Solutions

# C to ARM Solutions

## F18. C to ARM Points: \_\_\_/20

Convert the following snippet of C-code to ARM assembly. Assume X30 holds the memory address of the instruction you should return to after the function finishes.

**Hints:** \*\* Register assignments:

x -> X1

&(arr[x]) -> X3

arr[x].d -> X6

\*\* The size of struct bar (with padding) is 24 bytes.

\*\*XZR is a register with value 0

| **C-code** |  | **ARM assembly** |
| --- | --- | --- |
| #include <inttypes.h>  struct Bar {  char a; // 1B  int64\_t b; // 8B size int64\_t\* c; // 4B size int16\_t d; // 2B size  };  void foo() {  struct Bar arr[8];  int32\_t x = 0;  for (; x < 8; x++) {  arr[x].d = x \* 16;  if (arr[x].d == 32  || arr[x].d == 64) {  arr[x].a = 0;  }  }  return;  } | 0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | Foo MOVI X1, #0  MOV X3, arr  loop CMPI X1 , #8  B.GE \_done\_  LSL X6, \_X1\_, #4  STURH \_X6\_, [\_X3\_, #20]  CMP X6, #32  B.EQ \_true\_  CMP X6, #64  B.NE \_endif\_  true STURB XZR, [\_X3\_, #\_0\_]  endif ADDI X3, X3, #\_24\_  ADDI X1, X1, #\_1\_  B \_loop\_  done BR X30 |

## F20: C-to-ARM Points: \_\_\_/14

Convert the following C program into ARM assembly by filling in the blanks. The function **isFound** should be ABI compliant and return true(1) or false(0) in register X0.

Assume X0 contains the starting address of addr\_book[]. The program returns 1 if an address is found in the address book (addr\_book).

| **C** | **ARM** |
| --- | --- |
| struct House {  int32\_t num; //4B padding: 4B  char\* street; //8B  };  int32\_t  **isFound**(struct House addr\_book[]) {  char\* foo\_street = 0xBEAD;  int32\_t foo\_num = 370;  for(int32\_t i = 0; i < 256; i++) {  if(addr\_book[i].num == foo\_num &&  addr\_book[i].street == foo\_street)  {  return 1;  }  }  return 0;  } | isFound:  MOVZ X1, #0xBEAD, LSL 0  MOVZ X2, #370, LSL 0  MOVZ X3, \_#0\_, LSL 0  loop:  CMPI X3, #256  B.EQ ret0  LSL X4, X3, \_#4\_  ADD X7, X4, \_X0\_  LDURSW X5, [X7, \_#0\_]  CMP \_X5\_, X2  B.NE cont  LDUR X6, [X7, \_#8\_]  CMP \_X1\_, \_X6\_ can swap  B.EQ ret1  cont:  ADDI \_X3\_, X3, #1  B loop  ret1:  MOVZ X0, \_#1\_, LSL 0  B end  ret0:  MOVZ \_X0\_, #0, LSL 0  end: |

## W18: Get Your C LEGv8’s Points: \_\_\_/ 15

Consider the following C code for (virtual) sailing. Finish converting the body of the function to assembly using the LEGv8 subset of ARM by filling out the blanks in the code provided. Assume that the system is **little-endian**. Assume that the arguments to sail are passed in via the specified registers, and that the address of wind is preloaded into register x3. Use the register assignments specified in the comments below.

x0 = lat\_dest x1 = fuel x2 = lat\_now x3 = wind

| C | LEGv8 |
| --- | --- |
| #include <stdint.h>  struct {  int32\_t time;  int64\_t lat\_speed;  } wind;    sail(int64\_t lat\_dest,  int64\_t fuel) {  int lat\_now = 0;    while (lat\_now != lat\_dest) {  lat\_now += wind.lat\_speed;  if (fuel > 0) {  if (lat\_now <= lat\_dest) {  lat\_now += 1;  }  else {  lat\_now -= 1;  }  --fuel;  }  wind.time += 1;  }  } | mov x2, #0  b while\_cond  loop: ldur x4, [x3, **#8**]  add x2, x2, x4  cmp x1, **xzr**  b.gt **has\_fuel**  b **time\_inc**  has\_fuel: cmp x0, x2  b.lt lat\_bigger  addi x2, x2, #1  b **dec\_fuel**  lat\_bigger: addi x2, x2, #-1  dec\_fuel: addi x1, x1, #-1  time\_inc: ldursw x4, [x3, #0]  addi x4, x4, #1  sturw x4, [x3, #0]  while\_cond: cmp x0, **x2**  b.ne loop |

## W20: C-to-ARM: I’ll Give You An ARM Points: \_\_\_/ 15

Convert the following C program into ARM assembly. Your function calcBMI should be ABI compliant and return BMIs in register X0.

