CSE 141L Milestone 1

John Adams A16499049

Academic Integrity

Your work will not be graded unless the signatures of all members of the group are present beneath the honor code.

To uphold academic integrity, students shall:

- Complete and submit academic work that is their own and that is an honest and fair representation of their knowledge and abilities at the time of submission.
- Know and follow the standards of CSE 141L and UCSD.

Please sign (type) your name(s) below the following statement:

I pledge to be fair to my classmates and instructors by completing all of my academic work with integrity. This means that I will respect the standards set by the instructor and institution, be responsible for the consequences of my choices, honestly represent my knowledge and abilities, and be a community member that others can trust to do the right thing even when no one is watching. I will always put learning before grades, and integrity before performance. I pledge to excel with integrity.

John P Adams

0. Team

John Adams.

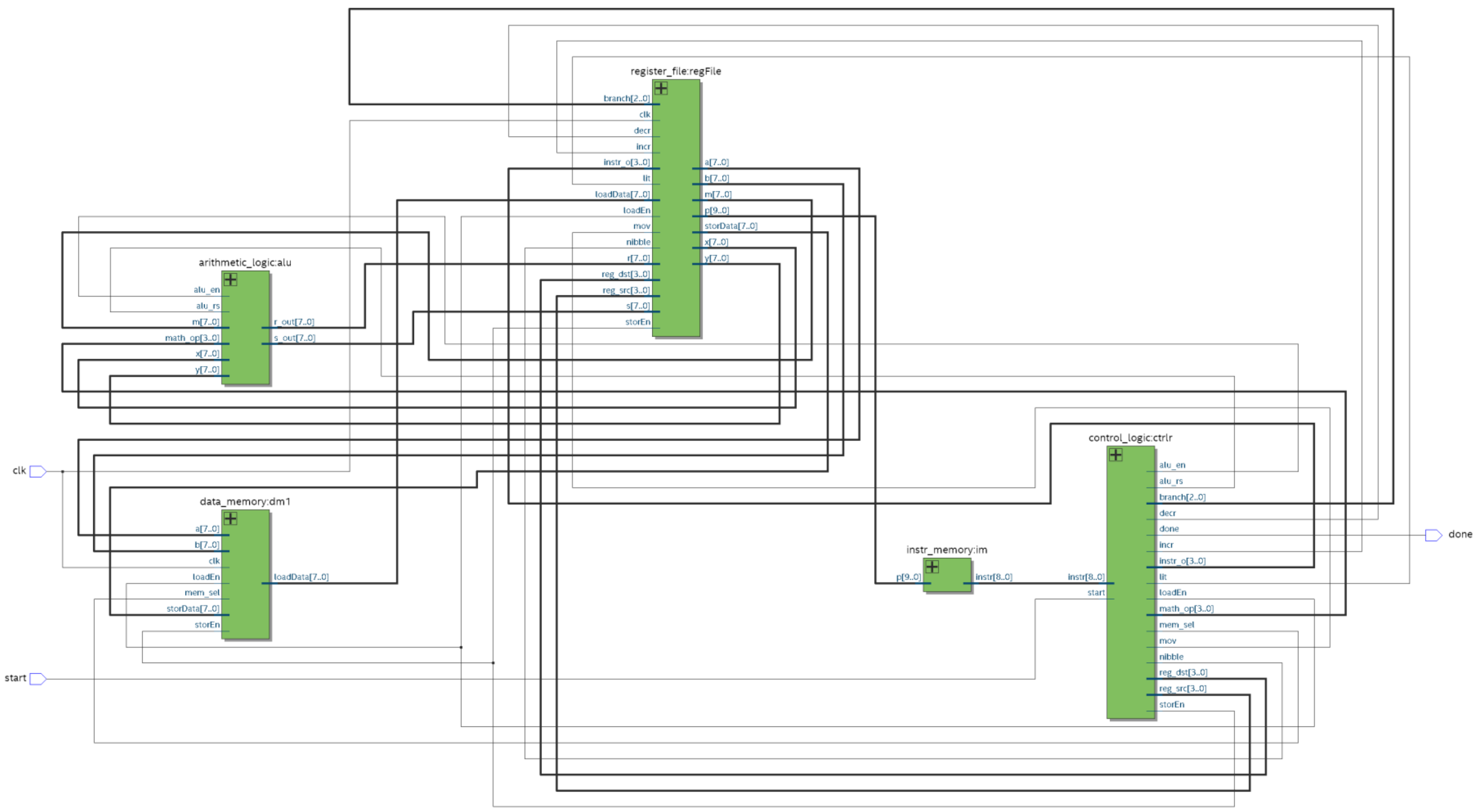
1. Introduction

Name: TMR (too many registers)
Philosophy: Use specialized registers so you can do more “stuff” without having to specify where it comes from.
Goals: make a cpu that was:

