Name: Solution

a) Translate the following code into Tiny8v1 assembly (5 points):

```
int *src_ptr = 0x.....
int *dest_ptr = 0x....
int I = 0;
int sum = 0;
for(int I = 0; I < 16; i++) {
   sum += src_ptr[2*i] * 20
   dest_ptr[i] = sum
}
```

```
; r0 holds ~~address of~~ src_ptr = 0xA0
; r1 holds ~~address of~~ dest_ptr = 0xC0
; r2 is uninitialized
; r3 initially holds I = 15
; acc is 0x45 initially
; YOUR CODE HERE

;Clear acc
SSD r1, 0
LSD r2, 0
ADS r2, -1
LOOP: LSD r2, r0, -2   ; Load source
ADS r2, 7              ; imm4 has range
ADS r2, 7              ; -8-7
ADS r2, 6
SSD r1, -1             ; store sum
BDD r3, LOOP
```

b) Your boss takes a look at Tiny8v1 and commends your initiative. However, she tells you that there is a huge functional omission which would make running the program given in part A impossible in practice. What common operation is impossible to perform with Tiny8v1? (3 points)

Loading constants / addresses to registers.

Long jumps to enter this function.

Both the operations may be required to set up the code the way part a requires

Name: Solution

c) With the realization of part b), you decide to fix the ISA for Tiny8v2. How would you add the operation from part b) without losing any original functionality? (3 points)

STP use 6 bits only (2 opcode, 2 rd, 2 delta)

Use these extra 2 bits to add functionality

e.g. LIR imm6 ; load immediate <<2 into register
JL imm6 ; jump immediate <<2

d) Your boss is happy with the capabilities of Tiny8v2. However, DRAM accesses are very costly in terms of energy for the prototype Tiny8v2 chips running the code from part a). She suggests that you examine your d-cache organization and minimize number of memory accesses without adding any instructions.

The cache specifications are:

- 32 byte capacity → 8 lines → 3 bit index
- 4 byte cache lines → 2 bit offset
- direct-mapped

8-3-2 = 3 bit tag

Propose a software optimization to minimize cache misses. (4 points)

src_ptr = 0xA0 (tag index offset) [101 000 00]

dest_ptr = 0xC0 (tag index offset) [110 000 00]

src_ptr & dest_ptr have same index

① Move source / dest pointer to non-aliasing addresses: eg. 0xA0, 0xB0 (reduce conflict misses)

② Overlap src/dest addresses (reduce cold misses)

③ Unroll loop and change load order

... as long as it reduces misses

Name: _____

2. MP

Testing and debugging are critical to processor development. You, as a student in ECE 411, just finished implementing RTL code for MP1, and have written some simple test code to test the load instructions.

The first instruction you want to test is LDR. Following is the test code. Note: keyword data8 puts an 8-byte word into memory with a little endian fashion.

```
ORIGIN 4x0000

LDR R4, R0, long9

HALT:
BRnzp HALT

long1: data8 4x12345678abcdefff
long2: data8 4x12345678abcdefff
long3: data8 4x12345678abcdefff
long4: data8 4x12345678abcdefff
long5: data8 4x12345678abcdefff
long6: data8 4x12345678abcdefff
long7: data8 4x12345678abcdefff
long8: data8 4x12345678abcdefff
long9: data8 4x12345678abcdefff
long10: data8 4x12345678abcdefff
```

a) The above test code, when executed, will cause an error due to hardware constraints. Identify what the bug is. (3 points)

LDR's offset is 6-bits. That means it can only load data within 32 words distance. 'long9' is too far away.

Name: _____

b) You changed the code to be

```
ORIGIN 4x0000

ADD R0, R0, 2
LDR R4, R0, long1

HALT:
BRnzp HALT
```

And the data segment stays the same. What is the value expected to be loaded to R4? (2 points)

0xabcd. (-1 for not understanding little endian -1 for ignoring R0).

(or -1 if LDR isn't loading a word (16bit))

The second instruction you want to test is LDI. Following is the test code.

