ECE190 Exam 2, Spring 2007

Thursday 29 March

Name:

- **Be sure that your exam booklet has 14 pages.**
- **The exam is <u>meant</u> to be taken apart!**
- **Write your name at the top of each page.**
- **This is a closed book exam.**
- **You may not use a calculator.**
- **You are allowed two 8.5×11" sheet of handwritten notes.**
- **Absolutely no interaction between students is allowed.**
- **Show all of your work.**
- **Don't panic, and good luck!**

" 'Tis pity if the case require ... We speak the literal ..."
—R. Frost

Problem 1	20 points	_____
Problem 2	15 points	_____
Problem 3	25 points	_____
Problem 4	20 points	_____
Problem 5	20 points	_____
Total	100 points	_____

Problem 1 (20 points): Short Answers

Please answer concisely. If you find yourself writing more than a few words or a simple drawing, your answer is probably wrong.

Part A (6 points): Variable 'X' is stored in memory location with label 'X'. Write LC-3 assembly code to place the value of each of the C expressions into R0, assuming that the memory location of 'X' is within a 9-bit offset. Each of your answers should be a single LC-3 instruction.

i. (2 points) X

ii. (2 points) &X

iii. (2 points) *X

Part B (4 points): When compiling C for an LC-3 processor, how many memory locations are needed to represent a single variable of each of the following types?

i. (2 points) `float`

ii. (2 points) `float*`

Problem 1, continued:

Part C (5 points): Write the output of the program below.

```
int main ()
{
    int i, j, k;

    j = 0;
    i = 7;
    k = (j++ && ++i);
    printf ("%d %d %d\n", i, j, k);
}
```

Part D*** (5 points): Certain instruction set architectures, such as x86, use variable-length instruction encodings, such that some instructions require only a byte, while others require more than one byte. In particular, conditional branches can take short (say, two-byte) forms if the branch target is nearby, or long (say, four-byte) forms if the branch target is more distant.

In terms of the two-pass assembly process discussed in class and in the textbook, explain why such variability can pose a problem for an assembler.

Problem 2 (15 points): Systematic Decomposition

As you may already know, a number in base 10 is divisible by nine if and only if the sum of its digits are also divisible by nine. For example, consider the number 729. Since $7+2+9=18$ (which is divisible by nine), 729 itself must be divisible by nine. Similarly, 1857382992 is divisible by 9, since $1+8+5+7+3+8+2+9+9+2=54$, and $5+4=9$.

