CSE 141L
Lab 3
Zhisheng Liu A123991234
Amal Alhaidari A98069538

Notes on using the assembler:

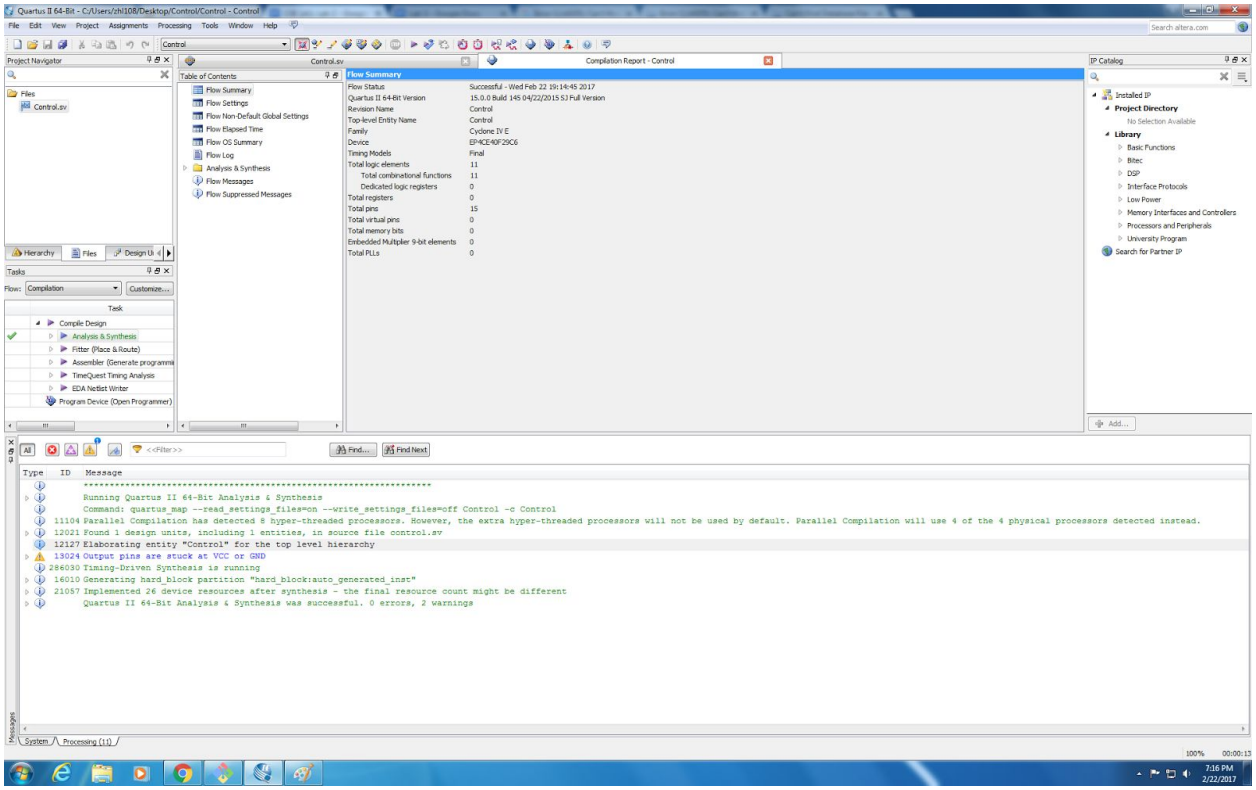
1. The assembler is written in python, included in the turned-in zip file as assembler.py
2. A third-party library "bitstring" is required to run the assembler, install it using pip:
   ```
   pip install bitstring
   ```
3. usage:
   ```
   python assembler.py <assembly_code_input.asm> <machine_code_output.txt>
   ```
4. Note: any other format of input than the format in 3 will be rejected and warned with the correct usage
5. Some other nice thing about our assembler:
   a. It is not hard coded (for the branch), so changing the content of assembly code does not require change of the assembler
   b. It will detect some basic syntax error such as
      i. non-existing operation
      ii. non-existing register arguments
      iii. incorrect instruction format
6. Only supports comments lead by "#"
7. Some limitations:
   a. does not support consecutive labels, example:
      ```
      add r1
      LABEL_1:
      LABEL_2:
      add r2
      ```

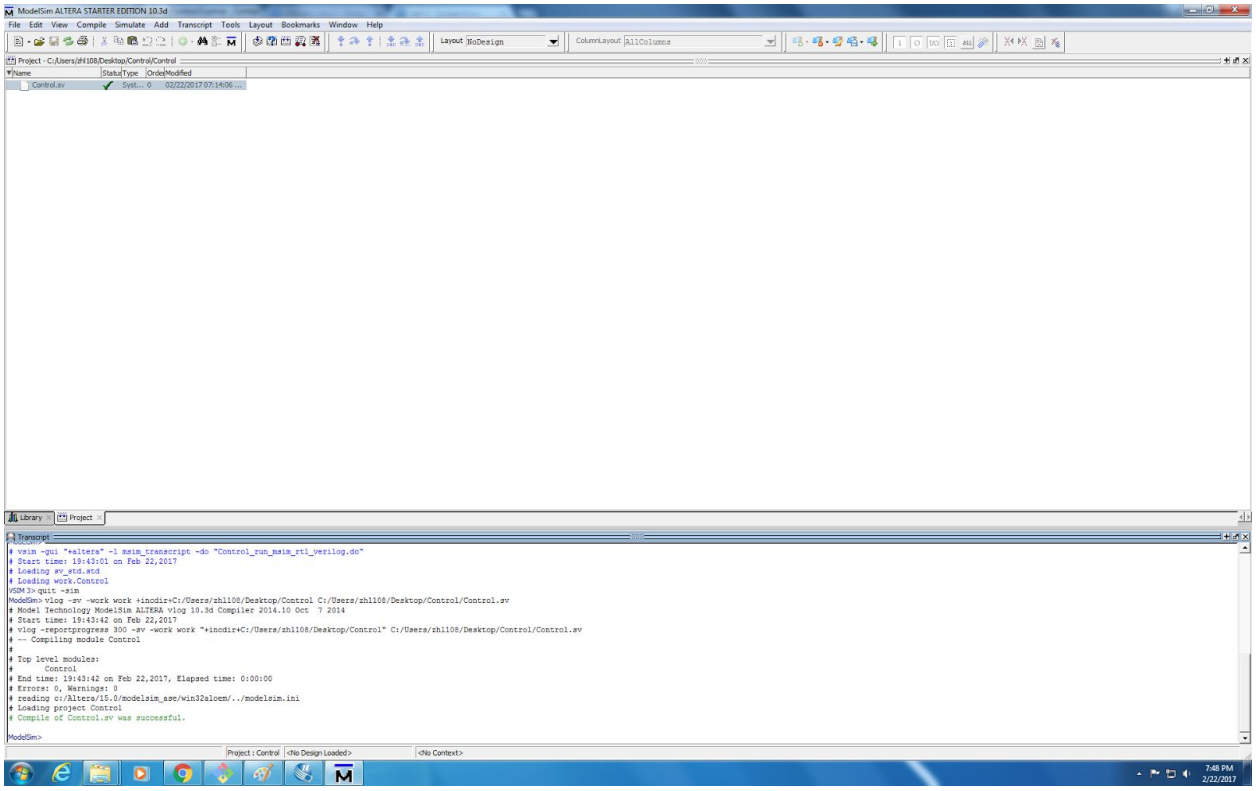
1. Brief introduction of module connection

We implement data memory module and control module in this lab, and connect them with our previous implementation (register files, fetching unit, ALU and lookup table). We first set up a top level system verilog file and works as the overall processor, and instantiates all modules on the processor (TopLevel.sv). After the instantiations, we declared a bunch of wire on the processor and drive them between different ports of different modules, connecting corresponding input/output by names. Since we apply a one bit overflow register in our design, we declare it as a "logic" on the top level, and set up the correct relation between the overflow register and ALU. Which more, we use two multiplexor on our top level for deciding input of register file and second input of ALU, in the form of (condition? a: b). No compilation error were reported in Quartus and Modelsim.