Hints:

// X0 - contains starting address of arr[]

// X1 - contains starting address of BMIs[]

// X2 - holds i

UDIV Rd, Rn, Rm // Unsigned divide: Rd = Rn / Rm

What is the size of struct Person in bytes? \_\_\_\_\_\_\_\_

| **C** | **ARM** |
| --- | --- |
| struct Person {  char name[10]; // 10B  int32\_t age; // 4B  int64\_t weight; // 8B  int64\_t height; // 8B  };  int32\_t \*  **calcBMI**(struct Person arr[],  int32\_t BMIs[]) {  int32\_t i = 0;  for(; i < 100; i++) {  if(arr[i].age > 0) {  BMIs[i] = (int32\_t)arr[i].weight/  (arr[i].height\*arr[i].height);  }  }  return BMIs;  } | calcBMI:  MOVZ X2, #0  loop:  CMPI X2, #**100**  **B.GE** end  LSL X7, X2, #**5**  ADD X3, X0, X7  **LDURSW** X4, [**X3**, #**12**]  CMPI X4, #0  **B.LE** inc  true:  LSL X7, X2, #**2**  ADD X4, **X1**, X7  LDUR X5, [X3, #16 ]  LDUR X6, [X3, #**24**]  MUL X6, **X6**, X6  UDIV X5, X5, **X6**  **STURW** X5, [X4, #**0**]  inc:  ADDI X2, **X2**, #1  B loop  end:  MOV **X0**, X1 //return |

## W21: C-to-ARM and the SODA370 factory

The SODA370 factory produces soda packs with a multiple of four soda cans in each pack. However, some packing machines are starting to malfunction, making some soda packs contain a number of soda cans not divisible by four. Someone has written a quality check function below in C to check how many soda packs contain incorrect amounts of soda cans. Convert part of this function into ARM.

Assume X0 contains the starting address of data[]. The function **quality\_check** returns the amount of bad soda packs in register X2.

| **C** | **ARM** |
| --- | --- |
| struct soda\_pack {  char pack\_id[4]; //4B  int64\_t num\_cans; //8B    };  int64\_t  **quality\_check**(struct soda\_pack data[]) {  int32\_t bad\_packs = 0;  for(int32\_t i = 0; i < 256; i++) {  if(data[i].num\_cans % 4 != 0){  bad\_packs++;  }  }  return bad\_packs;  } | qcheck:  MOVI X2, #0x00  MOVI X3, #0x00  loop:  CMPI X3, **#256**  B.**GE** **end** (or B.EQ)  **LSL** X4, **X3**, **#4**  **ADD** X5, **X4**, **X0**  LDUR X6, [**X5**, **#8**]  **ANDI** X7, **X6**, **#3**  CMPI **X7**, #0  B.EQ good  bad:  ADDI X2, #1  good:  ADDI X3, #1  B loop  end: |

## W22: Convert C to ARM Assembly [12 Points]

Convert this C code to ARM assembly by completing the ***twelve missing instruction parts***. Registers **X0** and **X1** hold the starting addresses of **set1** and **set2**, while registers **X2**, and **X3** hold the length of each set, respectively. The input argument of **search\_value** is mapped to **X5** andreturn value of **find\_mutual\_sum** is at **X7**. Note that both set1 and set2 include **signed values**.

| **C** | **ARM** | | | | |
| --- | --- | --- | --- | --- | --- |
| **int64\_t set1[] = [1, 5, -3, -8, …];**  **int32\_t set2[] = [-3, 2, -4, 5, …];**  **int64\_t len1; // length of set1**  **int64\_t len2; // length of set2**  **int64\_t search\_value**(int64\_t input);  **int64\_t find\_mutual\_sum**()  {  int64\_t sum = 0;  int idx1 = 0;  for ( ; idx1 < len1 ; idx1++) {  int64\_t input = set1[idx1];  if (**search\_value**(input) > 0)  sum += input;  }  return sum;  }  **int64\_t search\_value**(int64\_t input) {  int idx2 = len2 - 1;  for ( ; idx2 >= 0 ; idx2--) {  int64\_t search =  **(int64\_t)** set2[idx2];    if (search == input)  return 1;  }  return 0;  } | **FIND\_MUTUAL\_SUM:** | | | |  |
|  | MOVZ | X7, | #0, | LSL 0 |
|  | MOVZ | X10, | #0, | LSL 0 |
| **FIND\_LOOP:** | | | |  |
|  | CMP | X10, | X2 |  |
|  | B.GE | END\_FIND\_LOOP | | |
|  | LSL | X11, | **X10**, | **#3** |
|  | ADD | X11, | X11, | X0 |
|  | LDUR | X5, | [X11, | #0] |
|  | **BL** | SEARCH\_VALUE | | |
|  | CBZ | **X8**, | CONT\_FIND\_LOOP | |
|  | ADD | X7, | X7, | **X5**, |
| **CONT\_FIND\_LOOP:** | | |  |  |
|  | ADDI | X10, | X10, | #1 |
|  | B | FIND\_LOOP | |  |
|  | |  |  |  |
| **SEARCH\_VALUE:** | |  |  |  |
|  | MOVZ | X8, | **#0**, | LSL 0 |
|  | **SUBI** | X12, | X3, | #1 |
| **SEARCH\_LOOP:** | |  |  |  |
|  | CMP | X12, | XZR |  |
|  | **B.LT** | Return |  |  |
|  | LSL | X13, | X12, | **#2** |
|  | ADD | X13, | X13, | X1 |
|  | **LDURSW** | X6, | [X13, | #0] |
|  | CMP | X6, | **X5** |  |
|  | B.EQ | IF\_ELSE\_SEARCH | |  |
|  | SUBI | X12, | X12, | #1 |
|  | B | SEARCH\_LOOP | |  |
| **IF\_ELSE\_SEARCH** | | |  |  |
|  | ADDI | X8, | XZR, | #1 |
| **Return:** | |  |  |  |
|  | **BR** | X30 |  |  |
| **END\_FIND\_LOOP:** | | |  |  |

