CIS 240 Fall 2018: Final

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

Question 1 {10 pts}

Your job is to design a gate level combinational circuit that takes as input a 3 bit unsigned number and produces as output that number plus 1 as a 3 bit unsigned value, so if the input number is 5 the output should be 6, if the input is 3 the output should be 4, if the input is 7 the output should be 0 because of wraparound. The three inputs should be labeled I_2, I_1 and I_0 where I_2 is the MSB and I_0 is the LSB. Similarly, the output bits must be labeled 0_2, O_1 and O_0.

Part 1 {4 pts}: Produce a table showing what the output should be for every possible input.
Part 2 {6 pts}: Design a circuit that performs the operation. Make sure to clearly label all inputs and outputs. You can use any gates you want, including xor gates. More points will be given for answers that use fewer gates so think carefully before implementing.

Question 2 (10 pts)
The following piece of C code was compiled with the lcc compiler.

```
int fact1 (int n) {
  if (n <= 0)
    return 1;
  else
    return n * fact1(n-1);
}

int fact2 (int n) {
  int i, output = 1;

  for (i=2; i<=n; ++i) {
    output *= i;
  }

  return output;
}
```

The resulting LC4 assembly code fragment is shown below. Five of the assembly instructions have been blacked out. Your job is to figure out what those 5 assembly instructions must have been.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;fact1;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                .CODE
                .FALIGN
fact1
        ;; 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
        LDR R7, R5, #3
        CONST R3, #0
        CMP R7, R3
        BRp L2_Final_2018_2
        CONST R7, #1
        JMP L1_Final_2018_2
L2_Final_2018_2
        LDR R7, R5, #3
        STR R7, R5, #-1
        ADD R3, R7, #-1
        ADD R6, R6, #-1
        MISSING_INSN_1
        MISSING_INSN_2
        LDR R7, R6, #-1         ;; grab return value
        ADD R6, R6, #1          ;; free space for arguments
        LDR R3, R5, #-1
        MUL R7, R3, R7
L1_Final_2018_2
        ;; 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
        LDR R7, R6, #-2         ;; restore return address
        RET

;;;;;;;;;;;;;;;;;;;;;;;;;;;;fact2;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                .CODE
                .FALIGN
fact2
        ;; prologue
        STR R7, R6, #-2         ;; save return address
        STR R5, R6, #-3         ;; save base pointer
```

```
        ADD R6, R6, #-3
        ADD R5, R6, #0
        MISSING_INSN_3
        ;; function body
        CONST R7, #1
        STR R7, R5, #-2
        CONST R7, #2
        STR R7, R5, #-1
        MISSING_INSN_4
L5_Final_2018_2
        LDR R7, R5, #-2
        LDR R3, R5, #-1
        MUL R7, R7, R3
        STR R7, R5, #-2
L6_Final_2018_2
        LDR R7, R5, #-1
        ADD R7, R7, #1
        STR R7, R5, #-1
L8_Final_2018_2
        LDR R7, R5, #-1
        LDR R3, R5, #3
        CMP R7, R3
        MISSING_INSN_5
        LDR R7, R5, #-2
L4_Final_2018_2
        ;; 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
        LDR R7, R6, #-2     ;; restore return address
        RET
```

Question 3 {10 pts}

In Question 2 the two functions, fact1 and fact2, compute the same quantity, the factorial of the input number. Which of the two compiled function contains more assembly instructions? On typical inputs, which version of the function would run faster (ie take fewer clock cycles) on an LC4 single cycle processor. Explain your answer briefly.

Question 4 {5 pts}

Explain briefly why the lcc compiler inserts a .FALIGN assembly directive at the beginning of every compiled function.

Question 5 {10 pts}

When the lcc compiler compiles an if statement like this:

```
if (some test) {
    code block inside if statement
}
```

it uses a BRANCH statement to implement the required control flow. What constraints, if any, does that place on the size of the code block inside the if statement?

Question 6 {10 pts}

Which of the following operations will cause PennSim to report an error?

1. Trying to execute code in the OS code section with privilege bit set to 0
2. Trying to execute code in the user data section with privilege bit set to 1
3. Trying to store into OS data section with privilege bit set to 1
4. Trying to execute code in USER code section with privilege bit set to 1
5. Trying to store into USER data section with privilege bit set to 1

Question 7 {15 pts}

Your cousin, Crazy Eddie, has decided to add a new instruction to the LC4 instruction set with an opcode of `0011`. Here are the settings for the control signals for this operation.

	PCMux.CTL	rsMux.CTL	rtMux.CTL	rdMux.CTL	regFile.WE	regInputMux.CTL	NZP.WE	DATA.WE	Privilege.CTL	ALUInputMux.CTL	ALU.CTL
FOO	1	1	1	1	1	0	1	1	2	1	6

Your LC4 microprocessor is about to execute the following 16 bit instruction
`0011011000000001`

The table below shows the state of the LC4 processor right before this new instruction is executed. Your job is to fill in the table to indicate what the state will be after the instruction is executed. All values are given in hex, your answers should be as well. Please put the answer in your answer booklet.

	PC	PSR	R0	R1	R2	R3	R4	R5	R6	R7
Before	001F	0002	0003	0011	0022	0009	0017	0101	0013	004F
After	????	????	????	????	????	????	????	????	????	????

Are there any other relevant changes to the machine state?

Explain how this new instruction could be used to accelerate the implementation of the memset() function on an LC4 system. That is explain how would you use this new instruction to make a faster implementation of memset than you could before.

