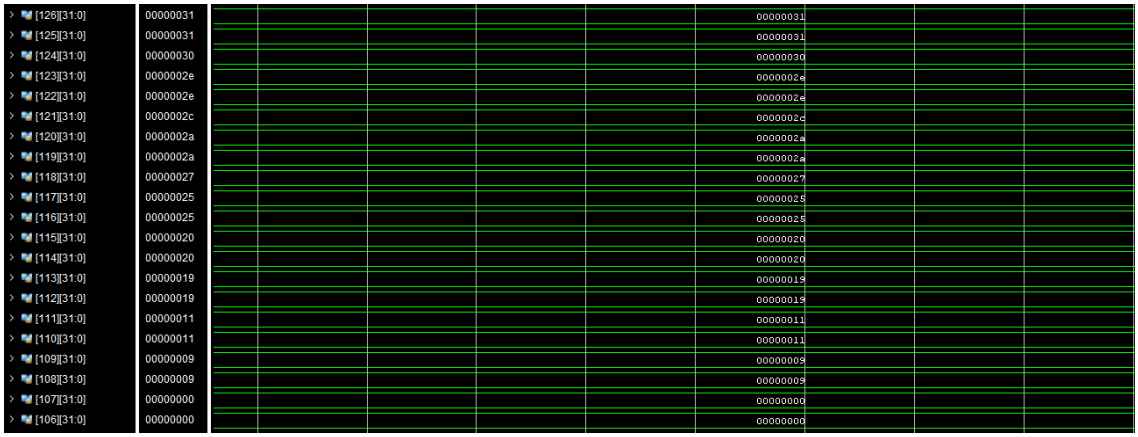
算法结果：$0x00000031 = (49)_{10}$

*注：之后的资源消耗与时序性能是根据此测试数据进行仿真和计算。实际上，算法的时间复杂度为$O(nw)$，空间复杂度为$O(w)$，n 为物品数量，w 为最大背包重量。因此若更改数据，资源消耗和程序运行时间等都会有所改变。

多周期处理器仿真结果

从图【7】中，易见最终 v0 的值为 0x00000031，与预想结果相符。图【8】中则显示了算法的中间结果，得以验证算法有顺利运行。

图【7】多周期处理器算法仿真结果

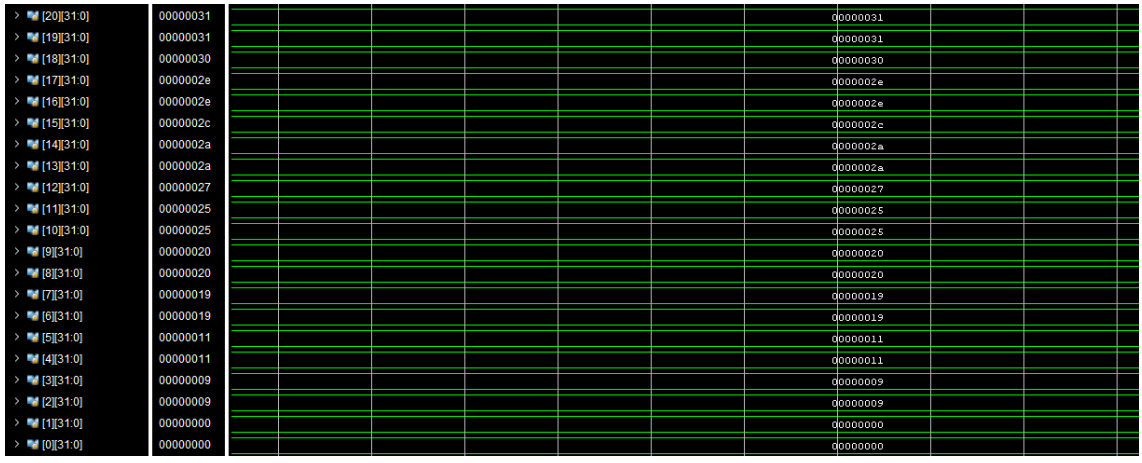


图【8】多周期处理器算法仿真中间结果

ASIC 仿真结果

同样地，图【9】清楚显示了正确的计算结果，同时图【10】所获取的中间结果与图【8】相同，因此可以得知 ASIC 与多周期处理器分别用不同的设计思想实现同样的算法。