# C to LC2K Solutions

## F19: Can you C LC2K? [16 points]

Below is a function written in C and an LC2K assembly function which does the same thing.

int main() {

letao(5);

}

int letao(int n) {

int a;

if (n == 0) {

return 1;

}

a = letao(n-1);

a = a + a;

return a;

}

main lw 0 1 value

lw 0 4 func

lw 0 2 one

lw 0 5 neg1

jalr 4 7

halt

letao beq 1 0 base

sw 6 7 Stack

add 6 2 6

add 1 5 1

jalr 4 7

add 6 5 6

lw 6 7 Stack

add 3 3 3

jalr 7 4

base add 0 2 3

jalr 7 4

func .fill letao

one .fill 1

neg1 .fill -1

value .fill 5

Stack .fill 0

.fill 0

.fill 0

.fill 0

For each register, write the letter which best describes what that register is used for. You may use a given letter more than once. [2 points each, no partial credit

R0: \_\_\_B\_\_\_\_\_

R1: \_\_\_A\_\_\_\_\_

R2: \_\_\_I\_\_\_\_\_

R3: \_\_\_C\_\_\_\_\_

R4: \_\_\_H\_\_\_\_\_

R5: \_\_\_E\_\_\_\_\_

R6: \_\_\_D\_\_\_\_\_

R7: \_\_\_F\_\_\_\_\_

A. n

B. 0

C. a

D. Stack pointer

E. -1

F. Return address

G. 5

H. letao’s address

I. 1

J. n+2a

## W18: Identity Matrix Points: \_\_\_/ 15

Consider the MATLAB function eye(N), which generates a NxN matrix with ones along the diagonal and the rest of the matrix values as zero.

Example: eye(3) matrix printed below.

1 0 0

0 1 0

0 0 1

Implement this function in LC2K by completing the blanks below. Instead of a 2D array the matrix is represented as a 1D array called eye of size N\*N (See example in C code). The final result of eye(3) should look like the comment in the C code. Use the registers naming as shown:

R0 = 0 R1 = 1 R2 = N

R3 = i R4 = j R5 = k + starting address of EYE

| // const int N = 3;  // int eye[N\*N] = {1, 0, 0,  // 0, 1, 0,  // 0, 0, 1};  // eye(N) function in C code:  const int N = 3;  int eye[N\*N];  for(int k=0; k<N\*N; ++k) {  eye[k] = 0;  }  **//LC2K implementation starts here**  int k = 0;  for(int i=0; i<N; ++i) {  for(int j=0; j<N; ++j) {  if(i == j) {  eye[k] = 1;  }  ++k;  }  }  **//LC2K implementation ends here** | lw 0 5 EYE  lw 0 1 ONE  lw 0 2 N  lw 0 3 ZERO  BEG\_I beq **3** 2 END\_I  lw 0 4 ZERO  BEG\_J beq **4 2** END\_J  beq 3 4 **SET\_1**  ADD\_K add **5 1** 5  add 4 1 4  beq 0 0 **BEG\_J**  END\_J **add** 3 1 3  beq 0 0 BEG\_I  SET1 **sw** 5 1 0  beq 0 0 ADD\_K  END\_I halt  EYE .fill 100  ONE .fill 1  N .fill 3  ZERO .fill 0 |
| --- | --- |

## W20: LC2K Assembly Points: \_\_\_/20

Convert the following C-code to LC2K assembly. **Assume all registers are initialized to 0.** Hint. Less than (<) operation is not supported natively in LC2K. However a < b can be done in LC2K by subtracting b from a, using the sign bit of the result and comparing the sign bit to 0.

| **C-code** |  | **LC2K assembly** |
| --- | --- | --- |
| int size = 10; //size --> R2  int arr1[5] =  {1, 3, 5, 7, 9};  int arr2[5] =  {2, 4, 6, 8, 10};  int dest[10];  int ind1 = 0, ind2 = 0;  //ind1 --> R3, ind2 --> R4  int num = 0; //num --> R5  while (num != size)  {  if (arr1[ind1]<arr2[ind2]) {  dest[num]= arr1[ind1];  ind1++;  num++;  } else {  dest[num]= arr2[ind2];  ind2++;  num++;  }  } | 0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | lw 0 1 one  lw 0 2 size  while beq 5 2 end  lw **3** 6 arr1  lw **4** 7 arr2  nor 0 **7** 7  add 1 7 7  add 6 **7** 6  lw 0 7 mask  nor **6** **7** 6  beq 6 0 **less**  lw 4 6 arr2  sw **5** 6 dest  add 1 4 4  add 1 5 5  beq 0 0 **while**  less lw 3 **7** arr1  sw 5 7 **dest**  add 1 **3** 3  add 1 5 5  beq 0 0 **while**  end halt  one .fill 1  size .fill 10  mask .fill 2147483647 #0x7FFFFFFF arr1 .fill 1  .fill 3  .fill 5  .fill 7  .fill 9  arr2 .fill 2  .fill 4  .fill 6  .fill 8  .fill 10  dest .fill 0 |