```
void *memset ( void * ptr, int value, size_t num );
```

Fill block of memory:
Sets the first *num* slots of the block of memory pointed by *ptr* to the specified *value*.

Question 8 {10 pts}

You are tasked with writing a program that will maintain a list of integers sorted in ascending order. We are providing some C code that does this but we have blanked out some of the lines. Your job is to tell us what these missing lines should be. (HINT: The missing lines are all assignment statements.)

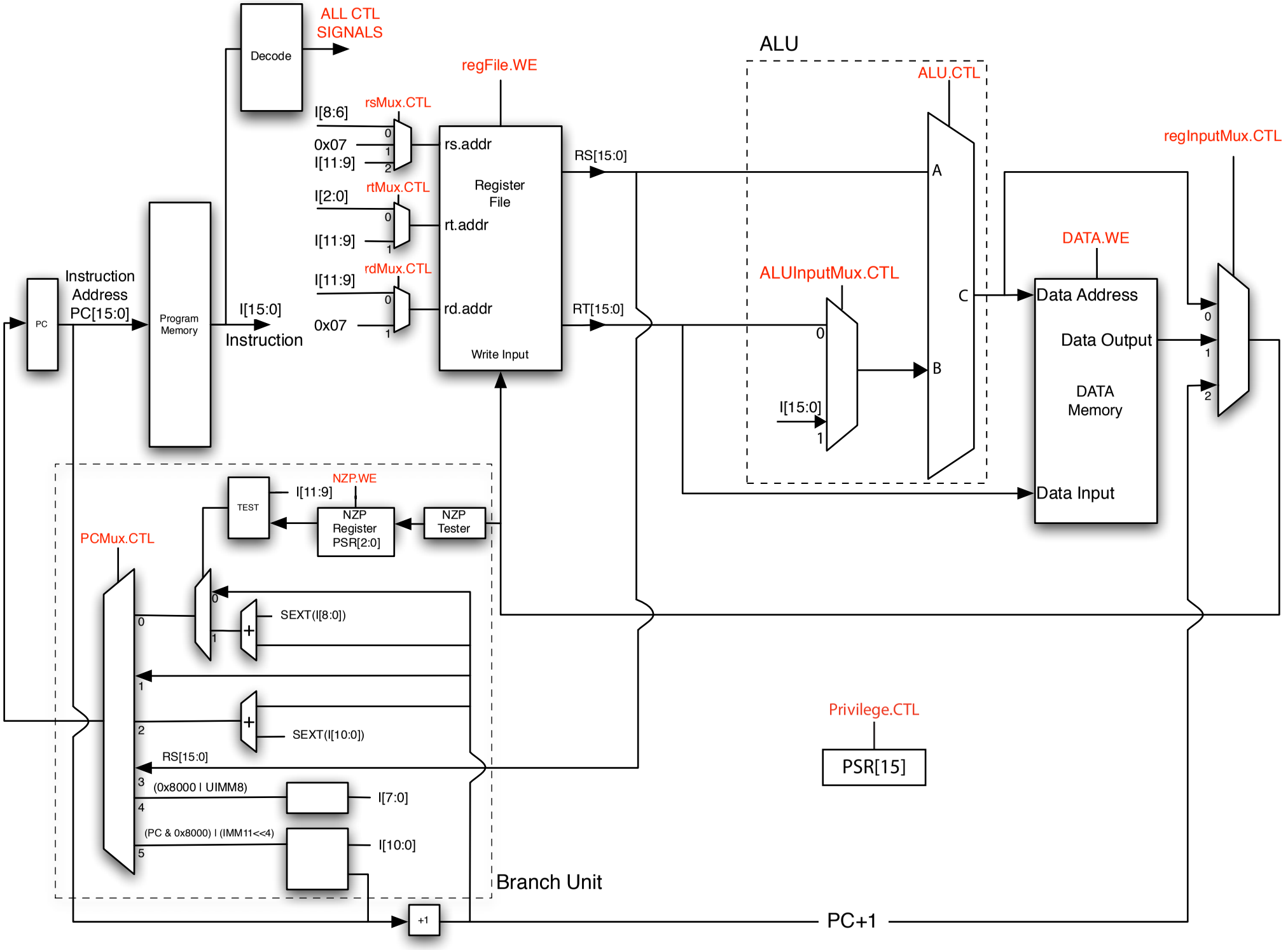
```
#include <stdio.h>
#include <stdlib.h>

typedef struct list_elt_tag {
  int elt;
  struct list_elt_tag *next;
} list_elt;

list_elt *LIST = NULL; // Initialize the list to empty

// Insert the number n on the list sorted in ascending order
void insert_entry_sorted (int n) {
  list_elt *entry, *new_entry;

  MISSING_LINE_1

  if (new_entry == NULL) exit(2);

  new_entry->elt = n;

  if ((LIST == NULL) || (LIST->elt >= n)) {
    MISSING_LINE_2
    MISSING_LINE_3
  } else {
    entry = LIST;

    while ((entry->next) && (((entry->next)->elt) < n)) {
      MISSING_LINE_4
    }

    MISSING_LINE_5
    MISSING_LINE_6
  }
}
```

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 ddd i 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 & OxFF) | (B[7:0] << 8)