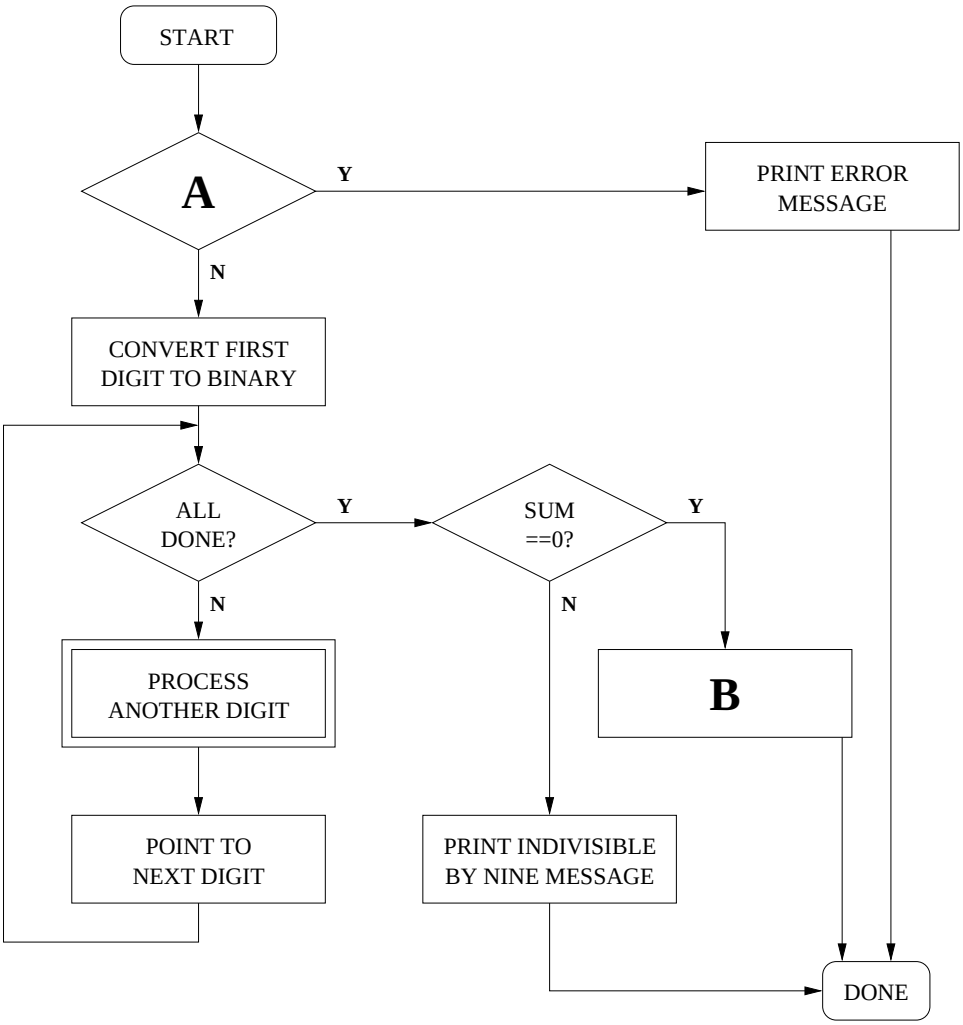
In this problem, you will implement an algorithm that applies this test to check whether a NUL-terminated ASCII string of digits representing a very large decimal number is divisible by 9.

For simplicity, you will leverage the associativity of addition by recognizing that the order of subsequent additions is not important to the final answer, allowing you to completely process each digit into a running sum before moving on to the next digit.

With "823436," for example, you start with 8, then add 2 to obtain 10. Before moving on to the third digit, however, you break 10 into 1 and 0 and add them up to obtain 1. After adding 3, the running sum of 4 requires only one digit, so we can go ahead and add the 4 from the string to obtain 8. Again, only a single digit, so we add the 3 to obtain 11. Now we again break the 11 into 1 and 1 and add them to obtain 2 before processing the final digit. Finally, adding 6 to our running sum of 2 gives 8, which is not divisible by 9.

It is important to realize that the process just described is entirely equivalent (but much easier to automate!) than adding $8+2+3+4+3+6$ to obtain 26, then repeating the process to obtain 8.

The flow chart below is used for the problem on the next page.



Problem 2, continued:

In **Parts A** through **C**, you must complete the systematic decomposition of the given problem. Assume that all non-NUL characters in the string provided are ASCII digits (0 through 9, encoded as x30 through x39).

Part A (2 points): Write a description of the test to be made in the box marked "A" in the flow chart on the previous page.

Part B (2 points): Write a description of the task to be performed in the box marked "B" in the flow chart on the previous page.

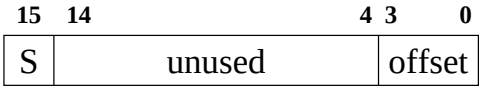
Part C (8 points): Decompose the double-lined box marked "PROCESS ANOTHER DIGIT" in the flow chart on the previous page such that each task or test can be accomplished with individual LC-3 instructions (not counting branches). Hint: In your code, you should keep the running sum between 0 and 8 after processing each digit. Note that sums of 0 and 9 are equivalent for the human-based testing process.