## W21: C to LC2K

*This question was removed from the final version of the exam, but has been available in practice exams since.*

1. Convert C code to LC2K by filling the blanks.

Note this does not follow the standard LC2K ABI:

reg2 has the start address of arr.

reg4 is the input real of find\_number.

reg5 is the stack pointer.

| **C** | **LC2K** |
| --- | --- |
| struct number{  int real;  int imagine;  }  struct number arr[5];  extern int find\_number(struct number arr[], int idx, int real)  int main(){  find\_number(arr, 4, 3);  } | main.as  lw 0 2 **arr**  lw 0 3 size  lw 0 6 neg1  lw 0 4 real  add 3 6 3  lw 0 6 **Faddr**  jalr 6 1  halt  arr .fill Array  size .fill 5  neg1 .fill -1  real .fill 3  Array .fill 3  .fill 3  .fill 2  .fill 3  .fill 3  .fill 5  .fill 8  .fill 3  .fill 5  .fill 0 |

| **C** | **LC2K** |
| --- | --- |
| struct number{  int real;  int imagine;  }  int find\_number(struct number arr[], int idx, int real){  if (arr[idx].real == real){  return idx;  }  else if(idx == 0){  return -1;  }  else{  return find\_number(arr, idx-1, real)  }  } | Func add **3** **3** 7 //step1: arr[idx]  add 7 2 7 //step2: arr[idx]  lw 7 7 0 //step3: arr[idx]  beq 7 4 **ret**  beq 3 0 **noHit**  lw 0 6 neg1  add 6 3 3  lw 0 6 **Faddr**  lw 0 7 one  sw 5 1 Stack  add 7 5 5  jalr **6** **1**  lw 0 7 neg1  add 7 5 5  lw 5 1 Stack  beq 0 0 **ret**  noHit lw 0 3 neg1  ret jalr **1** **6**  Faddr .fill Func  neg1 .fill -1  one .fill 1 |

1. What is register 3 used for? (select all apply)

A. return address

**B. idx**

C. arr

D. real

E. Stack pointer

**F. return\_value**

G. temporary register

1. Which register has the return address? Is this register a caller save register or a callee save register? (select all apply)
2. **r1**
3. r2
4. r3
5. r4
6. r5
7. r6
8. r7
9. **Caller save register**
10. Callee save register

4. What is the value that find\_number returns?

**2**

# C to Other Solutions

## F18. New ISA Points: \_\_\_/25

In 982 years, the College of Engineering decides to release its next generation ISA, the LC3K. LC3K features the following instruction types:

| **Type** | **Instruction Name** | **Opcode** | **Function** |
| --- | --- | --- | --- |
| **R-type** | add | 000 | Add contents of regA with, contents of regB, store results in destReg. |
| **R-type** | nand | 001 | NAND contents of regA with contents of regB store results in destReg. This is a bitwise operation; each bit is treated independently. |
| **I-type** | bgt | 010 | If the contents of regA is greater than the contents of regB then branch to the address PC+1+offsetField, where PC is the address of this BGT instruction. |
| **I-type** | lw | 011 | Load regB from memory. Memory address is formed by adding offsetField with the contents of regA. |
| **I-type** | sw | 100 | Store regB into memory. Memory address is formed by adding offsetFIeld with the contents of regA. |
| **S-type** | shift | 101 | Shift the contents in regA to the left (positive) or right(negative) by the number of bits in the offsetField. |
| **K-type** | swap | 110 | Swap the contents of regA and regB. |
| **O-type** | halt | 111 | Increment the PC (as with all instructions), then halt the machine (let the simulator notice that the machine halted). |

bits 24–22: opcode

bits 21–19: reg A

bits 18–16: reg B *(unused for S-type and O-type)*

bits 15–0: offsetField or destReg *(unused for O-type)*

**All unused bits in the instruction are set to zero.**

a) Translate the following LC3K assembly code to binary and hexadecimal machine code:

| **LC3K** | **Binary machine code** | **Hex machine code** |
| --- | --- | --- |
| bgt 6 7 8 | **0b 010 110 111 0000000000001000** | **0xB70008** |
| shift 1 -2 | **0b 101 001 000 1111111111111110** | **0x148FFFE** |

b) Assume that there are n distinct random numbers in memory starting at memory location arr. Write an LC3K **Bubble Sort** so that when your program halts, the numbers in arr are in strictly increasing order. *All registers are initialized to 0.*