- easy to code for (lots of registers, designed with for loops in mind, 8-bit literals, many math operations)
- entirely from scratch, because why would you learn how to use an api when you can spend twice as long writing and debugging your version.

My cpu is a Load-Store (register-register) architecture. Although by using one of the address registers, it would be possible to implement a stack in software with only 2 instructions for push (stor b {reg}, incr b) and pop (load b {reg}, decr b).

2. Architectural Overview



3. Machine Specification

Instruction formats

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
I	5-bit OP code, 4-bit val	`litl, lith,`

R	5-bit OP code, 4-bit reg	`movc, movd,`
		`movm, movn, movx, movy,`
		`mova, movb, movi, movj,`
		`movk, movl, movz, movp,`
		`bizr, bnzr, incr, decr, flip`

F	5-bit OP code, 4-bit operand	`mthr, mths, func`
I’	5-bit OP code, 1-bit reg, 3-bit val	`jizr, jnzr, lslc, lsrc, seth`
R’	5-bit OP code, 1-bit reg, 3-bit reg	`load, stor`

Operations

Preface: I don’t have any bit-breakdown to do (there is only one way to .

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES

amp
 	
logical and
	F	5-bit OP code (1101X)
		
		4-bit operation (0000)
			`# let x hold b11111100`
			`# let y hold b00111111`
			
			`mth r amp`
			
			`# r now holds b00111100`
				For all operations, source registers are x and y, and the result is stored in either r or s register

lor
 	
logical or
	F	5-bit OP code (1101X)
		
		4-bit operation (0001)
			`# let x hold b11111100`
			`# let y hold b00111111`
			
			`mth r lor`
			
			`# r now holds b11111111`	

flp
 	
logical not
	F	5-bit OP code (1101X)
		
		4-bit operation (0010)
			`# let x hold b11111100`
			
			`mth r flp`
			
			`# r now holds b00000011`	

eor
 	
logical exclusive or
	F	5-bit OP code (1101X)
		
		4-bit operation (0011)
			`# let x hold b11111100`
			`# let y hold b00111111`
			
			`mth r eor`
			
			`# r now holds b11000011`	

rsc	F	5-bit OP code (1101X)	`# let x hold b10111100` `# let y hold b00111111`	res = {y[0],x[7:1]}

right shift carry		4-bit operation (0100)	`mth r rsc`
			
			`# r now holds b11011110`	

lsc
 	
left shift carry
	F	5-bit OP code (1101X)
		
		4-bit operation (0101)
			`# let y hold b00111111`
			
			`mth r lsc`
			
			`# r now holds b00011111`
				res = {x[6:0,]y[7]}

rol
 	
rotate left
	F	5-bit OP code (1101X)
		
		4-bit operation (0110)
			`# let x hold b11101000`
			`# let y hold b00000011`
			
			`mth r rol`
			
			`# r now holds b01000111`
				rotate x by value in y[2:0]

add
 	
algebraic add
	F	5-bit OP code (1101X)
		
		4-bit operation (0111)
			`# let x hold b00001111`
			`# let y hold b00000001`
			
			`mth r add`
			
			`# r now holds b00010000`	

sub
 	
algebraic subtract
	F	5-bit OP code (1101X)
		
		4-bit operation (1000)
			`# let x hold b00010110`
			`# let y hold b00000111`
			
			`mth r sub`
			
			`# r now holds b00001111`	

eql8 check for	F	5-bit OP code (1101X) 4-bit operation (1001)	`# let x hold b10101000` `# let y hold b10101111`	

byte equality			`mth r eql8`
			
			`# r now holds b00000000`	

eql5
 	
check for upper 5-bit equality
	F	5-bit OP code (1101X)
		