Part D* ** (3 points): A friend suggests that you could just use a running sum from 1 to 9 and check whether the final sum is equal to 9 instead of 0. What strings would produce inappropriate results with such a change?

Problem 3 (25 points): I/O, Assembly, and C

Part A (10 points): One of your TAs has implemented a new storage device for ECE 395 and needs you to write a simple interface in LC-3 assembly to write to the device.

The device has a status register that is mapped into memory location xFE10. The device signals that it is ready to receive data by setting the most significant bit of this register, indicated by S in the figure below.



The device reads data by maintaining an array of 16 data registers. These are memory-mapped registers; the first register is mapped to xFE12, the next is mapped to xFE14, and so on, with each data register mapped two locations away from the previous one.

When the device signals that it is ready to receive data, it places a four-bit offset into bits [3:0] of its status register, as shown in the diagram above. These bits are an offset into the array of data registers; thus, an offset of 0 corresponds to the first data register, and an offset of 15 corresponds to the last data register.

You will write the STORE_DATA subroutine. The caller will place the data to be written in R0 before calling your subroutine. Your routine must wait until the device is ready, then store the contents of R0 to the data register indicated by the offset field. *You may not assume that unused bits are equal to zero.* You may assume that R0, R1, R2, and R3 are caller-saved.

Write your subroutine below. Some helpful constants, which you may use freely, have been provided for you.

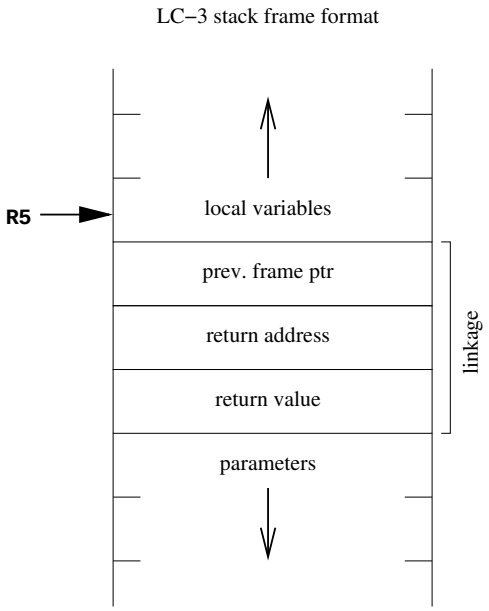
```
STORE_DATA    ; subroutine that you must write
```

```
RET

STATUS_ADDR .FILL xFE10
```

Problem 3, continued:

The following LC-3 assembly code was generated by compiling the C function `int my_op (int value)`. The canonical LC-3 stack frame (activation record) format appears in the diagram to the right.

```
; code to set up stack frame and save registers omitted

            AND   R0, R0, #0
            STR   R0, R5, #-1
            ADD   R0, R0, #1
            STR   R0, R5, #0

LOOP        LDR   R0, R5, #0
            BRz   MORE_FUN
            LDR   R1, R5, #4
            AND   R3, R3, #0
            AND   R2, R0, R1
            BRz   FUN
            ADD   R3, R3, #1

FUN         LDR   R0, R5, #-1
            ADD   R0, R0, R3
            STR   R0, R5, #-1

            LDR   R0, R5, #0
            ADD   R0, R0, R0
            STR   R0, R5, #0
            BRnzp LOOP

MORE_FUN    LDR   R0, R5, #-1
            STR   R0, R5, #3

; code to restore registers and tear down stack frame omitted
```



Part B (10 points): Based on the LC-3 assembly code above, fill in the body of the C function below.

```
int my_op (int value)
{




}
```

Part C (5 points): What does this function do?

Problem 4 (20 points): C and Stack Frames

This question focuses on the program below, and particularly on the stack frames (also called activation records) that are used by each function in the program.