图【9】ASIC 算法仿真结果



图【10】ASIC 算法仿真中间结果

FPGA 资源消耗

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Bonded IOB (106)	BUFGCTRL (32)
top	1582	2068	319	10	837	1582	112	22	2
ASIC_1 (ASIC)	1538	2015	319	10	812	1538	83	0	0
BCD7 (BCD7)	3	2	0	0	1	3	2	0	0
clk_gen (clk_gen)	14	18	0	0	8	14	10	0	0
clk_gen_10Hz (clk_ge...	27	33	0	0	16	27	17	0	0

图【11】ASIC 资源消耗情况

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Bonded IOB (106)	BUFGCTRL (32)
top	7272	17599	2488	1056	6782	7272	1107	24	2
BCD7 (BCD7)	11	2	0	0	6	11	2	0	0
clk_gen (clk_gen)	22	18	0	0	10	22	17	0	1
MultiCycleCPU_1 (Mult...	7223	17579	2488	1056	6773	7223	1088	0	0

图【12】多周期处理器资源消耗情况

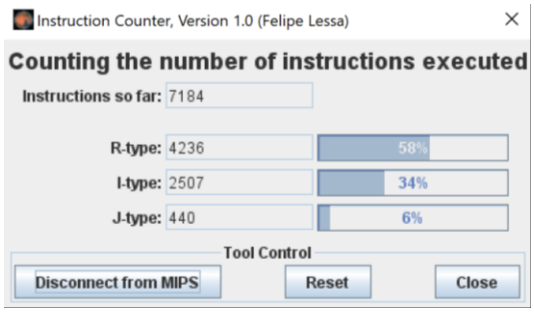
分析：不难发现多周期处理器所消耗的资源远超 ASIC 所消耗的资源，因为多周期处理器是由许多不同模块，如储存器、控制器、寄存器堆等组成，而 ASIC 的实现仅仅是个较简单的有限状态机。值得一提的是，即使多周期处理器资源消耗较多，但无可否认，其更加通用，仅需要更改指令就能够实现其他算法，而 ASIC 显然不具备此特性。

时序性能

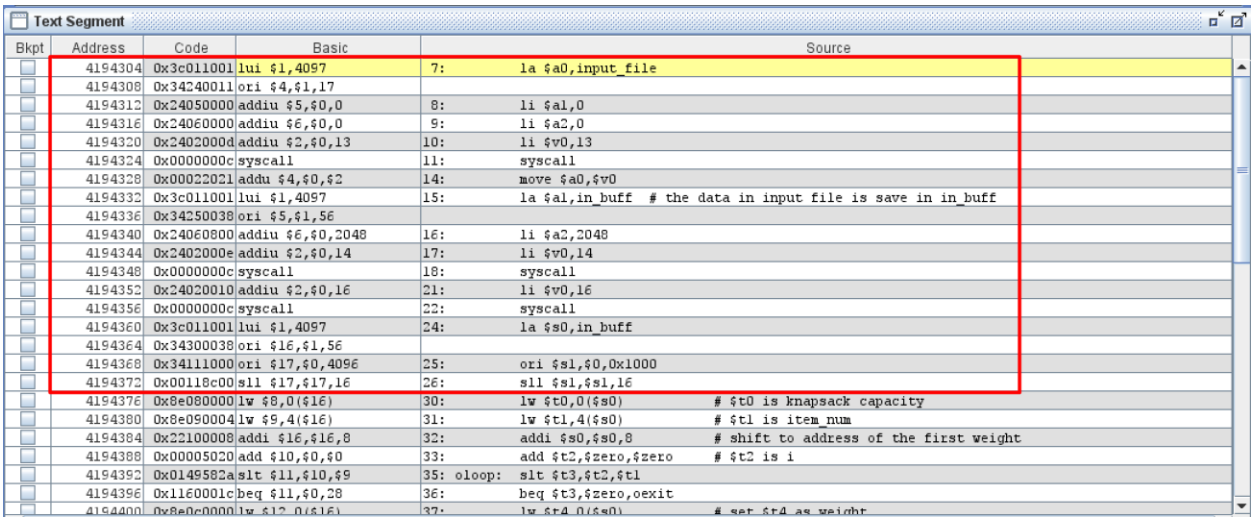
假设时序分析使用的测试时钟频率为 100MHz

1. 多周期处理器：

使用 Mars 模拟器计算的总指令数量为 7184（图【13】。但是，考虑到在 Mars 模拟器中使用了文件读写操作，因此与处理器中使用的指令有所差别（图【14】红框部分）。通过计算比较，可得实际在处理器运行的指令数为$7184 - 16 = 7168$