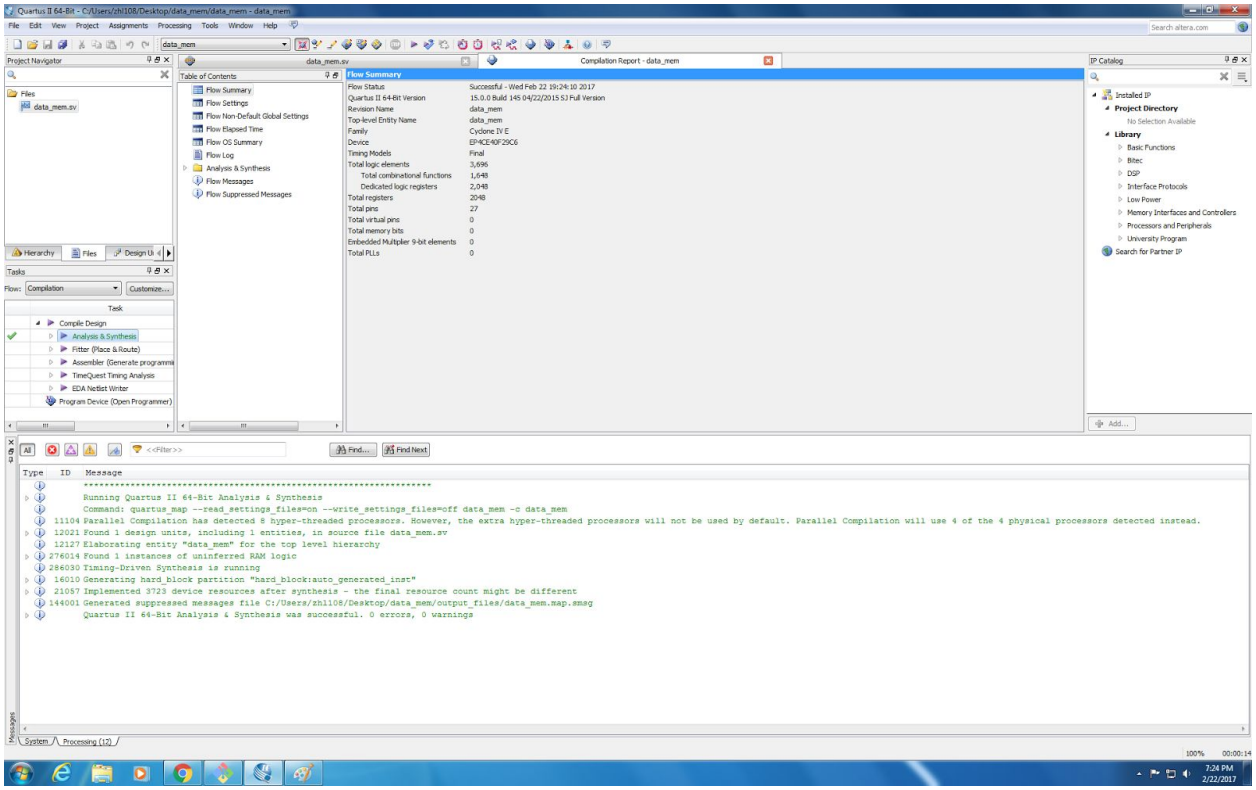
2. Screenshots of compilation

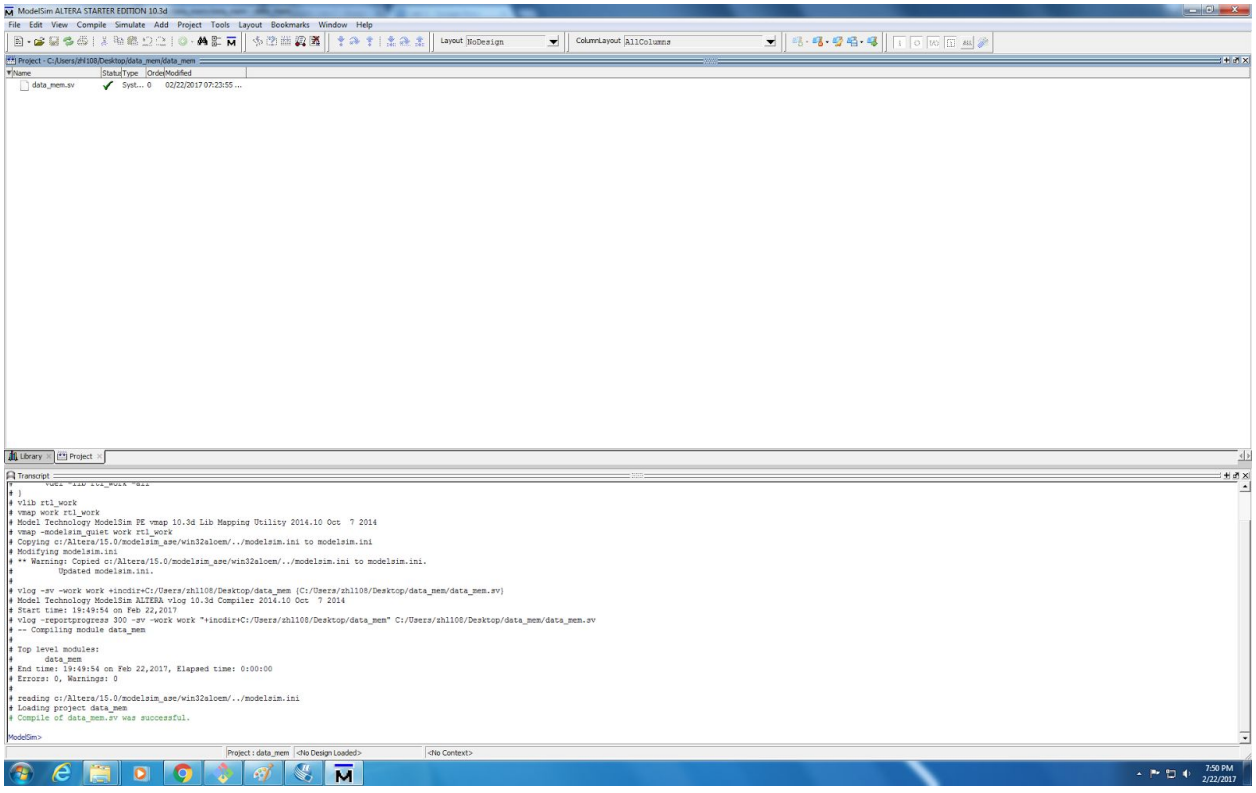
Control.sv is compiled successfully in both Quartus and ModelSim


Quartus II 64-Bit - C:/Users/zhl108/Desktop/Control/Control - Control
Flow Summary
Flow Status Successful - Wed Feb 22 19:14:45 2017
Quartus II 64-Bit Version 15.0.0 Build 145 04/22/2015 SJ Full Version
Revision Name Control
Top-level Entity Name Control
Family Cyclone IV E
Device EP4CE40F29C6
Timing Models Final
Total logic elements 11
Total combinational functions 11
Dedicated logic registers 0
Total registers 0
Total pins 15
Total virtual pins 0
Total memory bits 0
Embedded Multiplier 9-bit elements 0
Total PLLs 0
Compilation Report - Control
Type ID Message
Running Quartus II 64-Bit Analysis & Synthesis
Command: quartus_map --read_settings_files=on --write_settings_files=off Control -c Control
11104 Parallel Compilation has detected 8 hyper-threaded processors. However, the extra hyper-threaded processors will not be used by default. Parallel Compilation will use 4 of the 4 physical processors detected instead.
12021 Found 1 design units, including 1 entities, in source file control.sv
12127 Elaborating entity "Control" for the top level hierarchy
13024 Output pins are stuck at VCC or GND
286030 Timing-Driven Synthesis is running
16010 Generating hard_block partition "hard_block:auto_generated_inst"
21057 Implemented 26 device resources after synthesis - the final resource count might be different
Quartus II 64-Bit Analysis & Synthesis was successful. 0 errors, 2 warnings

