CIS 240 Fall 2019: Final

Please put all answers in the exam booklet and remember to number them clearly.

Question 1 {10 pts}

Part 1: {6 pts}
Your job is to design a PLA circuit that takes as input a 4 bit input I where I_0 is the LSB and I_3 is the MSB and returns a high output if and only if the number is a palindrome, which means that you get the same sequence of bits if you read it forward or backwards. As an example, 1001 and 0110 are 4-bit palindromes while 1100 is not. Please make sure to label your input signals clearly.

Part 2: {4 pts}
For this part you are going to redesign the circuit that you designed in part 1. This time it does not have to be a PLA but you are only allowed to use 2 input logic gates and you cannot have more than 5 of them. You can use any 2 input gates you wish, AND, OR, NAND, NOR, XOR, XNOR. More points will be given for designs that use fewer gates.

Question 2 {10 pts}

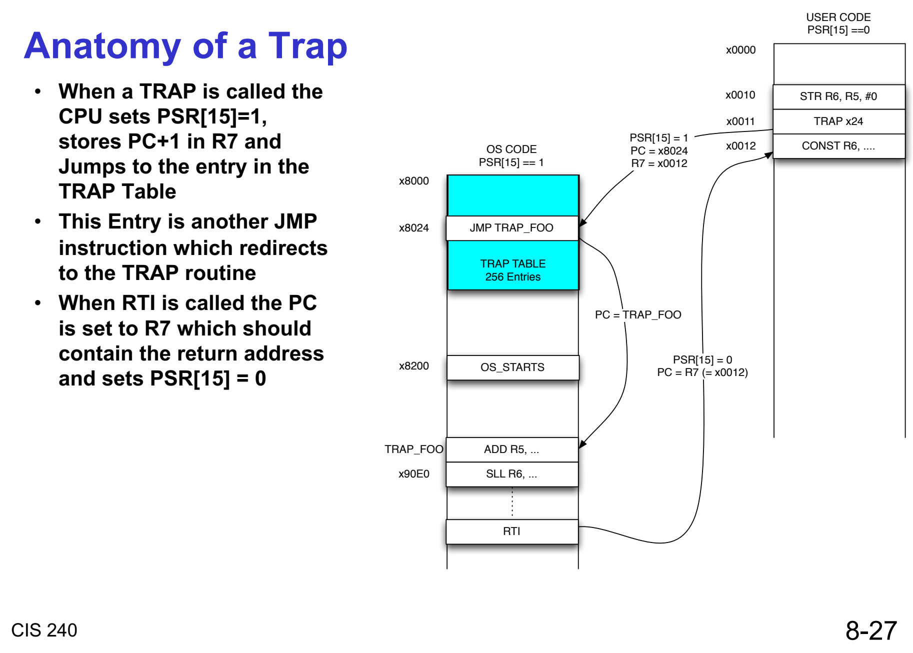
For your CIS 240 homework you write a TRAP routine that is designed to place a single character on the ASCII output console. It is called TRAP_PUTC and it works perfectly. You are then asked to write a second routine, TRAP_PUT_STRING that puts a sequence of characters on the ASCII console. Not wanting to waste effort, you figure that you can just call your TRAP_PUTC trap repeatedly as the following pseudo code shows.

```
TRAP_PUT_STRING
    WHILE (current character is not null – i.e. not at end of string)
        Send current character to screen by using TRAP
            instruction to call TRAP_PUTC
        Advance to next character
```

Would this idea of using the trap instruction to call TRAP_PUTC from inside TRAP_PUT_STRING work? Carefully explain why or why not.

Question 3 {10 pts}

Your cousin, Crazy Eddie, is looking over your slides for CIS 240 particularly the ones on how TRAPs are handled. When he comes across the slide shown below, he snorts and says: “This is way too complicated! Why don’t we just place the trap routine we want to call, TRAP_FOO, at the address that TRAP is going to jump to, x8024, and avoid the intermediate JMP instruction. That should work just fine.”



Anatomy of a Trap

- When a TRAP is called the CPU sets PSR[15]=1, stores PC+1 in R7 and Jumps to the entry in the TRAP Table
- This Entry is another JMP instruction which redirects to the TRAP routine
- When RTI is called the PC is set to R7 which should contain the return address and sets PSR[15] = 0

CIS 240 8-27

Question 1: Is Crazy Eddie right, would the trap routine work correctly if it were moved up to start at x8024 avoiding the JMP instruction.

Question 2: Would this change create other potential problems? Explain your answer.