图【13】指令数量

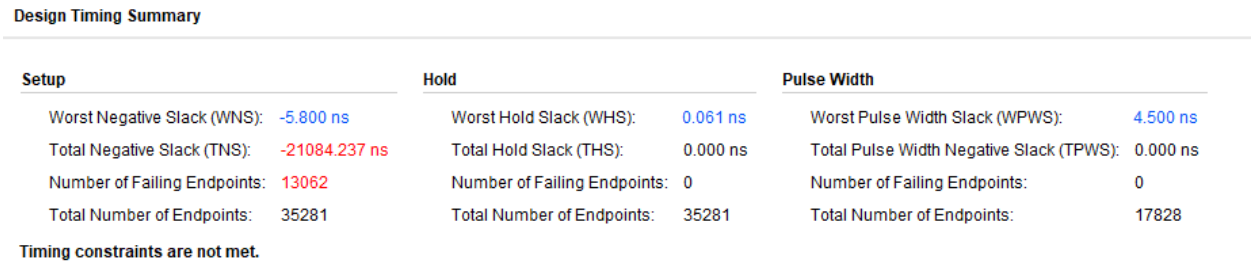


Text Segment

Bkpt	Address	Code	Basic		Source
	4194304	0x3c011001	lui $1,4097	7:	la $a0,input_file
	4194308	0x34240011	ori $4,$1,17		
	4194312	0x24050000	addiu $5,$0,0	8:	li $a1,0
	4194316	0x24060000	addiu $6,$0,0	9:	li $a2,0
	4194320	0x2402000d	addiu $2,$0,13	10:	li $v0,13
	4194324	0x0000000c	syscall	11:	syscall
	4194328	0x00022021	addu $4,$0,$2	14:	move $a0,$v0
	4194332	0x3c011001	lui $1,4097	15:	la $a1,in_buff # the data in input file is save in in_buff
	4194336	0x34250038	ori $5,$1,56		
	4194340	0x24060800	addiu $6,$0,2048	16:	li $a2,2048
	4194344	0x2402000e	addiu $2,$0,14	17:	li $v0,14
	4194348	0x0000000c	syscall	18:	syscall
	4194352	0x24020010	addiu $2,$0,16	21:	li $v0,16
	4194356	0x0000000c	syscall	22:	syscall
	4194360	0x3c011001	lui $1,4097	24:	la $s0,in_buff
	4194364	0x34300038	ori $16,$1,56		
	4194368	0x34111000	ori $17,$0,4096	25:	ori $s1,$0,0x1000
	4194372	0x00118c00	sll $17,$17,16	26:	sll $s1,$s1,16
	4194376	0x8e080000	lw $8,0($16)	30:	lw $t0,0($s0) # $t0 is knapsack capacity
	4194380	0x8e090004	lw $9,4($16)	31:	lw $t1,4($s0) # $t1 is item_num
	4194384	0x22100008	addi $16,$16,8	32:	addi $s0,$s0,8 # shift to address of the first weight
	4194388	0x00005020	add $10,$0,$0	33:	add $t2,$zero,$zero # $t2 is i
	4194392	0x0149582a	slt $11,$10,$9	35: oloop:	slt $t3,$t2,$t1
	4194396	0x1160001c	beq $11,$0,28	36:	beq $t3,$zero,oexit
	4194400	0x8e0c0000	lw $12,0($16)	37:	lw $t4,0($s0) # set $t4 as weight

图【14】算法中的上半部分 Mips 指令

*注：处理器最后一个死循环 beq 指令并未计算在指令数量内。



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-5.800 ns	Worst Hold Slack (WHS):	0.061 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	-21084.237 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	13062	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	35281	Total Number of Endpoints:	35281	Total Number of Endpoints:	17828

Timing constraints are not met.