Consider the following pair of C functions.

```
int bar (int x, int y)
{
    return ((x - y) * (x - y));    /* line A */
}

int foo (int a)
{
    int b, c;                      /* line B */

    printf ("Enter input:  ");
    scanf ("%d", &b);              /* line C */

    c = a - b;

    a = bar (b, c);

    return (a + b + c);            /* line D */
}

int main ()
{
    int q = foo (5);

    printf ("%d\n", q);
    return 0;
}
```

The program is run. When prompted for input, the user types "3". For both parts of this question, you are asked to fill in a diagram to show the stack frames (activation records) for all appropriate functions.

The diagrams are on the following two pages.

Problem 4, continued:

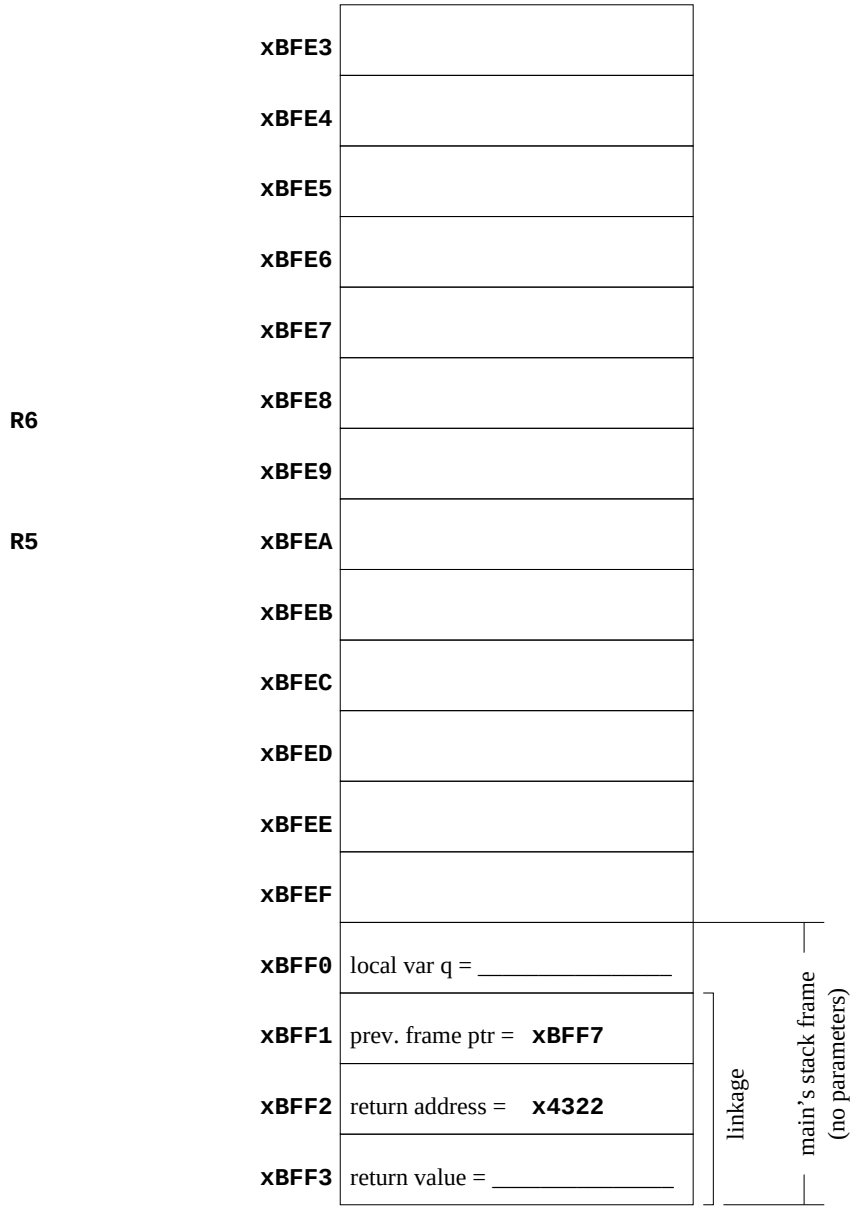
Part A (12 points): Using the diagram, fill in the state of the stack **just before line A** executes.