Question 4 {10 pts}

Consider the following C program

```
#include <stdlib.h>

/* allocate a buffer to store 100 ints */
void allocate_buffer (int *ptr) {
  ptr = malloc(100*sizeof(int));
}

int main () {
  int i, *buf;

  allocate_buffer (buf);
  for(i=0; i < 100; ++i) buf[i] = i;
}
```

Does the program compile without issue? Could the code cause a segmentation violation or not? Explain your answer briefly.

Question 5 {10 pts}

Your cousin, Crazy Eddie, is really annoyed by the fact that he needs to FALIGN all of his subroutines in LC4 assembly. He thinks that alignment wastes space and that he should be able to start subroutines on any address he pleases. He thinks about it for a bit and then brightens and replaces this line in his assembly code.

```
JSR my_subroutine
```

With this sequence.

```
LEA R0, my_subroutine
JSRR R0
```

Will Crazy Eddie's code work even when the subroutine being called starts on an address that isn't a multiple of 16? What are the performance consequences of this change?

Question 6 {10 pts}

Consider the following portion of a C library which provides two routines one for adding an element to a doubly linked list and another for deleting an element. A few of the lines have been blanked out. Your job is to tell us what each of the missing lines should be to correctly complete the code.

```
#include <stdlib.h>

typedef struct list_elt_tag {
  int number;
  // Pointers to the previous and next elements in the list
  struct list_elt_tag *prev, *next;
} list_elt;


/*
 * Creates a new list element and pushes it on the front of the list
 * returns a pointer to the newly created element.
 */

list_elt *push (list_elt *first_elt, int number)
{
  list_elt *elt;

  // Allocate a new list element with malloc
  elt = malloc (sizeof(*elt));

  // If malloc fails end the program
  if (elt == NULL) {
    exit (1);
  }

  elt->number = number;

  MISSING_LINE_1;

  elt->next = first_elt;

  if (first_elt != NULL)
    first_elt->prev = elt;

  // return the pointer to the new list_elt
  return elt;
}
```

```
/*
 * delete: Deletes an element from the list returns a pointer to the new
 * first element of the list which may just be the old first element.
 */

list_elt *delete (list_elt *first_elt, list_elt *elt)
{
  list_elt *prev, *next;

  if (elt == NULL || first_elt == NULL) {
    return first_elt;
  }

  MISSING_LINE_2;

  next = elt->next;

  /* First we fix the pointers of the next and previous elements */
  if (prev) {
    prev->next = elt->next;
  }

  if (next) {
    MISSING_LINE_3;
  }

  MISSING_LINE_4;

  // Check if elt was the first element in the list
  if (elt == first_elt)
      MISSING_LINE_5;
  else
    return first_elt;
}
```

Question 7 (10 pts)

The following piece of C code was compiled with the lcc compiler.

```
void strcpy (char *src, char *dest, int n) {
  int i;
  for (i=0; i < n; ++i) {
    if (src[i]) {
      dest[i] = src[i];
    } else {
      break;
    }
  }
}
```