		4-bit operation (1010)
			`# let x hold b10101000`
			`# let m hold b10101111`
			
			`mth r eql5`
			
			`# r now holds b00000001`
				tests for x[7:4] = m[7:4]
				note this is the only math instruction that uses an register outside of x,y

revx
 	
reverse byte x
	F	5-bit OP code (1101X)
		
		4-bit operation (1011)
			`# let x hold b11110000`
			
			`mth r revx`
			
			`# r now holds b00001111`	

revy
 	
reverse byte y
	F	5-bit OP code (1101X)
		
		4-bit operation (1100)
			`# let y hold b10100011`
			
			`mth r revy`
			
			`# r now holds b11000101`	

parx
 	
compute x parity
	F	5-bit OP code (1101X)
		
		4-bit operation (1101)
			`# let x hold b00000111`
			
			`mth r parx`
			
			`# r now holds b00000001`	

pary
 	
compute y parity
	F	5-bit OP code (1101X)
		
		4-bit operation (1110)
			`# let y hold b10101010`
			
			`mth r pary`
			
			`# r now holds b00000000`	

seth
 	
set high
	I’	5-bit OP code (11001)
		
		1-bit register (X)
		
		3-bit value (XXX)
			`# let m = b00001111`
			
			`seth m 110`
			
			`# m now holds b01001111`
				m[val] = 1;

lslc
 	
logical shift left with carry
	I’	5-bit OP code (11100)
		
		1-bit register (X)
		
		3-bit value (XXX)
			`# let m = b00111100`
			`# let n = b10000000`
			
			`lslc m 1`
			
			`# m now holds b01111001`
				shifts (m/n) left by val, shifts in val highest bits from the other register (n/m)

lsrc
 	
logical shift right with carry
	I’	5-bit OP code (11101)
		
		1-bit register (X)
		
		3-bit value (XXX)
			`# let m = b00000101`
			`# let n = b00001000`
			
			`lsrc n 3`
			
			`# n now holds b10100001`
				shifts (m/n) right by val, shifts in val lowest bits from the other register (n/m)

flip
 	
bitwise xor
	R	5-bit OP code (11110)
		
		4-bit register (XXXX)
			`# let m = b00001111`
			`# let c = 000000010`
			
			`flip c`
			
			`# m now holds b00001101`
				flips a bit in (m/n) based on:
				reg[3] 0:m 1:n
				reg[2:0] val

Internal Operands

There are 16 registers (since I needed 11, I added the other 5 to use up the rest of the 4-bits needed to pick between more than 8) Several registers are special-purpose.

address:	a, b	- specify the memory address for load and store instructions
math:	x, y	- primary inputs for ALU
result:	r, s	- read-only result registers from ALU
bitwise:	m, n	- registers for bit-wise operations, and additional ALU input parameters (for special operations)
literal:	l	- location for literal value instructions
branch:	z	- branch target
pc:	p	- program counter
generic:	c, d, i, j, k	- generic registers for counters and other things.

Control Flow (branches)

Note: cycles are a measure of jumping to a **user-specified address** (i.e. directly from a literal value).

There are two conditional branches (branch if zero and branch if not zero) which can update the lower 8-bits of the program counter to the value in the branch (z) register based on the value of any register (range:256, precision:1, cycles: 4).

There are 4 ways to do a jump

- jmp instructions add or subtract (3'b * 2) from the program counter (range: 16, precision: 2, cycles: 1)
- mov instructions can copy the value of any register into the pc register's lower 8-bits (range:256, precision:1, cycles:3)
- long-jump functions modify the upper 2-bits of pc and load the branch (z) register into lower 8-bits (range:1024, precision:1, cycles:4)

Addressing Modes

Memory is handled indirectly. Memory addresses must be stored in either of the two 8-bit address registers (a, b).
Load instructions can read either address register and store into only one of the 3-bit accessible registers (c, d, m, n, x, y) excluding the read-only result registers (r, s).
Store instructions can also read either address register and then store from only one of the 3-bit accessible registers (r, s, c, d, m, n, x, y).