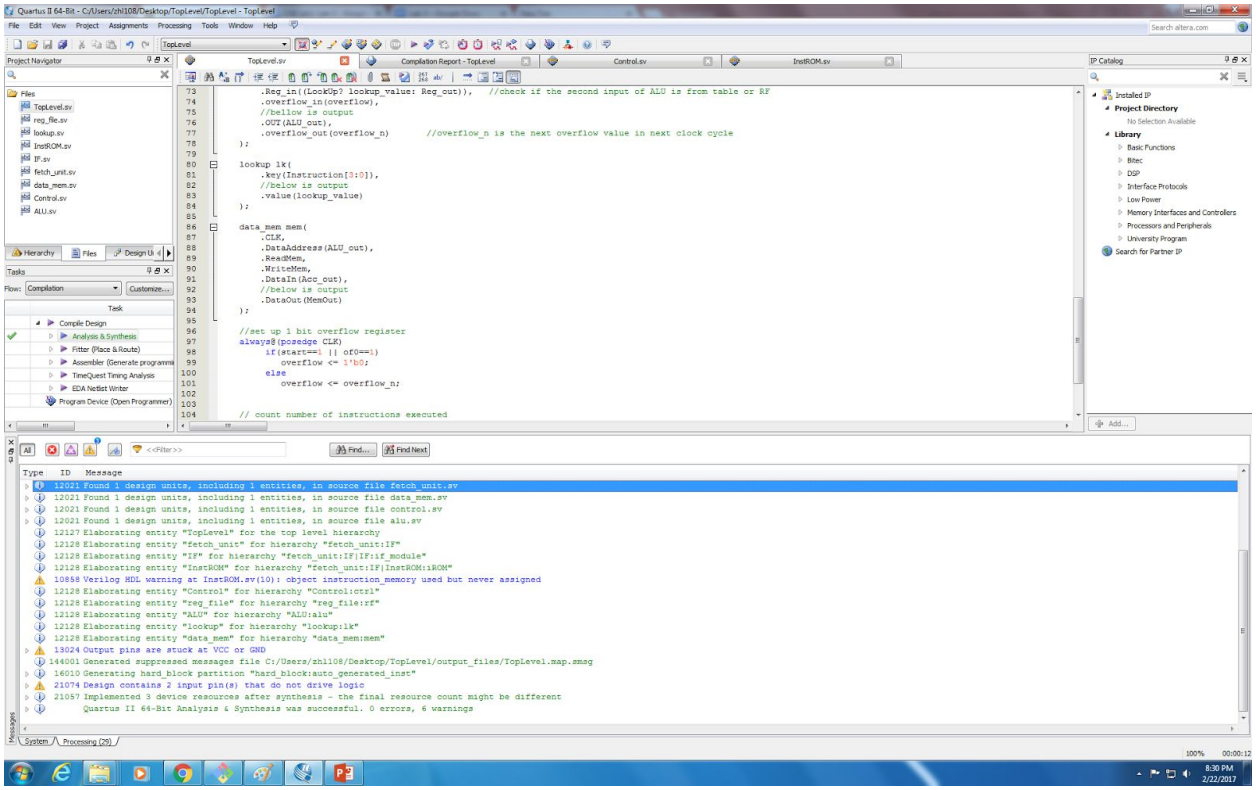

ModelSim ALTERA STARTER EDITION 10.3d
vsim -gui "+altera" -l msim_transcript -do "Control_run_msim_rtl_verilog.do"
Start time: 19:43:01 on Feb 22,2017
Loading sv_std.std
Loading work.Control
VSIM 3> quit -sim
ModelSim> vlog -sv -work work +incdir+C:/Users/zhl108/Desktop/Control C:/Users/zhl108/Desktop/Control/Control.sv
Model Technology ModelSim ALTERA vlog 10.3d Compiler 2014.10 Oct 7 2014
Start time: 19:43:42 on Feb 22,2017
vlog -reportprogress 300 -sv -work work "+incdir+C:/Users/zhl108/Desktop/Control" C:/Users/zhl108/Desktop/Control/Control.sv
-- Compiling module Control
#
Top level modules:
Control
End time: 19:43:42 on Feb 22,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
reading c:/Altera/15.0/modelsim_ase/win32aloem/../modelsim.ini
Loading project Control
Compile of Control.sv was successful.
ModelSim>

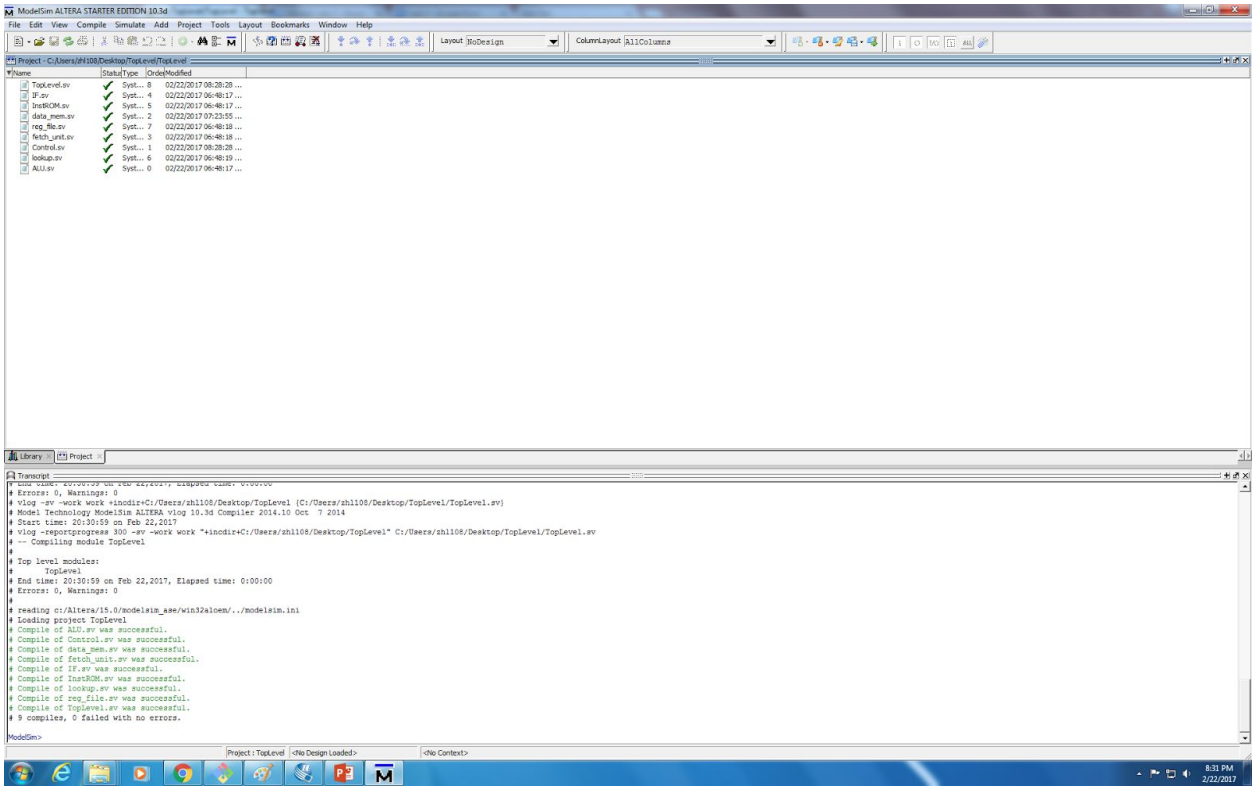
data_mem.sv is compiled successfully in both Quartus and ModelSim