| **C-code** |  | **LC3K assembly** |
| --- | --- | --- |
| //n is length of arr-1  //n --> R2  i = 0;//i --> R3  do{  j = 0;//j --> R4  status = 0;//status --> R5  do {  //arr[j] --> R6  //arr[j+1] --> R7  if(arr[j] >= arr[j+1])  {  swap(&arr[j],  &arr[j+1]);  status++;  }  j++;  } while (j < n-i);  if (status == 0)  break;  i++;  } while (i < n-1); | 0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | lw 0 1 one  lw 0 2 nsub1  outer lw 0 4 zero  lw 0 5 zero  inner **lw** 4 6 arr  add 1 4 7  **lw** 7 7 arr  bgt **7** **6** skip  **swap** 6 **7**  sw 4 6 arr  add 1 4 6  sw 6 7 arr  add 1 5 5  skip add 1 4 4  //next 3 insts compute n-i  **nand** 3 3 6  add 1 6 6  add 2 6 6  bgt 6 4 **inner**  bgt 5 0 **inc**  bgt 1 0 end  inc add 1 3 3  bgt 2 3 outer  end halt  zero .fill 0  one .fill 1  nsub1 .fill 4  arr .fill 2  .fill 4  .fill 6  .fill 0  .fill 1 |

## 

## W18: LIMB ISA Points: \_\_\_/ 25

We have come up with a new ISA called LIMB. LIMB is a stack-based, byte-addressable ISA with 16-bit instructions. Registers hold 16 bits of data. The stack is separate hardware structure, each entry is 32 bits. Three instruction types are detailed below. All offsets are two’s complement numbers.

| **S-type** | **15-12: Opcode** | **11-8: regA** | **7-0: signed offset** |
| --- | --- | --- | --- |

| Instruction | Opcode | Description |
| --- | --- | --- |
| breq | 0011 | If the value at the top of the stack == the contents of regA, branch to PC + 2 + offset |
| ldpsh | 0100 | Load word to the top of the stack from memory[regA + offset] (push value of loaded word) |
| store | 0101 | Store the value at the top of the stack into memory at address regA + offset |
| addi | 0110 | Add offset (immediate value) to the contents of regA, putting the result back into regA |

| **Z-type** | **15-12: Opcode** | **11-8: regA** | **7-0: unused** |
| --- | --- | --- | --- |

| Instruction | Opcode | Description |
| --- | --- | --- |
| add | 0111 | Add the value at the top of the stack to the contents of regA, **store in regA** |
| pop | 1000 | Pop value at the top of the stack and put it in regA |
| push | 1001 | Push the contents of regA to the top of the stack |
| cmpz | 1010 | Compare contents of regA with 0. Set appropriate flags for branch (b.lt = branch if less than 0, b.gt = branch if greater than 0, b.eq = branch if equal to 0) |

| **A-type** | **15-12: Opcode** | **11-0: unused** | |
| --- | --- | --- | --- |

| Instruction | Opcode | Description |
| --- | --- | --- |
| halt | 1111 | Stop the program |

**a)** Fill in the following table to convert the instructions to machine code in **Hexadecimal**. Unused bits are set to 0.

| Instruction | Machine Code (Hexadecimal 0x) |
| --- | --- |
| store 7 7 | **0x5707 (== 0b0101 0111 0000 0111)** |
| ldpsh 2 7 | **0x4207 (== 0b0100 0010 0000 0111)** |
| push 4 | **0x9400 (== 0b1001 0100 0000 0000)** |

**b)** For the following program, **assume register 1 contains value 100** and that memory is organized in a **little endian** manner. What are the resulting stack and memory layouts?

0. ldpsh 1 0 // load word to stack from address 100

1. ldpsh 1 4 // load word to stack from address 104

2. ldpsh 1 8 // load word to stack from address 108

3. store 1 0 // store word to memory address 100

Each **stack entry is 32 bits**. Write your **answers in hexadecimal**.

| Stack |  | Memory Address | Value | New Value |
| --- | --- | --- | --- | --- |
| **0x4D77788C** |  | 100 | 0xAB | **0x8C** |
| **0xEE6D4912** |  | 101 | 0x10 | **0x78** |
| **0xFB3310AB** | (Initial top of stack) | 102 | 0x33 | **0x77** |
|  |  | 103 | 0xFB | **0x4D** |
|  |  | 104 | 0x12 |  |
|  |  | 105 | 0x49 |  |
|  |  | 106 | 0x6D |  |
|  |  | 107 | 0xEE |  |
|  |  | 108 | 0x8C |  |
|  |  | 109 | 0x78 |  |
|  |  | 110 | 0x77 |  |
|  |  | 111 | 0x4D |  |

**c)** The branch instruction and conditions are defined below:

| **15-12: Opcode** | **11-0: Offset** | |
| --- | --- | --- |

| Instruction | Description |
| --- | --- |
| b | Unconditionally branch to PC + 2 + Offset |
| b.lt | Branch to PC + 2 + Offset if “cmpz regA” instruction results in “less than”  (reg[regA] < 0) |
| b.gt | Branch to PC + 2 + Offset if “cmpz regA” instruction results in “greater than” (reg[regA] > 0) |
| b.eq | Branch to PC + 2 + Offset if “cmpz regA” instruction results in “equal to”  (reg[regA] == 0) |