The resulting LC4 assembly code fragment is shown below. Several of the assembly instructions have been blacked out. Your job is to figure out what those assembly instructions must have been.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;strcpy;;;;;;;;;;;;;;;;;;;;;;;;;;
                .CODE
                .FALIGN
strcpy
        ;; prologue
        STR R7, R6, #-2        ;; save return address
        STR R5, R6, #-3        ;; save base pointer
        ADD R6, R6, #-3
        ADD R5, R6, #0
        ADD R6, R6, #-1        ;; allocate stack space for local variables
        ;; function body
        MISSING_INSN_1
        STR R7, R5, #-1
        JMP L5_final2019
L2_final2019
        LDR R7, R5, #-1
        LDR R3, R5, #3
        ADD R7, R7, R3
        LDR R7, R7, #0
        CONST R3, #0
        CMP R7, R3
        MISSING_INSN_2
        LDR R7, R5, #-1
        LDR R3, R5, #4
        ADD R3, R7, R3
        LDR R2, R5, #3
        ADD R7, R7, R2
        LDR R7, R7, #0
        MISSING_INSN_3
L7_final2019
L3_final2019
        LDR R7, R5, #-1
        MISSING_INSN_4
        STR R7, R5, #-1
L5_final2019
        LDR R7, R5, #-1
        LDR R3, R5, #5
        CMP R7, R3
        BRn L2_final2019
L4_final2019
L1_final2019
        ;; epilogue
        ADD R6, R5, #0          ;; pop locals off stack
        ADD R6, R6, #3          ;; free space for return address, base pointer, and return
value
        STR R7, R6, #-1         ;; store return value
        LDR R5, R6, #-3         ;; restore base pointer
        MISSING_INSN_5
        RET
```

Question 8 (10 pts)

For the final assignment you were asked to implement a compiler that converted programs written in the stack-based J language into assembly instructions. We want to add the following two new commands to the J language so that we can write programs that access a fixed array in global memory:

global_write : Pops the first value off the stack and uses that as an index into the global array. Then pops the next value off of the stack and stores this value into the indicated location in the array. Egs 7 2 global_write would store the value 7 into the location global_array[2]. The entries 7 and 2 would be removed from the stack.

global_read : Pops the first element off of the stack and uses this as an index into the global array. It stores the value at that location in the global array at the top of the stack. Egs. 3 global_read – would place the value stored in global_array[3] at the top of the stack, the 3 would be removed.

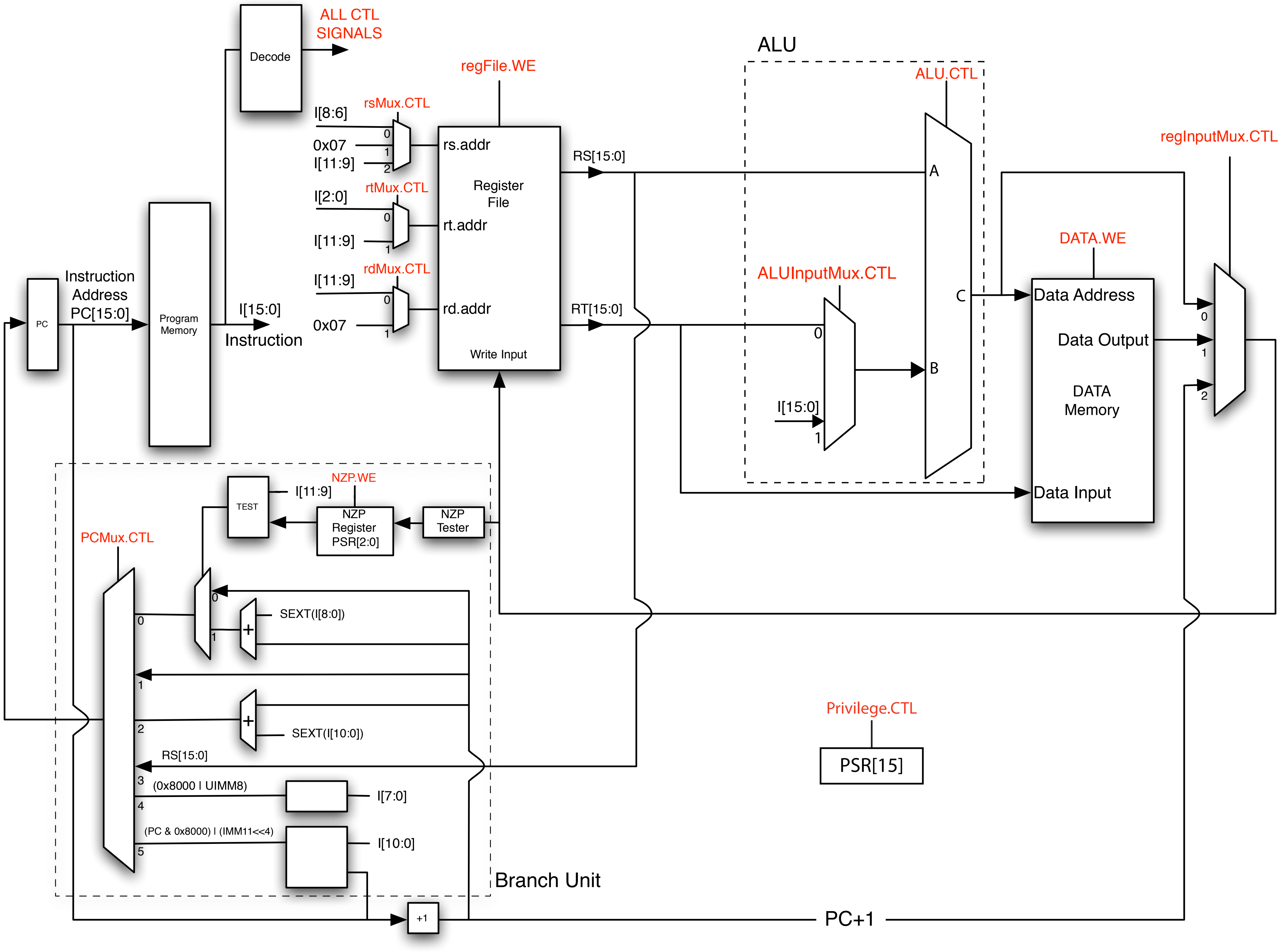
For each of these two J commands indicate the sequence of assembly instructions that your revised J compiler would emit when it encountered them in a program. You can assume that there is a label in the assembly code entitled "global_array" that marks the location where the global array will be stored in memory. Following C convention, the array indices start at 0 and you can assume that the at run time the array index values will be legal. Please use comments to indicate your thinking.