TopLevel.sv is compiled successfully in both Quartus and ModelSim





3. Assembly Code

17: product

```
# r0 = accumulator
# r1 = a1
# r2 = a2
# r3 = b1
# r4 = b2
# r5 = s1
# r6 = s2
# r7 = s3
# r8 = s4
# r9 = upper bound 8  /sign_a  /temp
# r10= loop counter i /sign_b
# r11= isNeg          /
# r12= sum_high
# r13= sum_low
# r14= partial_high
# r15= partial_low
# *r16 = overlow_bit (1 bit register)

   #load operand a1,a2,b1,b2
```

```
    lookup 1
    load r0     #acc = mem[1]
    put r1      #r1=a1

    lookup 2
    load r0
    put r2      #r2=a2

    lookup 2
    put r9
    lookup 1
    add r9      #acc = 1+r9 = 1+2 = 3
    load r0     #acc = mem[3]
    put r3      #r3=b1

    lookup 2
    add r0      #acc = r0 = 2+2 = 4
    load r0     #acc = mem[4]
    put r4      #r4=b2

    #declare sign_a, sign_b
    lookup 5
    put r12     #use r12(sum_high) to temp record shift amount
    lookup 2
    add r12
    put r12     #r12 = 2+5 = 7
    take r1
    shr r12
    put r9      #sign_a = r9

    take r3
    shr r12
    put r10     #sign_b = r10


#   complex condition unrolling
#   if((r9==1 && r10==0) || (r9==0 && r10==1))
#
#   isNeg=0
#   if(r9==1)
#   {
#       if(r10==0)
#           isNeg=1;
```

```
#   }
#   if(all==0)
#   {
#      if(r9==0)
#      {
#         if(r10==1)
#            isNeg=1;
#      }
#   }
#

   lookup 0
   put r11        #initialize: isNeg=0

   lookup 1
   eql r9         #check if r9==1, if yes, acc=1
   b0 S_1

   # case: r9==1
   lookup 0
   eql r10
   b0 S_1
   lookup 1       #r9==1 and r10==0
   put r11        #isNeg=1

   lookup 0
   b0 S_2         #jump over case r9==0 since it's examined before

S_1:
   # case: r9==0
   lookup 1
   eql r10        #check if r10==1, if yes, acc=1
   eql r9
   b0 S_2
   # r9==0 and r10==1
   lookup 1
   put r11        #isNeg=1

S_2:

   #do 2's complement on A and B
   #r1=a1, r2=a2, r3=b1, r4=b2
   #check if r9(sign_a)==1, if yes, a1<0, acc=1
```

```
    lookup 1
    eql r9
    b0 S_3        #if acc=0, means a1>=0, no need for 2's compliment

    take r1       #a1 = ~a1
    nand r0       #acc = a1 nand a1 = not a1
    put r1        #store flipped a1

    take r2       #a2 = ~a2
    nand r0
    put r2

    lookup 15     #acc = -1
    eql r2        #if(a2 == -1) if yes, acc=1, else acc=0 and jump out scope
    b0 S_4
    lookup 0
    put r2        #a2=0;
    lookup 1
    add r1        #a1=a1+1
    put r1
    lookup 0      #jump over the else case
    b0 S_3
S_4:
    lookup 1
    add r2        #a2=a2+1
    put r2        #store a2
S_3:

    #r1=a1, r2=a2, r3=b1, r4=b2
    #check if r10(sign_b)==1, if yes, b1<0
    lookup 1
    eql r10
    b0 S_5        #if acc==0, means b1(r3)>0, no need for 2's compliment

    take r3       #b1 = ~b1
    nand r0       #acc = b1 nand b1 = ~b1
    put r3        #store b1(r3)

    take r4       #negate b2 and store it
    nand r0
    put r4

    lookup 15     #acc = -1
```

```
    eql r4          #if(b2 == -1) if yes, acc=1, else acc=0 and jump out of scope
    b0 S_6
    lookup 0
    put r4          #b2=0
    lookup 1
    add r3          #b1=b1+1
    put r3

    lookup 0        #jump over the else case
    b0 S_5
S_6:
    lookup 1        #b2=b2+1
    add r4
    put r4
S_5:

    #set s1~s4=0
    lookup 0
    put r5
    put r6
    put r7
    put r8


    #a2xb2 (r2xr4)--------------------------------
    #reset partial sum to zero
    put r12
    put r13
    put r14
    put r15

    put r10         #i=0
A2B2:

    #mask=1 << i
    lookup 1    #acc=0000_0001
    shl r10     #acc = acc << i
            #acc(mask) and b2(r4)
    nand r4
    nand r0     #create an and gate: acc = b_bit = b2 & mask
            #acc = b2 & mask

    b0 A2B2_S   #if acc==0, means the digit i of b2 is zero, no partial product yield
```

```
    take r2     #acc = a2
    shl r10     #acc = a2 << i
    put r15     #save partial_low

    lookup 8    #acc = 8
    sub r10     #acc = 8-i
    put r9      #r9 = 8-i
    take r2     #acc = a2
    shr r9      #acc = a2 >> 8-i
    put r14     #save partial_high

    of0         #TODO: new instruction - set overlow_bit to zero
    take r13    #acc = sum_low
    add r15
    put r13     #sum_low = sum_low + partial_low

    take r12    #acc = sum_high
    add r14
    put r12     #sum_high = sum_high + partial_hight
    of0

A2B2_S:

    #check counter at the end of loop
    lookup 1
    add r10
    put r10        #i++

    lookup 8
    put r9         #r9 is the upper limit of loop, r9=8
    take r10       #acc = i
    lsn r9         #check if i<8, if yes keep looping, acc=1
    b0 A2B2_OUT

    lookup 0
    b0 A2B2

A2B2_OUT:

    #s3 = s3 + sum_high = r7 + r12
    #s4 = s4 + sum_low = r8 + r13
    of0
```

```
    take r8
    add r13
    put r8          #s4 += sum_low

    take r7
    add r12
    put r7          #s3 += sum_hight
    of0

    #a1xb2 (r1xr4)-------------------------------
    #reset partial sum to zero
    lookup 0
    put r12
    put r13
    put r14
    put r15

    put r10         #i=0
A1B2:

    #r1 x r4
    #mask=1 << i
    lookup 1
    shl r10
            #acc and b2(r4)
    nand r4
    nand r0     #create an and gate: acc = b_bit = b2 & mask

    b0 A1B2_S
    take r1
    shl r10
    put r15     #save partial_low

    lookup 8
    sub r10     #8-i
    put r9
    take r1
    shr r9      #a1>>8-i
    put r14     #save partial_high

    of0         #set overlow_bit to zero
    take r13
    add r15
```

```
   put r13     #sum_low = sum_low + partial_low

   take r12
   add r14
   put r12     #sum_high = sum_high + partial_high
   of0

A1B2_S:
   #check counter at the end of loop
   lookup 1
   add r10
   put r10

   lookup 8
   put r9         #r9 is the upper limit of loop
   take r10
   lsn r9
   b0 A1B2_OUT

   lookup 0
   b0 A1B2

A1B2_OUT:

   #s1(r5) = s1 + 0 + (of)
   #s2(r6) = s2 + sum_high = r6 + r12 (of)
   #s3(r7) = s3 + sum_low = r7 + r13 (of)

   of0
   take r7        #s3 = s3 + sum_low
   add r13
   put r7

   take r6        #s2 = s2 + sum_hight
   add r12
   put r6

   lookup 0
   add r5         #s1 = s1 + 0 + of
   put r5
   of0

   #a2xb1 (r2xr3)---------------------------------
```

```
   #reset partial sum to zero
   lookup 0
   put r12
   put r13
   put r14
   put r15

   put r10        #i=0
A2B1:

   #mask=1 << i
   lookup 1
   shl r10
            #acc and b1(r3)
   nand r3
   nand r0     #create an and gate: acc = b_bit = b1 & mask

   b0 A2B1_S
   take r2
   shl r10
   put r15     #save partial_low

   lookup 8
   sub r10     #8-i
   put r9
   take r2
   shr r9      #a2>>8-i
   put r14     #save partial_high

   of0         #set overlow_bit to zero
   take r13
   add r15
   put r13     #sum_low = sum_low + partial_low

   take r12
   add r14
   put r12
   of0

A2B1_S:
   #check counter at the end of loop
   lookup 1
   add r10
```

```
    put r10

    lookup 8
    put r9          #r9 is the upper limit of loop
    take r10
    lsn r9
    b0 A2B1_OUT

    lookup 0
    b0 A2B1

A2B1_OUT:

    #s1(r5) = s1 + 0 + (of)
    #s2(r6) = s2 + sum_high = r6 + r12 (of)
    #s3(r7) = s3 + sum_low = r7 + r13 (of)

    of0
    take r7         #s3 = s3 + sum_low
    add r13
    put r7

    take r6         #s2 = s2 + sum_hight
    add r12
    put r6

    lookup 0
    add r5          #s1 = s1 + 0 + of
    put r5
    of0

#a1xb1 (r1xr3)--------------------------------
    #reset partial sum to zero
    lookup 0
    put r12
    put r13
    put r14
    put r15

    put r10         #i=0
A1B1:

    #mask=1 << i
```

```
    lookup 1
    shl r10
            #acc and b1(r3)
    nand r3
    nand r0     #create an and gate: acc = b_bit = b1 & mask

    b0 A1B1_S
    take r1
    shl r10
    put r15     #save partial_low

    lookup 8
    sub r10     #8-i
    put r9
    take r1
    shr r9      #a1>>8-i
    put r14     #save partial_high

    of0         #set overlow_bit to zero
    take r13
    add r15
    put r13     #sum_low = sum_low + partial_low

    take r12
    add r14
    put r12
    of0

A1B1_S:
    #check counter at the end of loop
    lookup 1
    add r10
    put r10

    lookup 8
    put r9          #r9 is the upper limit of loop
    take r10
    lsn r9
    b0 A1B1_OUT

    lookup 0
    b0 A1B1
```

```
A1B1_OUT:

   #s1(r5) = s1 + sum_high = r5 + r12
   #s2(r6) = s2 + sum_low= r6 + r13 (of)
   of0
   take r6
   add r13
   put r6

   take r5
   add r12
   put r5
   of0

   #if isNeg is true, 2's complement result

   take r11    #acc = isNeg
   b0 SKIP     #if acc(isNeg) == 0, skip 2's complement

   take r5
   nand r0     #nand with acc will create not gate
   put r5      #s1 = ~s1
   take r6
   nand r0
   put r6
   take r7
   nand r0
   put r7
   take r8
   nand r0
   put r8      #flip s1~s4

   of0
   lookup 1    #acc = 1
   add r8      #s4 = s4 + 1, pass down overflow
   put r8

   lookup 0
   add r7
   put r7

   lookup 0
   add r6
```

```
    put r6

    lookup 0
    add r5
    put r5

SKIP:
    #write the result s1~s4 into memory location 5~8
    #make r1 the destination address
    lookup 5    #acc = 5
    put r1      #r1 = 5
    take r5     #acc = r5 = s1
    store r1    #mem[5] = s1 (r5)

    lookup 10   #acc = 6
    put r1      #r1 = 6
    take r6     #acc = r6 = s2
    store r1    #mem[6] = s2 (r6)

    lookup 11   #acc = 7
    put r1      #r1 = 7
    take r7     #acc = r7 = s3
    store r1    #mem[7] = s3 (r7)

    lookup 8    #acc = 8
    put r1      #r1 = 8
    take r8     #acc = r8 = s4
    store r1    #mem[8] = s4 (r8)
```