The stack frame for the `main` function is shown at the bottom of each diagram, and a canonical frame diagram was provided in **Problem 3** of this exam. During execution of `main`, the stack pointer `R6=xBFF0`, and the frame pointer `R5=xBFF0`.

Draw arrows to indicate the values of `R6` and `R5` at the point of program execution just described. For each memory location included in the stack (*i.e.*, between the stack pointer and the bottom of the figure), label the location with the type of information **and** the value stored there. If a memory location's value **cannot** be known, put a question mark by the description, *e.g.*, "x=?".

Only draw current activation records. If a function has already returned after being called, do not draw its activation record. **Do not mark or label any locations above the stack pointer, even if you know the values in those locations!**

The address of the `JSR bar` instruction in `foo` is `x3056`.



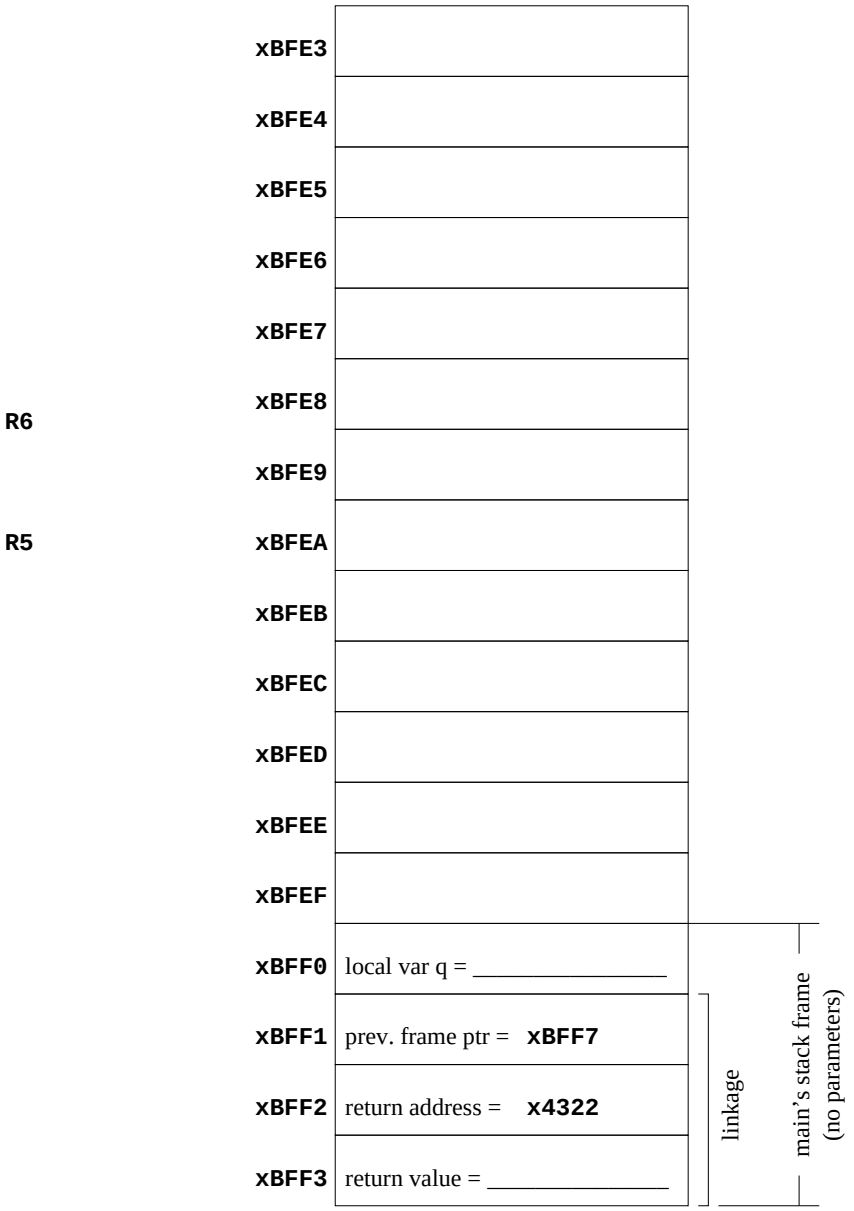
Problem 4, continued:

Part B (8 points): Using the diagram, fill in the state of the stack **just after line D** executes, i.e., just before `foo`'s stack frame is torn down and the function returns.

The stack frame for the `main` function is shown at the bottom of each diagram, and a canonical frame diagram was provided in **Problem 3** of this exam. During execution of `main`, the stack pointer `R6=xBFF0`, and the frame pointer `R5=xBFF0`.

Draw arrows to indicate the values of `R6` and `R5` at the point of program execution just described. For each memory location included in the stack (*i.e.*, between the stack pointer and the bottom of the figure), label the location with the type of information **and** the value stored there. If a memory location's value **cannot** be known, put a question mark by the description, *e.g.*, "x=?".

Only draw current activation records. If a function has already returned after being called, do not draw its activation record. **Do not mark or label any locations above the stack pointer, even if you know the values in those locations!**



Problem 5 (20 points): C Programming

Fill in the missing code and answer the following questions. Some code has been provided for you. Each answer should be at most a few lines of code; rethink your answer if you are writing more than a few lines.

Part A (4 points): The function below does not perform as expected.

```
/* assuming that num >= 0 and mult > 0, return num rounded down to the nearest multiple of mult */
int round_down (int num, int mult)
{
    return (   mult   *   num   /   mult   );
}
```

What does the function really return? A small change will fix the function. Make the change.

Part B (4 points): Fill in the first argument to make the function work as expected.

```
/* assuming that num >= 0 and mult > 0, return num rounded up to the nearest multiple of mult */
int round_up (int num, int mult)
{

    return round_down ( _____ , mult);
}
```

Part C (6 points): Fill in the body of this function that rounds the number `num` to the nearest multiple of `mult`. For example:

`round (7, 4)` should return 8.
`round (6, 4)` should return 8 as well (round midpoints up).
`round (5, 4)` should return 4.

You must call the functions `round_up` and `round_down` in your answer. Assume that they work as described.

```
/* assuming that num >= 0 and mult > 0, return num rounded to the nearest multiple of mult */
int round (int num, int mult)
{
    int answer;
    if ( (num % mult) < _____ ) {


        _____
    } else {


        _____
    }
    return answer;
}
```

Problem 5, continued:

Your wonderful TAs advocate a new grading function that rounds all grades up to the nearest letter grade! They write a test program to try it out.

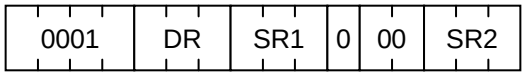
```
int grade_student1 (int grade)
{
    grade = round_up (grade, 10);
    return grade;
}

int grade_student2 (int* grade)
{
    *grade = round_up (*grade, 10);
    return *grade;
}

int main ()
{
    int s1 = 89;
    int s2 = 77;
    int s3 = 54;
    int s4 = 42;
    int grade;

    grade = s1;
    grade_student1 (grade);
    s1 = grade;

    grade = s2;
    s2 = grade_student1 (grade);
    s2 = grade;

    grade = s3;
    s3 = grade_student2 (&grade);
    s3 = grade;

    grade = s4;
    grade_student2 (&grade);
    s4 = grade;

    printf ("s1:  %d s2:  %d s3:  %d s4:  %d\n", s1, s2, s3, s4);
    return 0;
}
```

Part D (6 points): Write the program's output.

Use this page as scratch paper.