4. Programmer's Model [Lite]

4.1 There are a large number of registers, each of which supports increment/decrement and can be the cmp source for branch instructions. This allows for simultaneous counters to exist at the same time, without sacrificing too much space for other important values. This is especially true of the memory address registers, allowing for a 2-instruction increment-load or decrement-load style sequential memory access. The math x/y registers and result r/s registers are best suited to a particular workflow, that being load x, load y, compute > r, mov x <- r, compute > s, etc. A good example of this is a double-precision (16-bit) xor, which can be accomplished in 7 instructions. The relatively small distance provided by the conditional-relative-jump instructions makes it easier to do conditional branching forward, with one larger absolute jump back to the beginning of the program for looping processes.

4.2 The arm instruction set is proprietary protected by copyright and patent such that a license is required to modify and reproduce the same instruction set. I got around this by not looking too much at the arm instruction set. I came up with my own set of instructions needed for the programs, added some more unique instructions, and made my ISA take advantage of special use registers, which are not part of the arm ISA.

5. Program Implementation

Note: these look perfectly fine in a text editor, word is causing some ridiculous spacing. (just make sure tab-width=4)

Program 1 Assembly Code

```
litl 1101               // program 1 start (1)
lith 0001
mova l                      // store 29 in a
litl 1011
lith 0011
movb l                      // store 59 in b
load a d                // .load_routine
    decr a                  // d= {00000, b11, b10, b9}
    load a c            // c= {b8, b7, b6, b5, b4, b3, b2, b1}
    decr a
movm c                      // .parity_8
    movn d                  // {n,m} holding data
    lslc n 4            // n= {0, b11, b10, b9, b8, b7, b6, b5}
    movm m                  // m= 8'b00000000;
    lslc n 1            // n= {b11, b10, b9, b8, b7, b6, b5, 0}
    movx m                  // x= m
    mthr parx           // r= ^x
    jizr r    010           // if odd parity else jump by 0100 4 (pc = 22)
        lith 0000
        litl 1000
        flip l              // n^00000001
    movd n                  // d= n= {b11, b10, b9, b8, b7, b6, b5, p8}
movm c                      // .parity_4 m= {b8, b7, b6, b5, b4, b3, b2, b1}
    movn n                  // n= 00000000
    lslc m 4            // m= {b4, b3, b2, b1, 0, 0, 0, 0}
    lith 0001
```

```
litl 0000         // l= 00010000
movx l                 // x= 00010000
movy m                 // y= {b4, b3, b2, b1, 0000}
mthr amp          // r= {000, b1, 0000}
lith 1110         // l= 11100000
movx l                 // x= 11100000
mths amp          // s= {b4, b3, b2, 00000}
movc s                 // c= {b4, b3, b2, 00000}
lith 0000
litl 0001         // l= 00000001
movx r                 // x= {000, b1, 0000}