18: String Match

```
# r0 = accumulator
# r1 ~ r5 = single_match ~ quintuple_match
# r6 = string / temporary
# r7 = temp_string
# r8 = key
# r9 = temp base
# r10= result
# r11= count
# r12= counter of LOAD_MEM
# r13= counter of FIVE
# r14= memory address
# r15= temporary shift amount

    #initialize count and match counter
    lookup 0        #$acc  = table[0] = 0

    put r1          #single_match=0
    put r2          #...
    put r3          #
    put r4          #
    put r5          #quintuple_match=0

    #start loading key and string from memory
    lookup 9         #acc = table[9] = 9
    put r14         #r14 = 9
    load r14         #load the byte of memory at address r14 (which is 9) into accumulator
    put r8          #put the content of accumulator into r8, key stored

    #key = key << 4;
    #key = key >> 4;
    lookup 2         #acc = table[2] = 2
    add r0          #acc = acc + acc = 2 + 2 = 4 TODO: both operands = acc might cause problem

    put r15         #r15 = shift amount = 4
    take r8         #take key into accumulator, acc = key
    shl r15         #shift the key to the left by 4 bit
    shr r15         #shift the key to the right by 4 bit
    put r8          #put key back

    #since we finish loading key from the mem, relase r14 for string loading
    lookup 3         #acc = 32
```

```
    put r14          #first string address = 32


    #set up r12, counter of the loop LOAD_MEM
    lookup 0         #acc = 0
    put r12          #put 0 into r12, r12 is the counter to loop through 64 bytes

LOAD_MEM:
    lookup 0
    put r11          #count=0, reset the count at begin of loop loading string

    load r14         #load the byte of memory at address r14 into accumulator, r14 = mem[?]
    put r6           #save the string you are comparing into r6

    #set up counter of loop FIVE r13, start key matching
    lookup 0
    put r13    #i = r13 = 0

FIVE:
    #temp_string(r7) = string(r6)
    take r6          #acc = r6 = string
    shl r13          #temp_string << i
    shr r13          #temp_string >> i
    put r7           #save temp_string

    #acc = 4-i
    lookup 2
    add r0           #acc = 4
    sub r13          #acc = 4-i
    put r6           #r6 = 4-i

    #temp_string = temp_string << (4-i)
    take r7          #acc = temp_string
    shl r6           #shift right (4-i), acc = temp_string >> (4-i)
    put r7           #store the result back to temp_string

    #result = temp_string ^ key
    xor r8
    put r10

    #if result==0 count++
    lookup 0         #acc = 0
    eql r10          #if result == 0 (acc) acc=1 then count++
```

```
    b0 SKIP          #result != 0 means key unmatch => do NOT increment count

    lookup 1         #acc = 1
    add r11          #acc = 1 + r11
    put r11          #save r11, r11++
SKIP:
    #r13(i)++, if i<5 loop again
    lookup 5         #acc = 5
    put r15          #r15 = 5
    lookup 1
    add r13          #i++
    put r13
    lsn r15          #if (r13) i<5 (r15) acc = 1, else end Loop FIVE
    b0 OUT_FIVE      #end loop FIVE
    lookup 0
    b0 FIVE          #loop FIVE

OUT_FIVE:
    #check count(r11) and increment corresponding match
    lookup 1
    eql r11          #check if count == 1 (single match)
    put r15          #since if count == 1, we want to branch to SINGLE but we dont have b1
    lookup 0
    eql r15          #if count == 1, acc will become 0
    b0 SINGLE

    lookup 2
    eql r11
    put r15
    lookup 0
    eql r15
    b0 DOUBLE

    lookup 1
    put r15
    lookup 2
    add r15          #acc = 1+2 = 3
    eql r11          #check if count == 3
    put r15
    lookup 0
    eql r15
    b0 TRIPLE
```

```
    lookup 2
    add r0          #acc = 2+2 = 4 TODO: acc+acc
    eql r11
    put r15
    lookup 0
    eql r15
    b0 QUADRUPLE

    lookup 5
    eql r11
    put r15
    lookup 0
    eql r15
    b0 QUINTUPLE

    lookup 0
    b0 BREAK        #count == 0, dont NOT increment any match

SINGLE:
    lookup 1
    add r1
    put r1          #r1++
    lookup 0
    b0 BREAK
DOUBLE:
    lookup 1
    add r2
    put r2
    lookup 0
    b0 BREAK
TRIPLE:
    lookup 1
    add r3
    put r3
    lookup 0
    b0 BREAK
QUADRUPLE:
    lookup 1
    add r4
    put r4
    lookup 0
    b0 BREAK
QUINTUPLE:
```

```
    lookup 1
    add r5
    put r5
BREAK:

    #r14++, for loading next string from mem
    lookup 1
    add r14
    put r14             #r14 loop from 32 ~ 95

    #r12++, if r12<64 loop LOAD_MEM
    lookup 4            #acc = 64
    put r15             #r15 = 64
    lookup 1            #acc = 1
    add r12
    put r12             #counter of loop LOAD_MEM r12++
    lsn r15             #if acc(r12) < 64, acc = 1, else acc = 0
    b0 OUT_LOAD_MEM     #if acc(r12) < 64 is false(0), end loop

    lookup 0            #acc = 0
    b0 LOAD_MEM         #jump to LOAD_MEM

OUT_LOAD_MEM:
    #store match# into memory

    lookup 5
    add r0              #acc = 5+5 = 10
    put r9              #r9 = first mem location (10) to store

    take r1             #acc = count of Single Match
    store r9            #mem[r9] = mem[10] = r1 = single_match

    lookup 1
    add r9
    put r9              #r9++, now r9 = 11

    take r2
    store r9            #mem[r9] = mem[11] = r2 = double_match

    lookup 1
    add r9
    put r9              #r9++, now r9 = 12
```

```
take r3
store r9            #mem[r9] = mem[12] = r3 = triple_match

lookup 1
add r9
put r9              #r9++, now r9 = 13

take r4
store r9            #mem[r9] = mem[13] = r4 = quadruple_match

lookup 1
add r9
put r9              #r9++, now r9 = 14

take r5
store r9            #mem[r9] = mem[14] = r5 = quintuple_match
```