LC4 Instruction Set Reference v. 2017-01		
Mnemonic	**Semantics**	**Encoding**
NOP	PC = PC + 1	0000 000x xxxx xxxx
BRp <Label>	(P) ? PC = PC + 1 + (sext(IMM9) offset to <Label>)	0000 001i iiii iiii
BRz <Label>	(Z) ? PC = PC + 1 + (sext(IMM9) offset to <Label>)	0000 010i iiii iiii
BRzp <Label>	(Z|P) ? PC = PC + 1 + (sext(IMM9) offset to <Label>)	0000 011i iiii iiii
BRn <Label>	(N) ? PC = PC + 1 + (sext(IMM9) offset to <Label>)	0000 100i iiii iiii
BRnp <Label>	(N | P) ? PC = PC + 1 + (sext(IMM9) offset to <Label>)	0000 101i iiii iiii
BRnz <Label>	(N|Z) ? PC = PC + 1 + (sext(IMM9) offset to <Label>)	0000 110i iiii iiii
BRnzp <Label>	(N|Z|P) ? PC = PC + 1 + (sext(IMM9) offset to <Label>)	0000 111i iiii iiii
ADD Rd Rs Rt	Rd = Rs + Rt	0001 ddds ss00 0ttt
MUL Rd Rs Rt	Rd = Rs * Rt	0001 ddds ss00 1ttt
SUB Rd Rs Rt	Rd = Rs - Rt	0001 ddds ss01 0ttt
DIV Rd Rs Rt	Rd = Rs / Rt	0001 ddds ss01 1ttt
ADD Rd Rs IMM5	Rd = Rs + sext(IMM5)	0001 ddds ss1i iiii
MOD Rd Rs Rt	Rd = Rs % Rt	1010 ddds ss11 xttt
AND Rd Rs Rt	Rd = Rs & Rt	0101 ddds ss00 0ttt
NOT Rd Rs	Rd = ~Rs	0101 ddds ss00 1xxx
OR Rd Rs Rt	Rd = Rs | Rt	0101 ddds ss01 0ttt
XOR Rd Rs Rt	Rd = Rs ∧ Rt	0101 ddds ss01 1ttt
AND Rd Rs IMM5	Rd = Rs & sext(IMM5)	0101 ddds ss1i iiii
LDR Rd Rs IMM6	Rd = dmem[Rs + sext(IMM6)]	0110 ddds ssii iiii
STR Rt Rs IMM6	dmem[Rs + sext(IMM6)] = Rt	0111 ttts ssii iiii
CONST Rd IMM9	Rd = sext(IMM9)	1001 dddi iiii iiii
HICONST Rd UIMM8	Rd = (Rd & 0xFF) | (UIMM8 << 8) [1]	1101 dddx uuuu uuuu
CMP Rs Rt	NZP = sign(Rs - Rt) [2]	0010 sss0 0xxx xttt
CMPU Rs Rt	NZP = sign(uRs - uRt) [3]	0010 sss0 1xxx xttt
CMPI Rs IMM7	NZP = sign(Rs - sext(IMM7))	0010 sss1 0iii iiii
CMPIU Rs UIMM7	NZP = sign(uRs - UIMM7)	0010 sss1 1uuu uuuu
SLL Rd Rs UIMM4	Rd = Rs << UIMM4	1010 ddds ss00 uuuu
SRA Rd Rs UIMM4	Rd = Rs >>> UIMM4	1010 ddds ss01 uuuu
SRL Rd Rs UIMM4	Rd = Rs >> UIMM4	1010 ddds ss10 uuuu
JSRR Rs	R7 = PC + 1; PC = Rs	0100 0xxs ssxx xxxx
JSR <Label>	R7 = PC + 1; PC = (PC & 0x8000) | ((IMM11 offset to <Label>) << 4)	0100 1iii iiii iiii
JMPR Rs	PC = Rs	1100 0xxs ssxx xxxx
JMP <Label>	PC = PC + 1 + (sext(IMM11) offset to <Label>)	1100 1iii iiii iiii
TRAP UIMM8	R7 = PC + 1; PC = (0x8000 | UIMM8); PSR [15] = 1	1111 xxxx uuuu uuuu
RTI	PC = R7; PSR [15] = 0	1000 xxxx xxxx xxxx
Pseudo-Instructions		
RET	Return to R7	JMPR R7
LEA Rd <Label>	Store address of <Label> in Rd	CONST/HICONST
LC Rd <Label>	Store value of constant <Label> in Rd	CONST/HICONST
Assembler Directives		
.CODE	Current memory section contains instruction code	
.DATA	Current memory section contains data values	
.ADDR UIMM16	Set current memory address value to UIMM16	
.FALIGN	Pad current memory address to next multiple of 16	
.FILL IMM16	Current memory address's value = IMM16	
.STRINGZ "String"	Expands to a .FILL for each character in "String"	
.BLKW UIMM16	Reserve *UIMM16* words of memory from the current address	