movy l                 // y= 00000111
mthr rol          // r= {0000, b1, 000}
movx r                 // x= r
movy c                 // y= {b4, b3, b2, 00000}
mthr lor          // r= {b4, b3, b2, 0, b1, 000}
movm r                 // m= r
movy d                 // y= {b11, b10, b9, b8, b7, b6, b5, p8}
litl 0000
lith 1111         // l= {11110000}
movx l                 // x= l
mthr amp          // r= {b11, b10, b9, b8, 0000}
movx r                 // x= r
mthr parx         // r= ^{b11, b10, b9, b8}
movy c                 // y= {b4, b3, b2, 00000}
mths pary         // s= ^{b4, b3, b2}
movx r
movy s
mthr eor          // r= ^{b11, b10, b9, b8, b4, b3, b2}
jizr r     010         // if odd parity else jump by 0100 4 (pc= 60)
     lith 0000
     litl 0100
```

```
          flip l          // m^0001000
     movc m               // c= m= {b4, b3, b2, p4, b1, 000}
movx c                    // .parity_2 x= c
     lith = 1100
     litl = 1100
     movy l               // y= 11001100
     mthr amp        // r= x&y = {b4, b3, 00, b1, 000}
     movx d               // x= {b11, b10, b9, b8, b7, b6, b5, p8}
     mths amp        // s= {b11, b10, 00, b7, b6, 00}
     movx r
     movy s
     mthr parx       // r= ^{b4, b3, b1}
     mths pary       // s= ^{b11, b10, b7, b6}
     movx r
     movy s
     mthr eor        // r= ^{b11, b10, b7, b6, b4, b3, b1}
     jizr r    010        // if odd parity else jump by 0100 4 (pc = 79)
          lith 0000
          litl 0011
          flip l          // m^00000100
     movc m               // c= m= {b4, b3, b2, p4, b1, p2, 00}
movx c                    // .parity_1
     lith 1010
     litl 1010
     movy l               // y= 10101010
     mthr amp        // r= {b4, 0, b2, 0, b1, 000}
     movx d               // x= {b11, b10, b9, b8, b7, b6, b5, p8}
     mths amp        // s= {b11, 0, b9, 0, b7, 0, b5, 0}
     movx r
     movy s
     mthr parx
     mths pary
```

```
     movx r
     movy s
     mthr eor        // r= ^{b11, b9, b7, b5, b4, b2, b1}
     jizr r    010        // if odd parity else jump by 0100 4 (pc= 98)
          lith 0000
          litl 0001
          flip l          // m^00000010
     movc m               // c= m= {b4, b3, b2, p4, b1, p2, p1, 0}
movx c                    // .parity_0
     movy d
     mthr parx
     mths pary
     movx r
     movy s
     mthr eor        // r= {b11, b10, b9, b8, b7, b6, b5, p8, b4, b3, b2, p4, b1, p2, p1}
     jizr r    010        // if odd parity else jump by 0100 4 (pc = 110)
          lith 0000
          litl 0000
          flip l          // m^00000001
     movc m               // c= m= {b4, b3, b2, p4, b1, p2, p1, p0}
stor b n             // .stor_routine
     decr b
     stor b m
     decr b
litl 0111            // .prog1_complete
     ltlh 0000
     movz l
     bnzr a               // branch if a != 0
movl l          // l= 00000000
     func strl       // start_address = 0000000000
     litl 0001       // l= 00000001
     func strh       // start_address = 0100000000
```

```
    func done        // done = 1;
```