19: Hamming Distance

```
lookup 6                #acc = 127 mem&. TODO: change syntax of key
put r11         #returnReg = 127 mem&
lookup 1
add r11         #acc = 128 mem&
put r5          #i = r5 = acc = 128 mem&
lookup 2
add r11         #acc = 129 mem&
put r6                  #j = r6 = acc
lookup 0
put r2                  #r2 = acc = biggestHamDist
put r4                  #r4 = byte = acc = 0

                  #for loop from i=0 to 18 (19 times not 20)
        Binomial:               #for loop from j=i+1 to 19
                load r5         #acc = mem[i]
                put r7                  #r7 = acc = mem[i]
                load r6         #acc = mem[j]
                put r8                  #r8 = acc = mem[j]
                take r7         #acc = mem[i]
                xor r8                  #acc = mem[i] ^ mem[j]
                put r9                  #dist = acc = mem[i] ^ mem[j]
                lookup 0
                put r3                  #currHamDist = acc = 0

                CheckLSB:               #for loop from byte = 0 to 7 (8bits)
                        lookup 1                #mask.
                        nand r9         #acc = !(dist & mask)
                nand r0          #acc = !(acc & acc) = dist & mask
                        put r10         #temp r10 = dist & mask
                        lookup 1
                        eql r10         #if acc == temp, acc = 1
                        b0 NoMatch              #branch if acc == 0, acc != temp
                        lookup 1
                        add r3                  #acc = r3 + 1 = currHamDist + 1
                        put r3                  #currHamDist++
                        take r2         #acc = r2 = biggestHamDist
                        lsn r3                  #if acc=r2 < r3, if biggestHamDist < currHamDist, acc =
1
                        b0 NoMatch              #branch if !(biggestHamDist < currHamDist)
                        take r3         #acc = r3 = currHamDist
                        put r2                  #r2 = acc = r3, biggestHamDist = currHamDist
```

```
        lookup 8
        eql r2                  #if acc == r2, if biggestHamDist == 8, acc = 1
        b0 NoMatch
        lookup 0
        b0 ReturnResult

        NoMatch:                #here if my if statement checks fail
        lookup 1
        put r10         #temp = r10 = 1
        take r9         #acc = r9 = dist
        shr r10         #acc = r9 >> 1 = dist >> 1
        lookup 1
        add r4                  #acc = byte+1
        put r4                  #r4 = byte = acc = byte+1, byte++
        lookup 8
        put r10         #temp = 8
        take r4         #acc = r4 = byte
        lsn r10         #if acc < 8, acc = 1
        put r10         #temp = acc
        lookup 0
        eql r10         #if acc == temp, temp == 0, acc = 1
        b0 CheckLSB             #byte < 8, acc = 0
lookup 1
add r11         #acc = 1+127
put r10         #temp = 128
take r5         #acc = i
sub r10         #acc = i - 128
add r6                  #acc = i - 128 + j
put r10         #temp = i - 128 + j
lookup 1
add r10         #acc = i-128+j+1
put r6                  #j = r6 = acc = i-128+j+1, c code: j=j+i+1
lookup 2
add r11         #acc = 2+127
put r10         #temp = 129
take r6         #acc = j
sub r10         #acc = j-129
put r10         #temp = j-129
lookup 7                #acc = 20
put r7                  #temp2 = 20
take r10                #acc = temp = j-129
lsn r7                  #if acc < temp2=20, acc = 1
put r10         #temp = acc
```

```
                lookup 0
                eql r10         #if acc == temp, temp == 0, !(j<20), acc = 1
                b0 Binomial             #(j < 20)

        lookup 1
        add r5                  #acc = i + 1
        put r5                  #i++
        lookup 1
        add r11         #acc = 1 + 127 = 128
        put r10         #temp = 128
        take r5         #acc = i
        sub r10         #acc = i - temp = i - 128
        put r10
        lookup 1
        put r7                  #temp2 = 1
        lookup 7                #acc =20
        sub r7          #acc = 19
        put r7          #temp2 = 19
        take r10                #acc= temp = i-128
        lsn r7                  #if acc < 19, acc = 1
        put r10         #temp = acc
        lookup 0
        eql r10         #if acc == temp, temp == 0, !(j<20), acc = 1
        b0 Binomial             #branch to Binomial1 if i < 19
ReturnResult:
take r2         #acc = r2 = biggestHamDist
store r11               #mem[127] = acc = biggestHamDist
```