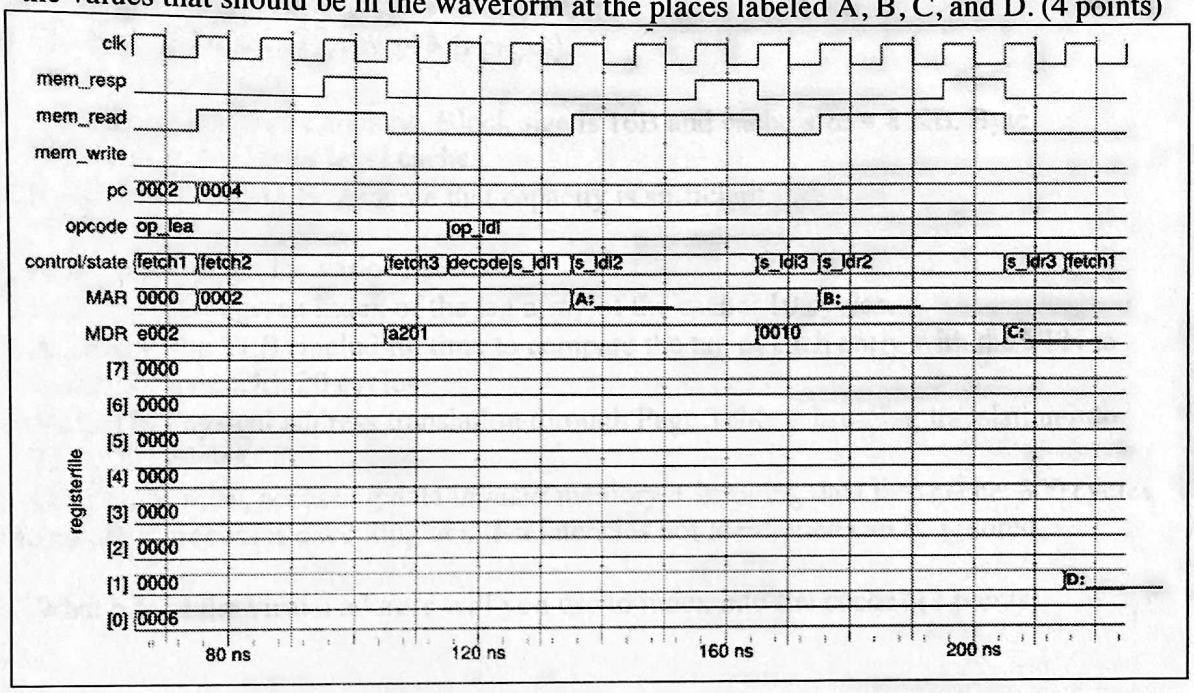
```
ORIGIN 4x0000
SEGMENT codeBlock:

LEA R0, dataBlock
LDI R1, R0, ptr2
HALT:
BRnzp HALT

SEGMENT dataBlock:
ptr1: data2 val1
ptr2: data2 val2
ptr3: data2 val3
val1: data2 4xbaad
val2: data2 4x600d
val3: data2 4xffee
```

Name: _____

c) The following waveform shows the LDI instruction executing for the above code. Fill in the values that should be in the waveform at the places labeled A, B, C, and D. (4 points)



A: 000A B: 0010

C: FFEE D: FFEE

d) You passed the unit test. But when you run a larger test code, LDI still causes error. What aspect of the LDI is not tested in the given code? Assume that all the registers values shown in the trace are set correctly (including the ones you filled in). (2 points)

1. LDI should set condition codes. This doesn't test it.

2. LDI should sign extend the offset. Wasn't tested.

(Answering one of the above two gets full credits).

Cache+VM

Consider a system with the following characteristics :

- 32 bit virtual address space (4 KB pages) .
- 1 GB main memory
- VIPT direct mapped L1 cache . Block size is 16B and cache size = 8KB . Byte addressable . No lower level caches .
- Fully associative TLB . Assume that capacity is sufficient .

These are the times taken for various accesses :

- Indexing into a given index of the tag array of the cache : 10 cycles .
- Accessing the TLB (including time to compare the tag of each entry with the VPN to search for a match) : 20 cycles
- Virtual to Physical address translation through Page Table + bringing translation into TLB : 200 cycles .
- On a cache miss , accessing data in main memory + bringing data into cache : 300 cycles

Assume no other process is executing and that latencies not mentioned can be ignored .

a) What bits of the virtual address will you use to index into the cache ?
Answer:
Bits [12:4], there are 8KB/16B = 512 sets in the cache, so we need 9 bits to index into the cache. Also , since it is byte addressable , bits[3:0] of the VA are used to access the corresponding bytes within the cache line.

b) What are the number of bits in the physical address and what is the bit range that the Physical Page number occupies ?
Since the size of main memory is 1 GB = 2^30 Bytes , we need 30 bits in the physical address. Since the page offset is going to be the lowest 12 bits of the physical address , bits[29:12] of the physical address indicates the PPN.

c) Consider the following sequence of **virtual address** accesses . Fill in the cycles required for each access and also fill in the accessed/created TLB and cache entries. . Assume that both the cache and TLB are initially empty.