Program 2 Assembly Code

```
// program 2 start (256)
litl 1101
lith 0001
mova l                     // store 29 in a
litl 1011
lith 0011
movb l                     // store 59 in b
load b d              // .load_routine
    decr b                 // d= {b11, b10, b9, b8, b7, b6, b5, p8}
    load b c          // c= {b4, b3, b2, p4, b1, p2, p1, p0}
    decr b
movm m                     // .parity_0
    movn n                 // m= n = 0
    movx c                 // x= c
    movy d                 // y= d
    mthr parx         // r= ^{b4, b3, b2, p4, b1, p2, p1, p0}
    mths pary         // s= ^{b11, b10, b9, b8, b7, b6, b5, p8} (parity 8)
    movx r
    movy s
    mthr eor          // r= p0 = ^{b11, b10, b9, b8, b7, b6, b5, p8, b4, b3, b2, p4, b1, p2, p1,
p0}
    movm s                 // m= p8
    lslc m 3          // m= {0000, p8, 000}
    movn r                 // n= p0
litl 0000             // .parity_4
    lith 1111
    movy l                 // y= 11110000
    movx c                 // x= {b4, b3, b2, p4, b1, p2, p1, p0}
```

```
     mthr amp        // r= {b4, b3, b2, p4, 0000}
     movx d               // x= {b11, b10, b9, b8, b7, b6, b5, p8}
     mths amp        // s= {b11, b10, b9, b8, 0000}
     movx r               // calculate parity with masked bits
     movy s
     mthr parx
     mths pary
     movx r
     movy s
     mthr eor        // r= p4 = ^{b11, b10, b9, b8, b4, b3, b2, p4}
     jizr r 0001          // if odd parity, else jump to 293
          seth 0010  // m = {0000, p8, p4, 0, 0}
litl 1100            // .parity_2
     lith 1100
     movy l               // y= 11001100
     movx c               // x= {b4, b3, b2, p4, b1, p2, p1, p0}
     mthr amp        // r= {b4, b3, 00, b1, p2, 00}
     movx d               // x= {b11, b10, b9, b8, b7, b6, b5, p8}
     mths amp        // s= {b11, b10, 00, b7, b6, 00}
     movx r               // calculate parity with masked bits
     movy s
     mthr parx
     mths pary
     movx r
     movy s
     mthr eor        // r= p2 = ^{b11, b10, b7, b6, b4, b3, b1, p2}
     jizr r 0001          // if odd parity, else jump to 309
          seth 0001  // m= {0000, p8, p4, p2, 0}
litl 1010            // .parity_1
     lith 1010
     movy l               // y= 10101010
     movx c               // x= {b4, b3, b2, p4, b1, p2, p1, p0}
```

```
      mthr amp         // r= {b4, 0, b2, 0, b1, 0, p1, 0}
      movx d                // x= {b11, b10, b9, b8, b7, b6, b5, p8}
      mths amp         // s= {b11, 0, b9, 0, b7, 0, b5, 0}
      movx r                // calculate parity with masked bits
      movy s
      mthr parx
      mths pary
      movx r
      movy s
      mthr eor         // r= p1= ^{b11, b10, b7, b6, b4, b3, b1, p2}
      jizr r 001       // if odd parity, else jump to 325
           seth 0000  // m= {0000, p8, p4, p2, p1}
movi m                      // .error_correction i= m
      movj n                // j= {0000000, b0}
      movm c
      movn d                // {n,m}= {b11, b10, b9, b8, b7, b6, b5, p8, b4, b3, b2, p4, b1, p2,
p1, p0}
      movy y
      movx n
      mthr lor         // r= {0000000, b0}
      mths amp         // s= 00000000
      jizr r 011-           // if b0= 1, else pc= 339
           flip i           // flip bit in {n,m} in position i[3:0]= {p8, p4, p2, p1}
           lith 0100  // (one error)
           litl 0000  // l= 01000000
           jizr s 101 // jump to 347
      movk k                // no op padding
      movx i
      mthr lor         // r = {0000, p8, p4, p2, p1}
      jizr r 011       // if b0= 0 && (p8|p4|p2|p1), else jump to 347
           lith 1000  // (two errors)
           litl 0000  // l= 10000000
```

```
          jizr s 001 // jump to 346
     movl l                 // (no errors) l= 00000000
     movk l
     movk l                 // k= {F1, F0, 000000}
     movc m                 // store data in {d,c}
     movd n
lith 1110              // .decode data
     litl 1000
     movy l                 // y= {11101000}
     movx m                 // x= {b4, b3, b2, p4, b1, p2, p1, p0}
     mthr amp          // r= {b4, b3, b2, 0, b1, 000}
     movm r                 // m= r
     movn n                 // n= 00000000
     lsrc m 011        // m= {000, b4, b3, b2, 0, b1}
     lsrc n 001        // n= {b1, 0000000}
     lsrc m 010        // m= {00000, b4, b3, b2}
     lslc m 101        // m= {b4, b3, b2, b1, 0000}
     movx d                 // x= {}
     lith 0000
     litl 0111
     movy l                 // y= 00000111
     mthr rol          // r= {p8, b11, b10, b9, b8, b7, b6, b5}
     movx r                 // x= r
     lith 0111
     litl 1111
     movy l                 // y= 01111111
     mthr amp          // r= {0, b11, b10, b9, b8, b7, b6, b5}
     movn r                 // n= r
     lsrc m 100        // m= {b8, b7, b6, b5, b4, b3, b2, b1}
     movc m                 // c stores lower decoded data
     movm m                 // m= 00000000
     lsrc n 100        // n= {00000, b11, b10, b9}
```

```
      movx n                 // x= n
      movy k                 // y= {F1, F0, 000000}
      mthr lor          // r= {F1, F0, 000, b11, b10, b9}
      movd r                 // d stores upper decoded data
stor a d                // .store_routine
      decr a
      stor a c
      decr a
lith 0000               // check completion
      litl 0101
      movz l
      bnzr a                 // if a!=0, then continue from 261 (0100000101)
movl l                       // l= 00000000
      func strl         // start_address = 0000000000
      litl 0010         // l= 00000010
      func strh         // start_address = 1000000000 (512)
      func done         // done = 1;
```