NOTES: RTL corresponds to execution (after fetch!); JSRR not shown

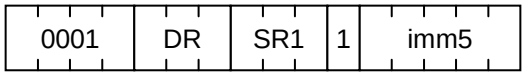
ADD



ADD DR, SR1, SR2

DR ← SR1 + SR2, Setcc

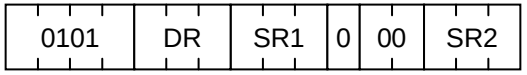
ADD



ADD DR, SR1, *imm5*

DR ← SR1 + SEXT(imm5), Setcc

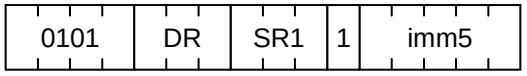
AND



AND DR, SR1, SR2

DR ← SR1 AND SR2, Setcc

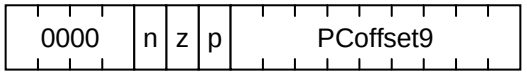
AND



AND DR, SR1, *imm5*

DR ← SR1 AND SEXT(imm5), Setcc

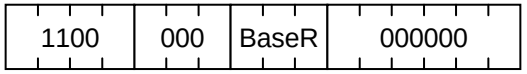
BR



BR{nzp} *PCoffset9*

((n AND N) OR (z AND Z) OR (p AND P)):
PC ← PC + SEXT(PCoffset9)

JMP



JMP BaseR

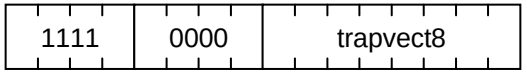
PC ← BaseR

JSR

JSR *PCoffset11*

R7 ← PC, PC ← PC + SEXT(PCoffset11)

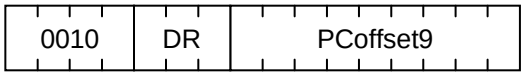
TRAP



TRAP *trapvect8*

R7 ← PC, PC ← M[ZEXT(trapvect8)]

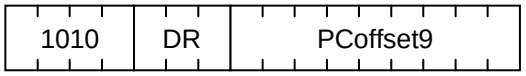
LD



LD DR, *PCoffset9*

DR ← M[PC + SEXT(PCoffset9)], Setcc

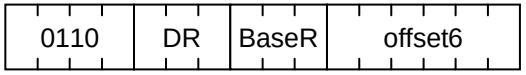
LDI



LDI DR, *PCoffset9*

DR ← M[M[PC + SEXT(PCoffset9)]], Setcc

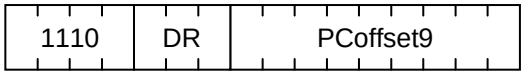
LDR



LDR DR, BaseR, *offset6*

DR ← M[BaseR + SEXT(offset6)], Setcc

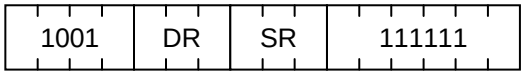
LEA



LEA DR, *PCoffset9*

DR ← PC + SEXT(PCoffset9), Setcc

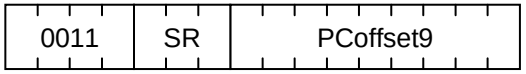
NOT



NOT DR, SR

DR ← NOT SR, Setcc

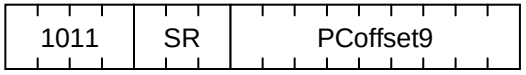
ST



ST SR, *PCoffset9*

M[PC + SEXT(PCoffset9)] ← SR

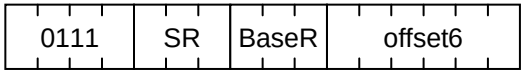
STI



STI SR, *PCoffset9*

M[M[PC + SEXT(PCoffset9)]] ← SR

STR



STR SR, BaseR, *offset6*

M[BaseR + SEXT(offset6)] ← SR