Virtual address accessed	Time taken for access (cycles)
0x00000000	TLB miss , Cache Miss. Answer is 520 (or 540) cycles .
0x0000000f	TLB hit, Cache hit. Answer is 20 cycles
0x0001000a	TLB miss, Cache miss. Answer is 520 (or 540) cycles.
0x0000de00	TLB miss , Cache miss. Answer is 520 (or 540) cycles.

0x0001010d	TLB hit , Cache miss. Answer is 320 cycles.

1. Since the wording of this question is not clear in this respect, answers involving re-accessing the TLB or restarting the memory access on a TLB miss (after Page Table translation) , and answers involving accessing the TLB only once on a TLB miss have both been given points. Usually a TLB miss generates an exception, and a TLB handler executes, brings the translation into the TLB and restarts the instruction. However, the question did not specify a handler, so points are given even if the answer accounts only for a single TLB access on a TLB miss.

2. The main thing to note is that cache indexing and TLB access happen in parallel.

L1 cache - Fill in the accessed indices along with the tag stored at each index after the sequence of accesses. Values can be in decimal.

Index	Tag
0	20
480	4
16	20

Required translations are given in the **page table** . Assume all VPN-PPN mappings are valid for the executing process .

VPN	PPN
0	10
4	12
3	7
16	20
13	4
32	8

TLB - Fill in the VPN - PPN mappings created during the sequence of accesses .

VPN	PPN
0	10
13	4
16	20

d) Now assume 2 processes P1 and P2 run on the CPU (only one at a time) .

- They share a physical page with PPN x . They also have a producer-consumer relationship , where P1 writes into the physical page x at a known offset , and P2 reads from physical page x at that particular offset . Assume this offset is known to both processes .

1. In process P1 , virtual page with VPN (2*k1) is mapped to PPN x, and in process P2, virtual page with VPN (2*k2+1) is mapped to PPN x, where k1 and k2 are integers. Explain the problem that arises due to this mapping , using an example if necessary.

   Since the VPN mapped to x by P1 is even and the VPN mapped to x by P2 is odd, the same memory locations within physical page x would be present at 2 different sets within the cache. Since the cache is write back, the change made by P1 in the cache will not be seen by P2, since P2 will load the stale line containing the memory location directly from main memory.

2. Keeping the L1 cache VIPT , name 2 modifications that can be made to the cache in order to solve the problem encountered .

   - Increasing Associativity by 2 , Decreasing Cache Size by 2 , Increasing Cache Block Size by 2 ensure that no VPN bits are used for indexing the cache.
   - Can also make cache write through since the write-read happens once or flush on context switch (more general solution).

Name: Solution

fpadd	6
br	6.5
ld	9
st	9
total avg CPI	8.114
total latency	56800

longest critical path = 20ns
⇒ clock period = 20ns

$\frac{6 \cdot 20}{350} + \frac{6.5 \cdot 100}{350} + \frac{9 \cdot (150+80)}{350} = 8.114$

$8.114 \frac{\text{cycles}}{\text{instruction}} \cdot 20\text{ns} \cdot 350 \frac{\text{instructions}}{\text{program}}$

$= 56800\text{ns/program}$

b) You are able to change the architecture by splitting any of the control states into two parts, each with 60% of the latency of the original control state. Describe the changes you would make to the state machine of this architecture in order to decrease total latency of the profiled program. (3 points)

Split fpadd state into two states with critical path of 12 ns.

c) Splitting the control states necessarily increases CPI. Without redoing the CPI and latency calculations, give a reason why running the profiled program on this new architecture will decrease total latency. (2 points)

Only CPI of fp add was increased
Only 20 of 350 instructions are fp add
Benefits of increasing frequency outweigh making few instructions take an extra cycle

Name: Solution

d) Calculate the new CPI for each instruction (4 points)

fpadd	7
br	6.5
ld	9
st	9
total avg CPI	8.17
total latency	42900

$\frac{7 \cdot 20}{350} + \cdots = 8.17$

new clock period = 15 ns

$8.17 \cdot 15 \cdot 350 = 42900$

e) Calculate the speedup of the new architecture vs. the old architecture. (2 points)

$\frac{56800}{42900} = 1.324\times$ speed up