4. Machine code output

17: Product

```
0_1000_0001    // lookup 1
0_0010_0000    // load r0
0_0001_0001    // put r1
0_1000_0010    // lookup 2
0_0010_0000    // load r0
0_0001_0010    // put r2
0_1000_0010    // lookup 2
0_0001_1001    // put r9
0_1000_0001    // lookup 1
0_1011_1001    // add r9
0_0010_0000    // load r0
0_0001_0011    // put r3
0_1000_0010    // lookup 2
0_1011_0000    // add r0
0_0010_0000    // load r0
0_0001_0100    // put r4
0_1000_0101    // lookup 5
0_0001_1100    // put r12
0_1000_0010    // lookup 2
0_1011_1100    // add r12
0_0001_1100    // put r12
0_0000_0001    // take r1
0_0111_1100    // shr r12
0_0001_1001    // put r9
0_0000_0011    // take r3
0_0111_1100    // shr r12
0_0001_1010    // put r10
0_1000_0000    // lookup 0
0_0001_1011    // put r11
0_1000_0001    // lookup 1
0_1010_1001    // eql r9
1_00001000    // b0 S_1
0_1000_0000    // lookup 0
0_1010_1010    // eql r10
1_00000101    // b0 S_1
0_1000_0001    // lookup 1
0_0001_1011    // put r11
0_1000_0000    // lookup 0
1_00000111    // b0 S_2
0_1000_0001    // lookup 1
```

```
0_1010_1010    // eql r10
0_1010_1001    // eql r9
1_00000011    // b0 S_2
0_1000_0001    // lookup 1
0_0001_1011    // put r11
0_1000_0001    // lookup 1
0_1010_1001    // eql r9
1_00010100    // b0 S_3
0_0000_0001    // take r1
0_0101_0000    // nand r0
0_0001_0001    // put r1
0_0000_0010    // take r2
0_0101_0000    // nand r0
0_0001_0010    // put r2
0_1000_1111    // lookup 15
0_1010_0010    // eql r2
1_00001000    // b0 S_4
0_1000_0000    // lookup 0
0_0001_0010    // put r2
0_1000_0001    // lookup 1
0_1011_0001    // add r1
0_0001_0001    // put r1
0_1000_0000    // lookup 0
1_00000100    // b0 S_3
0_1000_0001    // lookup 1
0_1011_0010    // add r2
0_0001_0010    // put r2
0_1000_0001    // lookup 1
0_1010_1010    // eql r10
1_00010100    // b0 S_5
0_0000_0011    // take r3
0_0101_0000    // nand r0
0_0001_0011    // put r3
0_0000_0100    // take r4
0_0101_0000    // nand r0
0_0001_0100    // put r4
0_1000_1111    // lookup 15
0_1010_0100    // eql r4
1_00001000    // b0 S_6
0_1000_0000    // lookup 0
0_0001_0100    // put r4
0_1000_0001    // lookup 1
0_1011_0011    // add r3
```

```
0_0001_0011    // put r3
0_1000_0000    // lookup 0
1_00000100    // b0 S_5
0_1000_0001    // lookup 1
0_1011_0100    // add r4
0_0001_0100    // put r4
0_1000_0000    // lookup 0
0_0001_0101    // put r5
0_0001_0110    // put r6
0_0001_0111    // put r7
0_0001_1000    // put r8
0_0001_1100    // put r12
0_0001_1101    // put r13
0_0001_1110    // put r14
0_0001_1111    // put r15
0_0001_1010    // put r10
0_1000_0001    // lookup 1
0_0110_1010    // shl r10
0_0101_0100    // nand r4
0_0101_0000    // nand r0
1_00010010    // b0 A2B2_S
0_0000_0010    // take r2
0_0110_1010    // shl r10
0_0001_1111    // put r15
0_1000_1000    // lookup 8
0_1100_1010    // sub r10
0_0001_1001    // put r9
0_0000_0010    // take r2
0_0111_1001    // shr r9
0_0001_1110    // put r14
0_1101_0000    // of0
0_0000_1101    // take r13
0_1011_1111    // add r15
0_0001_1101    // put r13
0_0000_1100    // take r12
0_1011_1110    // add r14
0_0001_1100    // put r12
0_1101_0000    // of0
0_1000_0001    // lookup 1
0_1011_1010    // add r10
0_0001_1010    // put r10
0_1000_1000    // lookup 8
0_0001_1001    // put r9
```

```
0_0000_1010    // take r10
0_1001_1001    // lsn r9
1_00000011    // b0 A2B2_OUT
0_1000_0000    // lookup 0
1_11100001    // b0 A2B2
0_1101_0000    // of0
0_0000_1000    // take r8
0_1011_1101    // add r13
0_0001_1000    // put r8
0_0000_0111    // take r7
0_1011_1100    // add r12
0_0001_0111    // put r7
0_1101_0000    // of0
0_1000_0000    // lookup 0
0_0001_1100    // put r12
0_0001_1101    // put r13
0_0001_1110    // put r14
0_0001_1111    // put r15
0_0001_1010    // put r10
0_1000_0001    // lookup 1
0_0110_1010    // shl r10
0_0101_0100    // nand r4
0_0101_0000    // nand r0
1_00010010    // b0 A1B2_S
0_0000_0001    // take r1
0_0110_1010    // shl r10
0_0001_1111    // put r15
0_1000_1000    // lookup 8
0_1100_1010    // sub r10
0_0001_1001    // put r9
0_0000_0001    // take r1
0_0111_1001    // shr r9
0_0001_1110    // put r14
0_1101_0000    // of0
0_0000_1101    // take r13
0_1011_1111    // add r15
0_0001_1101    // put r13
0_0000_1100    // take r12
0_1011_1110    // add r14
0_0001_1100    // put r12
0_1101_0000    // of0
0_1000_0001    // lookup 1
0_1011_1010    // add r10
```

```
0_0001_1010    // put r10
0_1000_1000    // lookup 8
0_0001_1001    // put r9
0_0000_1010    // take r10
0_1001_1001    // lsn r9
1_00000011    // b0 A1B2_OUT
0_1000_0000    // lookup 0
1_11100001    // b0 A1B2
0_1101_0000    // of0
0_0000_0111    // take r7
0_1011_1101    // add r13
0_0001_0111    // put r7
0_0000_0110    // take r6
0_1011_1100    // add r12
0_0001_0110    // put r6
0_1000_0000    // lookup 0
0_1011_0101    // add r5
0_0001_0101    // put r5
0_1101_0000    // of0
0_1000_0000    // lookup 0
0_0001_1100    // put r12
0_0001_1101    // put r13
0_0001_1110    // put r14
0_0001_1111    // put r15
0_0001_1010    // put r10
0_1000_0001    // lookup 1
0_0110_1010    // shl r10
0_0101_0011    // nand r3
0_0101_0000    // nand r0
1_00010010    // b0 A2B1_S
0_0000_0010    // take r2
0_0110_1010    // shl r10
0_0001_1111    // put r15
0_1000_1000    // lookup 8
0_1100_1010    // sub r10
0_0001_1001    // put r9
0_0000_0010    // take r2
0_0111_1001    // shr r9
0_0001_1110    // put r14
0_1101_0000    // of0
0_0000_1101    // take r13
0_1011_1111    // add r15
0_0001_1101    // put r13
```

```
0_0000_1100    // take r12
0_1011_1110    // add r14
0_0001_1100    // put r12
0_1101_0000    // of0
0_1000_0001    // lookup 1
0_1011_1010    // add r10
0_0001_1010    // put r10
0_1000_1000    // lookup 8
0_0001_1001    // put r9
0_0000_1010    // take r10
0_1001_1001    // lsn r9
1_00000011    // b0 A2B1_OUT
0_1000_0000    // lookup 0
1_11100001    // b0 A2B1
0_1101_0000    // of0
0_0000_0111    // take r7
0_1011_1101    // add r13
0_0001_0111    // put r7
0_0000_0110    // take r6
0_1011_1100    // add r12
0_0001_0110    // put r6
0_1000_0000    // lookup 0
0_1011_0101    // add r5
0_0001_0101    // put r5
0_1101_0000    // of0
0_1000_0000    // lookup 0
0_0001_1100    // put r12
0_0001_1101    // put r13
0_0001_1110    // put r14
0_0001_1111    // put r15
0_0001_1010    // put r10
0_1000_0001    // lookup 1
0_0110_1010    // shl r10
0_0101_0011    // nand r3
0_0101_0000    // nand r0
1_00010010    // b0 A1B1_S
0_0000_0001    // take r1
0_0110_1010    // shl r10
0_0001_1111    // put r15
0_1000_1000    // lookup 8
0_1100_1010    // sub r10
0_0001_1001    // put r9
0_0000_0001    // take r1
```

```
0_0111_1001   // shr r9
0_0001_1110   // put r14
0_1101_0000   // of0
0_0000_1101   // take r13
0_1011_1111   // add r15
0_0001_1101   // put r13
0_0000_1100   // take r12
0_1011_1110   // add r14
0_0001_1100   // put r12
0_1101_0000   // of0
0_1000_0001   // lookup 1
0_1011_1010   // add r10
0_0001_1010   // put r10
0_1000_1000   // lookup 8
0_0001_1001   // put r9
0_0000_1010   // take r10
0_1001_1001   // lsn r9
1_00000011   // b0 A1B1_OUT
0_1000_0000   // lookup 0
1_11100001   // b0 A1B1
0_1101_0000   // of0
0_0000_0110   // take r6
0_1011_1101   // add r13
0_0001_0110   // put r6
0_0000_0101   // take r5
0_1011_1100   // add r12
0_0001_0101   // put r5
0_1101_0000   // of0
0_0000_1011   // take r11
1_00011010   // b0 SKIP
0_0000_0101   // take r5
0_0101_0000   // nand r0
0_0001_0101   // put r5
0_0000_0110   // take r6
0_0101_0000   // nand r0
0_0001_0110   // put r6
0_0000_0111   // take r7
0_0101_0000   // nand r0
0_0001_0111   // put r7
0_0000_1000   // take r8
0_0101_0000   // nand r0
0_0001_1000   // put r8
0_1101_0000   // of0
```

```
0_1000_0001   // lookup 1
0_1011_1000   // add r8
0_0001_1000   // put r8
0_1000_0000   // lookup 0
0_1011_0111   // add r7
0_0001_0111   // put r7
0_1000_0000   // lookup 0
0_1011_0110   // add r6
0_0001_0110   // put r6
0_1000_0000   // lookup 0
0_1011_0101   // add r5
0_0001_0101   // put r5
0_1000_0101   // lookup 5
0_0001_0001   // put r1
0_0000_0101   // take r5
0_0011_0001   // store r1
0_1000_1010   // lookup 10
0_0001_0001   // put r1
0_0000_0110   // take r6
0_0011_0001   // store r1
0_1000_1011   // lookup 11
0_0001_0001   // put r1
0_0000_0111   // take r7
0_0011_0001   // store r1
0_1000_1000   // lookup 8
0_0001_0001   // put r1
0_0000_1000   // take r8
0_0011_0001   // store r1
```