Say we have an array Arr[ ] of 4 integers (int32\_t). At the end of the following LIMB program, the stack will contain the values from Arr[ ] that are negative. Complete the following program. **Assume all registers contain 0 initially and Arr[] starts at address 100.**

//pseudo code

for (int i = 3; i >= 0; i--) {

if (Arr[i] < 0) {

pushToStack(Arr[i]);

}

}

0. addi 2 **12** //initialize loop counter

1. loop **cmpz** 2

2. b.lt **end (16)**

3. **ldpsh** 2 100 // load word to stack from Arr[i]

4. pop **3** // put value in reg 3

5. cmpz 3

6. b.lt **pstack (4)**

7. dec addi 2 **-4** // else, decrease address and branch

8. b loop

9. pstack push 3 //push negative element to stack

10. b **dec (-8)**

11. end halt

**We have provided both the labels and the numerical offsets for branches.**

## W21: New ISA

To improve on LC2K, we’ve developed a new, modern ISA called the LCJINX with 12 opcodes and 16 registers. Unfortunately, since our design team got lazy, we can still only support 16 bit signed immediate values. (note: Jinx definitely did not write this question)

There are four types of instructions:

* J-type instructions take in three operands, which are all registers.
* I-type instructions take in three operands, where two are registers and the last is an immediate value.
* N-type instructions take in no operands.
* X-type instructions take in four operands, where all four operands are registers.

1. **What is the minimum word size of the LCJINX? Round your answer up to the nearest byte. (3 points)**

**Solution: Since there are 12 opcodes, we need 4 bits to hold the opcode. Each register is 4 bits, and the immediate value is 16 bits. Thus, a J-type instruction is 4 (opcode) + 3 \* 4 (three registers) = 16 bits. An I-type instruction is 4 (opcode) + 2 \* 4 (two registers) + 16 (immediate) = 28 bits. A N-type instruction is 4 (opcode) bits. An X-type instruction is 4 (opcode) + 4 \* 4 (four registers) = 20 bits. Thus, the minimum word size is 28 bits, which is rounded up to 32 bits (4 bytes).**

Assume the LCJINX is word-addressable, with a word size of 8 bytes. The instructions for the LCJINX are as follows:

| **Instruction** | **Type** | **Opcode** | **Description** |
| --- | --- | --- | --- |
| add rA rB rD | J | 0x0 | Adds the values in registers A and B and stores the result to register D (destination). |
| nand rA rB rD | J | 0x1 | Takes the bitwise NAND of the values in registers A and B and stores the result to register D (destination). |
| addi rA, rD, imm | I | 0x2 | Adds the value in register A with the immediate value and stores the result to register D (destination). |
| sft rA, rB, imm | I | 0x3 | Takes the value in register A and shifts it by the number of bits specified by *imm*. If *imm* is positive, then the value is shifted to the right by *imm* bits. If *imm* is negative, the value is shifted to the left by |*imm*| bits. The result is stored into register B. |
| ld rA, rB, imm | I | 0x4 | Loads the value in memory at address (value of register A + immediate value) to register B. |
| str rA, rB, imm | I | 0x5 | Stores the value in register B to memory address (value of register A + immediate value). |
| beq rA, rB, imm | I | 0x6 | If the values in rA and rB are the same, then branch to PC = *imm*. (note: different from LC2K’s beq!) |
| blt rA, rB, imm | I | 0x7 | No, not the sandwich. If the value in register A is less than (<) the value in register B, then branch to PC = *imm*. |
| meow | N | 0x8 | Equivalent to LC2K’s *noop* instruction. Does nothing. |
| halt | N | 0x9 | Equivalent to LC2K’s *halt* instruction. Halts the program. |
| cmov rA, rB, rC, rD | X | 0xA | If the values of register A and register B are equal, set the value of register C to be the value in register D. |
| beqlr rA, rB, rC, rD | X | 0xB | If the values of register A and register B are equal, store the current PC+1 into register D and branch to the value in register C. This is similar to LC2K’s *jalr* instruction. |

For backwards compatibility and ease of use, the *.fill* directive works exactly the same as LC2K. Labels have a maximum length of 12 characters, for the optimal programmer experience (or as much as you can get while writing in assembly). All symbolic addresses resolve to the memory address of that label. Assume all registers start with the value 0.