图【15】多周期处理器时序性能总结

因此，最短时钟周期为 $10ns + 5.8ns = 15.8ns$，对应最高工作频率$\frac{1}{15.8ns} = 63.29MHz$。这里计算出的最高时钟频率与之前实验的有些差别，预计是因为我对处理器做了些改动（添加了 bne 指令，增加内存容量，更改算法）

根据行为仿真：

程序开始时间：0ns

程序结束时间（执行完最后一个 lw 指令后）：2,770,400ns

程序完成运行所需时钟周期数$= \frac{2770400ns - 0ns}{100ns} = 27704$

最短执行时间$= 27704 \times 15.8ns = 437.72\mu s$

平均 1 条指令所需的周期数$= \frac{27704}{7168} = 3.86$

平均每秒执行指令数目$= (3.86 \times 15.8ns)^{-1} =$ 每秒 $16.38M$条指令

2. ASIC：

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-18.222 ns	Worst Hold Slack (WHS):	0.175 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	-30501.646 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	2728	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4046	Total Number of Endpoints:	4046	Total Number of Endpoints:	2034

Timing constraints are not met.

图【16】ASIC 时序性能总结

因此，最短时钟周期为 $10ns + 18.222ns = 28.222ns$，对应最高工作频率$\frac{1}{15.323ns} = 35.43MHz$。

根据行为仿真：

程序开始时间：100ns

程序结束时间（执行第一次状态 3 后）：46300ns

程序完成运行所需时钟周期数$= \frac{46300ns - 100ns}{100ns} = 462$

最短执行时间$= 462 \times 28.222ns = 13.04\mu s$

分析：从上述的执行时间易见 ASIC 能够在更短时间解决相同的问题，毕竟与较复杂的处理器相比，其结构更加简单直接。需要紧记的是，虽然 ASIC 在此问题上性能较好，但其通用性较低，设计花费大（毕竟在其他问题上不能重复使用）。综上所述，在进行设计时，需要在通用性和性能之间做取舍。在解决更注重效率和性能的问题时，可以考虑使用 ASIC；在解决一般的问题时，使用处理器的总体效率会更好。

硬件调试情况

硬件运行结果与行为仿真结果相符（已通过验收），且能够根据 LED 判断出程序中的状态转移&指令执行是正常运行的。此外，也能够通过七段数码管的输出对结果进行检验。

思想体会

经过了两个学期（秋季和夏季）的实验课，我已对硬件设计有一定的了解。随着实验课的推进，我能够清楚感受到自己对 verilog 的熟悉度日益提升，且能够更合理地使用组合与时序逻辑解决问题。值得一提的是，当我现在回头去翻阅学期初期所进行的实验,我很容易就发现到我的代码有瑕疵，比如赋值语句中并没写明数字的进制（容易引起混淆），组合与时序逻辑的使用较不好（第二次实验）。上述发现能够证明我完成实验的能力有慢慢进步。另外，即使无法返校，我很高兴自己能够顺利且准时地完成两个学期的实验，让我不需要将实验延后至明年，毕竟我认为同时进

行理论课与实验课的学习，最终的效果和效率较好。在此非常感激老师和助教对线上同学的付出。再者，我认为实验课的设置也挺合理，实验时间充足，实验难度循序渐进，有适度的挑战性（不会难得逆天）。

实际上，我认为数字逻辑与处理器基础是门非常关键的课，只有掌握这门课，才能够了解数字系统的设计方法，同时将电子电路与软件编程之间的关系摸透。毋庸置疑，这门课让我的未来发展方向又多了一个选择。而作为一个与涉及工程问题的课程，单单学习理论是不足够的，需要累积“实战”经验才能够更好地吸收相关的知识，所以这门配套的实验课也是必不可少的。总结地说，实验课不仅能够加深理论知识的掌握，也能提升我作为未来工程师的能力，因此毫无疑问这门实验课令我收获满满！

关键代码与文件清单

汇编代码

- KnapsackProblem.asm：背包问题汇编代码（标记“#”号的部分为文件读的代码）

共同代码

- clk_gen_10Hz：10Hz 时钟分频代码
- clk_gen：1kHz 时钟分频代码
- BCD7：七段数码管代码

ASIC

- top.v：将所有模块连接在一起的顶层模块
- ASIC.v：背包问题算法的数字逻辑电路实现（有限状态机）
- test_asic.v：ASIC 仿真代码
- ASIC.xdc

多周期处理器

- MultiCycleCPU.v：将所有模块连接在一起的处理器顶层模块
- PC.v：程序计数器
- InstReg.v：指令寄存器
- InstAndDataMemory.v：储存器
- Controller.v：控制器
- RegisterFile.v：寄存器堆
- RegTemp：暂时寄存器
- ImmProcess.v：Immediate 处理器
- ALUControl.v：ALU 控制器
- ALU.v
- test_cpu.v：多周期处理器仿真代码
- multicycleCPU.xdc