18: String match

```
0_1000_0000   // lookup 0
0_0001_0001   // put r1
0_0001_0010   // put r2
0_0001_0011   // put r3
0_0001_0100   // put r4
0_0001_0101   // put r5
0_1000_1001   // lookup 9
0_0001_1110   // put r14
0_0010_1110   // load r14
0_0001_1000   // put r8
0_1000_0010   // lookup 2
```

```
0_1011_0000   // add r0
0_0001_1111   // put r15
0_0000_1000   // take r8
0_0110_1111   // shl r15
0_0111_1111   // shr r15
0_0001_1000   // put r8
0_1000_0011   // lookup 3
0_0001_1110   // put r14
0_1000_0000   // lookup 0
0_0001_1100   // put r12
0_1000_0000   // lookup 0
0_0001_1011   // put r11
0_0010_1110   // load r14
0_0001_0110   // put r6
0_1000_0000   // lookup 0
0_0001_1101   // put r13
0_0000_0110   // take r6
0_0110_1101   // shl r13
0_0111_1101   // shr r13
0_0001_0111   // put r7
0_1000_0010   // lookup 2
0_1011_0000   // add r0
0_1100_1101   // sub r13
0_0001_0110   // put r6
0_0000_0111   // take r7
0_0110_0110   // shl r6
0_0001_0111   // put r7
0_0100_1000   // xor r8
0_0001_1010   // put r10
0_1000_0000   // lookup 0
0_1010_1010   // eql r10
1_00000100    // b0 SKIP
0_1000_0001   // lookup 1
0_1011_1011   // add r11
0_0001_1011   // put r11
0_1000_0101   // lookup 5
0_0001_1111   // put r15
0_1000_0001   // lookup 1
0_1011_1101   // add r13
0_0001_1101   // put r13
0_1001_1111   // lsn r15
1_00000011    // b0 OUT_FIVE
0_1000_0000   // lookup 0
```

```
1_11100101    // b0 FIVE
0_1000_0001    // lookup 1
0_1010_1011    // eql r11
0_0001_1111    // put r15
0_1000_0000    // lookup 0
0_1010_1111    // eql r15
1_00011111    // b0 SINGLE
0_1000_0010    // lookup 2
0_1010_1011    // eql r11
0_0001_1111    // put r15
0_1000_0000    // lookup 0
0_1010_1111    // eql r15
1_00011110    // b0 DOUBLE
0_1000_0001    // lookup 1
0_0001_1111    // put r15
0_1000_0010    // lookup 2
0_1011_1111    // add r15
0_1010_1011    // eql r11
0_0001_1111    // put r15
0_1000_0000    // lookup 0
0_1010_1111    // eql r15
1_00011010    // b0 TRIPLE
0_1000_0010    // lookup 2
0_1011_0000    // add r0
0_1010_1011    // eql r11
0_0001_1111    // put r15
0_1000_0000    // lookup 0
0_1010_1111    // eql r15
1_00011000    // b0 QUADRUPLE
0_1000_0101    // lookup 5
0_1010_1011    // eql r11
0_0001_1111    // put r15
0_1000_0000    // lookup 0
0_1010_1111    // eql r15
1_00010111    // b0 QUINTUPLE
0_1000_0000    // lookup 0
1_00011000    // b0 BREAK
0_1000_0001    // lookup 1
0_1011_0001    // add r1
0_0001_0001    // put r1
0_1000_0000    // lookup 0
1_00010011    // b0 BREAK
0_1000_0001    // lookup 1
```

```
0_1011_0010    // add r2
0_0001_0010    // put r2
0_1000_0000    // lookup 0
1_00001110    // b0 BREAK
0_1000_0001    // lookup 1
0_1011_0011    // add r3
0_0001_0011    // put r3
0_1000_0000    // lookup 0
1_00001001    // b0 BREAK
0_1000_0001    // lookup 1
0_1011_0100    // add r4
0_0001_0100    // put r4
0_1000_0000    // lookup 0
1_00000100    // b0 BREAK
0_1000_0001    // lookup 1
0_1011_0101    // add r5
0_0001_0101    // put r5
0_1000_0001    // lookup 1
0_1011_1110    // add r14
0_0001_1110    // put r14
0_1000_0100    // lookup 4
0_0001_1111    // put r15
0_1000_0001    // lookup 1
0_1011_1100    // add r12
0_0001_1100    // put r12
0_1001_1111    // lsn r15
1_00000011    // b0 OUT_LOAD_MEM
0_1000_0000    // lookup 0
1_10011000    // b0 LOAD_MEM
0_1000_0101    // lookup 5
0_1011_0000    // add r0
0_0001_1001    // put r9
0_0000_0001    // take r1
0_0011_1001    // store r9
0_1000_0001    // lookup 1
0_1011_1001    // add r9
0_0001_1001    // put r9
0_0000_0010    // take r2
0_0011_1001    // store r9
0_1000_0001    // lookup 1
0_1011_1001    // add r9
0_0001_1001    // put r9
0_0000_0011    // take r3
```

```
0_0011_1001    // store r9
0_1000_0001    // lookup 1
0_1011_1001    // add r9
0_0001_1001    // put r9
0_0000_0100    // take r4
0_0011_1001    // store r9
0_1000_0001    // lookup 1
0_1011_1001    // add r9
0_0001_1001    // put r9
0_0000_0101    // take r5
0_0011_1001    // store r9
```

19: Hamming distance

```
0_1000_0110    // lookup 6
0_0001_1011    // put r11
0_1000_0001    // lookup 1
0_1011_1011    // add r11
0_0001_0101    // put r5
0_1000_0010    // lookup 2
0_1011_1011    // add r11
0_0001_0110    // put r6
0_1000_0000    // lookup 0
0_0001_0010    // put r2
0_0001_0100    // put r4
0_0010_0101    // load r5
0_0001_0111    // put r7
0_0010_0110    // load r6
0_0001_1000    // put r8
0_0000_0111    // take r7
0_0100_1000    // xor r8
0_0001_1001    // put r9
0_1000_0000    // lookup 0
0_0001_0011    // put r3
0_1000_0001    // lookup 1
0_0101_1001    // nand r9
0_0101_0000    // nand r0
0_0001_1010    // put r10
0_1000_0001    // lookup 1
0_1010_1010    // eql r10
1_00001110    // b0 NoMatch
0_1000_0001    // lookup 1
```

```
0_1011_0011    // add r3
0_0001_0011    // put r3
0_0000_0010    // take r2
0_1001_0011    // lsn r3
1_00001000    // b0 NoMatch
0_0000_0011    // take r3
0_0001_0010    // put r2
0_1000_1000    // lookup 8
0_1010_0010    // eql r2
1_00000011    // b0 NoMatch
0_1000_0000    // lookup 0
1_00111100    // b0 ReturnResult
0_1000_0001    // lookup 1
0_0001_1010    // put r10
0_0000_1001    // take r9
0_0111_1010    // shr r10
0_1000_0001    // lookup 1
0_1011_0100    // add r4
0_0001_0100    // put r4
0_1000_1000    // lookup 8
0_0001_1010    // put r10
0_0000_0100    // take r4
0_1001_1010    // lsn r10
0_0001_1010    // put r10
0_1000_0000    // lookup 0
0_1010_1010    // eql r10
1_11011110    // b0 CheckLSB
0_1000_0001    // lookup 1
0_1011_1011    // add r11
0_0001_1010    // put r10
0_0000_0101    // take r5
0_1100_1010    // sub r10
0_1011_0110    // add r6
0_0001_1010    // put r10
0_1000_0001    // lookup 1
0_1011_1010    // add r10
0_0001_0110    // put r6
0_1000_0010    // lookup 2
0_1011_1011    // add r11
0_0001_1010    // put r10
0_0000_0110    // take r6
0_1100_1010    // sub r10
0_0001_1010    // put r10
```

```
0_1000_0111    // lookup 7
0_0001_0111    // put r7
0_0000_1010    // take r10
0_1001_0111    // lsn r7
0_0001_1010    // put r10
0_1000_0000    // lookup 0
0_1010_1010    // eql r10
1_10111101    // b0 Binomial
0_1000_0001    // lookup 1
0_1011_0101    // add r5
0_0001_0101    // put r5
0_1000_0001    // lookup 1
0_1011_1011    // add r11
0_0001_1010    // put r10
0_0000_0101    // take r5
0_1100_1010    // sub r10
0_0001_1010    // put r10
0_1000_0001    // lookup 1
0_0001_0111    // put r7
0_1000_0111    // lookup 7
0_1100_0111    // sub r7
0_0001_0111    // put r7
0_0000_1010    // take r10
0_1001_0111    // lsn r7
0_0001_1010    // put r10
0_1000_0000    // lookup 0
0_1010_1010    // eql r10
1_10101001    // b0 Binomial
0_0000_0010    // take r2
0_0011_1011    // store r11
```