**J-type instructions are encoded as follows:**

**bits 35-32: opcode**

**bits 31-28: register A**

**bits 27-24: register B**

**bits 23-4: unused (all 0)**

**bits 3-0: register D**

**I-type instructions are encoded as follows:**

**bits 35-32: opcode**

**bits 31-28: register A**

**bits 27-24: register B**

**bits 23-16: unused (all 0)**

**bits 15-0: 16-bit immediate**

**N-type instructions are encoded as follows:**

**bits 35-32: opcode**

**bits 31-0: unused (all 0)**

**X-type instructions are encoded as follows:**

**bits 35-32: opcode**

**bits 31-28: register A**

**bits 27-24: register B**

**bits 23-20: register C**

**bits 19-16: register D**

**bits 15-0: unused (all 0)**

1. **Complete the machine code translation of the LCJINX code shown below. Express your answers in hex.**

**Assuming all registers start as 0 at the beginning of the program, what is the value in register 5? (7 points, 4.5 for MC and 2.5 for register)**

**LCJ1NX Assembly Machine Code**

addi 1 1 16 0x211000010

sft 1 2 3 0x312000003

nand 0 0 3 0x100000003

add 2 3 4 0x**023000004**

sft 4 4 -4 0x**34400FFFC**

cmov 1 4 5 3 0x**A14530000**

halt 0x900000000

**The value in register 5 at the end is -1.**

addi 1 1 16: set register 1 to 16

sft 1 2 3: set register 2 to 16 >> 3 = 2

nand 0 0 3: set register 3 to -1 (0xFFFFFFFF)

add 2 3 4: set register 4 to 1 (2 - 1)

sft 4 4 - 4: set register 4 to 16 (1 << 4)

cmov 1 4 5 3: since register 1 & register 4 are both 16, set register 5 equal to register 3 = -1.

1. **Your team has decided to use the 370-developed J1 chip to run LCJINX, and you’re now tasked with a program to run binary search in a sorted array. Your teammates had a working prototype, but a sleep- and caffeine-deprived programmer accidentally deleted a few lines of code. Help restore the program below based on the comments. (10 points)**

**Solution 1: branch if array[m] < target**

ld 0 14 searchAddr

beqlr 0 0 14 15 // call binary search function

halt

meow

searchFunc add 0 0 1 // l = 0

ld 0 2 arraySize // r = array size

ld 0 9 target // load target

loop beq 1 2 nofind // if l == r, not found

**add** **1** **2** **3** // find m = (l+r) / 2 (2 lines)

**sft** **3** **3** **1** // store m into **reg 3**

**ld** 3 5 array // load array[m] into **reg 5**

beq 5 9 equal // if array[m] == target

blt 5 9 lessthan // handle inequalities

add 0 3 2 // else

beq 0 0 loop // else

equal add 0 3 13 // return m in **reg 13**

beq 0 0 done // done!

lessthan add 0 3 1 // inequality case 1

beq 0 0 loop // inequality case 1

done beqlr 0 0 15 14 // return

nofind addi 0 13 -1 // load return value with -1

beqlr 0 0 15 14 // return

searchAddr .fill searchFunc

arraySize .fill 8

target .fill 7

arrayAddr .fill array

array .fill 1

.fill 3

.fill 4

.fill 7

.fill 10

.fill 12

.fill 13

.fill 16

**Solution 2: branch if array[m] > target**

ld 0 14 searchAddr

beqlr 0 0 14 15 // call binary search function

halt

meow

searchFunc add 0 0 1 // l = 0

ld 0 2 arraySize // r = array size

ld 0 9 target // load target

loop beq 1 2 nofind // if l == r, not found

add 1 2 3 // find m = (l+r) / 2 (2 lines)

sft 3 3 1 // store m into **reg 3**

ld 3 5 array // load array[m] into **reg 5**

beq 5 9 equal // if array[m] == target

blt 9 5 grtthan // handle inequalities

add 0 3 2 // else

beq 0 0 loop // else

equal add 0 3 13 // return m in **reg 13**

beq 0 0 done // done!

grtthan add 0 3 1 // inequality case 1

beq 0 0 loop // inequality case 1

done beqlr 0 0 15 14 // return

nofind addi 0 13 -1 // load return value with -1

beqlr 0 0 15 14 // return

searchAddr .fill searchFunc

arraySize .fill 8

target .fill 7

arrayAddr .fill array

array .fill 1

.fill 3

.fill 4

.fill 7

.fill 10

.fill 12

.fill 13

.fill 16

## W22: New ISA [12 Points]

To cut hardware costs, a team of engineers has developed a simplified version of the LC2K ISA with only four registers—**buffer**, **RZR**, **RPR**, and **RSC**—which have the following functionality:

| **Register** | **Description** |
| --- | --- |
| buffer | Used to move register values to/from memory via the instructions **loadbuf** and **storebuf**, or between different registers via the instructions **loadreg** and **storereg** |
| R**ZR** (“**Z**e**r**o Reg”) | Holds the value 0 and cannot be changed |
| R**PR** (“**Pr**imary Reg”) | Holds values for compute, akin to standard LC2K registers |
| R**SC** (“**S**e**c**ondary Reg”) | Holds values for compute, akin to standard LC2K registers |