<Label> .CONST IMM16	Associate *<Label>* with IMM16	
<Label> .UCONST UIMM16	Associate *<Label>* with UIMM16	

0101: opcode or sub-opcode	ddd: destination register	sss: source register 1	ttt: source register 2
	iii: signed immediate value	uuu: unsigned immediate value	xxx: "don't care" value

[1] In this case the source and destination register are one and the same as HICONST reads and modifies the same register.
[2] sign(Rs- Rt) results in one of three values: +1, 0, or -1, which set the appropriate bit in the NZP register.
[3] sign(uRs- uRt) indicates that Rs and Rt are treated as unsigned values.
[4] The NZP register is updated on any instruction that writes to a register, and on CMPx instructions.

Single Cycle Implementation of the LC4 ISA



Description of Control Signals in Single Cycle Implementation of the LC4 ISA

Signal Name	# of bits	Value	Action
PCMux.CTL	3	0	Value of NZP register compared to bits I[11:9] of the current instruction if the test is satisfied then the output of TEST is 1 and NextPC = BRANCH Target, (PC+1) + SEXT(IMM9); otherwise the output of TEST is 0 and NextPC = PC + 1
		1	Next PC = PC+1
		2	Next PC = (PC+1) + SEXT(IMM11)
		3	Next PC = RS
		4	Next PC = (0x8000 | UIMM8)
		5	Next PC = (PC & 0x8000) | (IMM11 << 4)
rsMux.CTL	2	0	rs.addr = I[8:6]
		1	rs.addr = 0x07
		2	rs.addr = I[11:9]
rtMux.CTL	1	0	rt.addr = I[2:0]
		1	rt.addr = I[11:9]
rdMux.CTL	1	0	rd.addr = I[11:9]
		1	rd.addr = 0x07
regFile.WE	1	0	Register file not written
		1	Register file written: rd.addr indicates which register is updated with the value on the Write Input
regInput.Mux.CTL	2	0	Write Input = ALU output
		1	Write Input = Output of Data Memory
		2	Write Input = PC + 1
NZP.WE	1	0	NZP register not updated
		1	NZP register updated from Write Input to register file
DATA.WE	1	0	Data Memory not written
		1	Data Input written into location on Data Address lines
Privilege.CTL	2	0	PSR[15] = 0 – Clear privilege bit
		1	PSR[15] = 1 – Set privilege bit
		2	PSR[15] unchanged – no change to privilege bit
ALUInputMux.CTL	1	0	B[15:0] = RT[15:0] – B input to ALU = RT
		1	B[15:0] = I[15:0] – B input to ALU = Instruction Word

Signal Name	# of bits	Value	Action
ALU.CTL	6		
Arithmetic Ops		0	C = A + B : Addition
		1	C = A * B : Multiplication
		2	C = A - B : Subtraction
		3	C = A / B : Division
		4	C = A % B : Modulus
		5	C = A + SEXT(B[4:0])
		6	C = A + SEXT(B[5:0])
Logical Ops		8	C = A AND B : Bitwise Logical Product
		9	C = NOT A: Bitwise Negation
		10	C = A OR B: Bitwise Logical Sum
		11	C = A XOR B: Bitwise Exclusive OR
		12	C = A AND SEXT(B[4:0])
Comparator Ops		16	C = signed-CC(A-B) [-1, 0, +1]
		17	C = unsigned-CC(A-B) [-1, 0, +1]
		18	C = signed-CC(A-SEXT(B[6:0])) [-1, 0, +1]
		19	C = unsigned-CC(A-SEXT(B[6:0])) [-1, 0, +1]
Shifter Ops		24	C = A << B[3:0] : Shift Left Logical
		25	C = A >>> B[3:0] : Shift Right Arithmetic
		26	C = A >> B[3:0] : Shift Right Logical
Constant Ops		32	C = SEXT(B[8:0])
		33	C = (A & 0xFF) | (B[7:0] << 8)