Program 3 Assembly Code

```
// program 3 (512) 1000000000
litl 0000                    // .initialization
      lith 0010
      movc c                      // c= 00000000 (occurences in byte)
      movd d                      // d= 00000000 (occurences across bytes)
      movb l                      // b= 00100000 (32)
      movi l                      // i= 00100000 (32)
      mova a                      // a= 00000000 (0)
      load a m               // m= 01234567
      incr a
      decr i
movj j                            // j= 00000000 (occured in byte) .setup_next_byte
```

```
     load b x               // x= vwxyz000
     load a n               // n= 89abcdef .load_next_byte
     incr a
     decr i
mthr eql5                   // r= (x[7:4] == m[7:4]) .check_pos0
     jizr r 010             // if equal, else jump +4
          incr c
          incr j
          incr j
lslc m 001                  // m= {1234567, 8} .check_pos1
     lslc n 001             // n= {9abcdef, 1}
     mthr eql5              // r= (x[7:4] == m[7:4])
     jizr r 010             // if equal, else jump +4
          incr c
          incr j
          incr j
lslc m 001                  // m= {234567, 89}
     lslc n 001             // n= {abcdef, 12}
     mthr eql5              // r= (x[7:4] == m[7:4])
     jizr r 010             // if equal, else jump +4
          incr c
          incr j
          incr j
lslc m 001                  // m= {34567, 89a} .check_pos3
     lslc n 001             // n= {bcdef, 123}
     mthr eql5              // r= (x[7:4] == m[7:4])
     jizr r 010             // if equal, else jump +4
          incr c
          incr j
          incr j
lslc m 001                  // m= {4567, 89ab} .check_pos4
     lslc n 001             // n= {cdef, 1234}
```

```
     mthr eql5              // r= (x[7:4] == m[7:4])
     jizr r 001             // if equal, else jump +2
          incr d
lslc m 001                  // m= {567, 89abc} .check_pos5
     lslc n 001             // n= {def, 12345}
     mthr eql5              // r= (x[7:4] == m[7:4])
     jizr r 001             // if equal, else jump +2
          incr d
lslc m 001                  // m= {67, 89abcd} .check_pos6
     lslc n 001             // n= {ef, 123456}
     mthr eql5              // r= (x[7:4] == m[7:4])
     jizr r 001             // if equal, else jump +2
          incr d
lslc m 001                  // m= {7, 89abcde} .check_pos7
     lslc n 001             // n= {f, 1234567}
     mthr eql5              // r= (x[7:4] == m[7:4])
     jizr r 001             // if equal, else jump +2
          incr d
     lslc m 001             // m= {89abcdef}
movx j                           // x= j (0 if no in-byte occurrences, >0 if atleast one)
     movy y                      // y= 0
     mthr lor               // r= x= k
     jizr r 001
          incr k
litl 1010                   // .check_completion
     lith 0000
     movz l                      // z=00001010 (10)
     bnzr i                      // if i = 0, else jump back to 1000001010 (522)
movj j                           // j= 00000000 .last_byte0 -------------------------
     movn n                      // n= 00000000
     load b x               // x= vwxyz000
     mthr eql5              // r= (x[7:4] == m[7:4])
```

```
     jizr r 010           // if equal, else jump +4
          incr c
          incr j
          incr j
lslc m 001                // m= {1234567, 0} .last_byte1
     mthr eql5            // r= (x[7:4] == m[7:4])
     jizr r 010           // if equal, else jump +4
          incr c
          incr j
          incr j
lslc m 001                // m= {234567, 00} .last_byte2
     mthr eql5            // r= (x[7:4] == m[7:4])
     jizr r 010           // if equal, else jump +4
          incr c
          incr j
          incr j
lslc m 001                // m= {34567, 000} .last_byte3
     mthr eql5            // r= (x[7:4] == m[7:4])
     jizr r 010           // if equal, else jump +4
          incr c
          incr j
          incr j
movx j                         // x= j (0 if no in-byte occurrences, >0 if atleast one)
     movy y                    // y= 0
     mthr lor             // r= x= k
     jizr r 001
          incr k
lith 0010                 // .store_complete
     litl 0001
     movb l                    // b= 00100001 (33)
     stor b c             // mem[33] = occurrences in byte
     movm k
```

```
incr b
stor b m                // mem[34] = bytes with occurrences
incr b
movx c
movy d
mthr add                // r= (c + d) = (occurences in byte) + (occurences across bytes)
stor b r                // mem[35] = total occurrences
func done
```