This new ISA[[1]](#footnote-0) has the following instructions and semantics:

| **Assembly Code** | **Execution  Semantics** | **Usage Example** |
| --- | --- | --- |
| addi **imm.** | **RPR** = **RPR** + **imm.** (Add **imm.** value to **RPR**) | addi 10 |
| add | **RPR** = **RPR** + **RSC** | add (**No args**) |
| nandi **imm.** | **RPR** = **RPR** NAND **imm.** (Nand **imm.** value with **RPR**) | nandi 42 |
| nand | **RPR** = **RPR** NAND **RSC** | nand (**No args**) |
| beq **label** | Branch to **label** if **RPR** == **RSC** | beq end\_label |
| loadreg **reg** | **reg** = **buffer** (**reg** can be **RPR** or **RSC**) | loadreg RPR |
| storereg **reg** | **buffer** = **reg**  (**reg** can be **RPR**, **RSC**, or **RZR**) | storereg RZR |
| loadbuf **imm.** | **buffer** = mem[**imm.**] (load from **imm.** address) | loadbuf 32 |
| storebuf **imm.** | mem[**imm.**] = buffer (store to **imm.** address) | storebuf 64 |

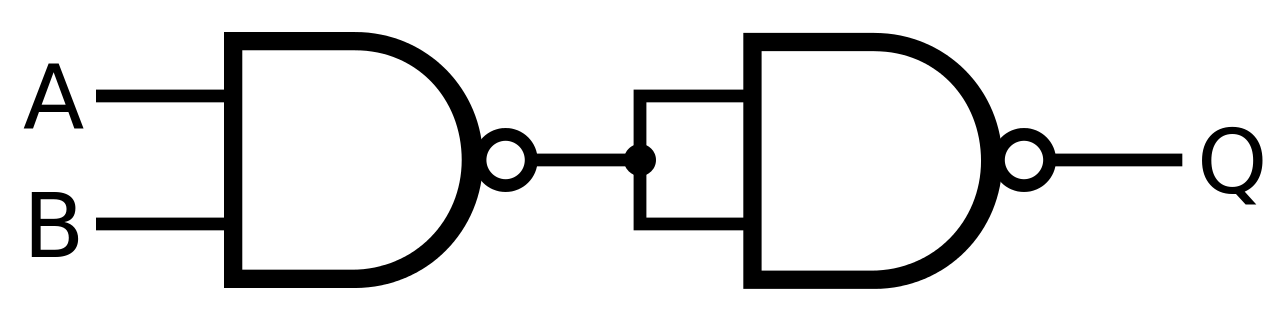
| **(a)** | **[2 pts]** If each instruction is encoded in 16 bits, and the addresses in **loadbuf** and **storebuf** instructions are unsigned immediates, what is the maximum number of address bits that can be in this ISA? (Assume opcodes are the same size in all instructions) What is the highest address a load or store instruction could refer to? | |
| --- | --- | --- |

Maximum number of address bits: \_\_\_\_\_\_**12**\_\_\_\_\_\_

9 instructions requires 4 bits to encode opcode. 16 - 4 = 12 remaining bits for the addresses in loads and stores

Highest loadable/storable address: \_\_\_\_\_\_**2^12 - 1 = 4095**\_\_\_\_\_\_\_\_

| **(b)** | **[5 pts]** Use the new ISA to implement the C pseudocode on the left. Solutions exist that don’t use all lines. The variable mem is an array of 32-bit values, thus each index corresponds to a single memory address. (*Hint*: NAND gates can create AND gates as shown in this diagram): | |
| --- | --- | --- |



| **C Pseudocode** | **New ISA** | | | | |
| --- | --- | --- | --- | --- | --- |
| if (mem[8] & 1) {  Branch to label “goal”  } | SOL’N 1 SOL’N 2  loadbuf 8  loadreg RPR  \_\_\_\_\_\_\_\_\_\_\_ nandi 1  \_\_\_\_\_\_\_\_\_\_\_ storereg RPR nandi -1 (0xFFFF)  \_\_\_\_\_\_\_\_\_\_\_ loadreg RSC <blank>  \_\_\_\_\_\_\_\_\_\_\_ nand <blank>  storereg RZR  \_\_\_\_\_\_\_\_\_\_\_ loadreg RSC  beq end  \_\_\_\_\_\_\_\_\_\_\_ loadreg RPR addi -1  beq goal  end | | | | |

| **(c)** | **[5 pts]** Use the new ISA to implement the C pseudocode on the left. Solutions exist that don’t use all lines. The variable mem is an array of 32-bit values, thus each index corresponds to a single memory address. | |
| --- | --- | --- |

| **C Pseudocode** | **New ISA** | | | | |
| --- | --- | --- | --- | --- | --- |
| mem[100] = (mem[64] + 3) << 2; | SOL’N 1 SOL’N 2 SOL’N 3  loadbuf 64  loadreg RPR  \_\_\_\_\_\_\_\_\_\_\_ addi 3 addi 3  \_\_\_\_\_\_\_\_\_\_\_ storereg RPR storereg RPR  \_\_\_\_\_\_\_\_\_\_\_ loadreg RSC loadreg RSC loadreg RSC  \_\_\_\_\_\_\_\_\_\_\_ add add add  \_\_\_\_\_\_\_\_\_\_\_ storereg RPR add add  \_\_\_\_\_\_\_\_\_\_\_ loadreg RSC add add  \_\_\_\_\_\_\_\_\_\_\_ add <blank> addi 12  storereg RPR  storebuf 100  storereg RPR  storebuf 100 | | | | |

1. A copy of ISA can be found in the supplemental material for ease of reference. [↑](#footnote-ref-0)