# COMP 3050-201 Computer Architecture

Homework #4 Spring, 2019

• This assignment is due no later than midnight (11:59:59 PM) **March 20**.

• All submissions must be made electronically using the submit command as described below.

- **File 1:** A short **write-up** that **first** specifies what you think your **degree of success** with a project is (**from 0% to 100%**), followed by a brief discussion of your approach to the project along with a **detailed description** of any problems that you were **not** able to resolve for this project. **Failure to specifically provide this information will result in a 0 grade** on your assignment. If you do **not disclose** problems in your write-up and problems are detected when your program is tested, you will receive a grade of 0. **Make sure that you include your email address in your write-up so that the corrector can email you your grade.**

- **File(s) 2(a, b, c, …)**: For this assignment you must submit 3 files which will include:
  - your **complete new commented microcode source code**, in the form of an **.mc** file, as discussed in class.
  - your **complete new commented flex source file**, in the form of an **.ll** file, as discussed in class
  - your **complete new commented masm C source file**, in the form of an **.c** file, as discussed in class
- **File 3:** A **make file** to build your assignment. This file must be named **Makefile**. (The makefile in this case will need to direct the build of the masm assembler after modifications.)

- **File 4:** A file that includes your **resulting output** run(s) from your project. This is a simple text file that shows your output, but make sure that you **annotate it** so that it is self-descriptive and that all detailed output is well identified. Use the **script** command from the shell as discussed in class to capture all screen output to a file. This file must show:
  - Your building of the new masm tool
  - Your assembly into object files of each of the 3 test files
  - Your mic1 execution using the new microcode and the resulting debug dump of each of the 3 test files
  - Make sure each of these files is appropriately annotated for readability

• The files described above should be the only files placed in one of your subdirectories, and this subdirectory should be the target of your submit command, this time specifying “hw4”. See the on-line file **Assignment_Submit_Details.pdf** for specific directions.

- You will provide a new version of the MIC-1 microcode which will support the implementation of **three new MACRO machine instructions:**

  **RSHIFT <argument is value 0 – 15>**
  Right shift the current accumulator by the argument number of bits. This is a zero-fill operation and the argument must be a value between **0 and 15** (a four bit magnitude only value, located in the least significant 4 bits)

  **MULT <argument is value 0 – 63>**
  **multiply** the value at the **top of the stack** by the six **bit immediate value** in the instruction (remember, we're using 10 bit opcodes here). The **result of the multiplication should replace the value at the top of the stack,** provided that no overflow occurred. If the multiplication **would overflow a 16 bit result**, the top of the stack **must not be changed**. After the operation, the **accumulator** must be set to **0 if the multiplication succeeded, and -1 if an overflow occurred**.

  **DIV <no arguments>**
  **divide** the 16 bit 2s complement number at the top stack location **SP** (the dividend) by the 16 bit 2s complement number at the location just under the top location **SP+1** (the divisor), and push on **two new locations** with **SP-1 having the (unsigned) remainder** of the division and **SP-2 (new top of stack)** having the **16 bit 2s complement quotient**. If the absolute value of the divisor is greater than the absolute value of the dividend, **the quotient should be set to 0 and the (unsigned) remainder set to the absolute value of the dividend,** and if the divisor is **equal to 0 (this is an illegal case)** the **remainder** should be set to **-1** and the **quotient** should be set to **0**. After the operation, the **accumulator** must be set to **0 if the division was legal, and -1 if an attempt to divide by 0.**

- The approach we'll use here is to extend the op-code field from 7 bits (as used in the INSP and DESP instructions for example) to a total of 10 bits, using the eighth bit as a gateway (i.e., bit-on a means to break out to a 10 bit op-code, while bit-off means a 7 bit op-code)

- Since bit 8 is used as a gateway, the 10 bit op-code will provide us with 4 new combinations, and we'll use them as follows:

  - `1111111 1 00 mmmmmm for` **`MULT`** `(6 bit multiplier field mmmmmm as operand)`

- `1111111 1 01 xxssss for` **`RSHIFT`** `(4 bit shift field ssss as operand)`
- `1111111 1 10 xxxxxx for` **`DIV`** `(no operand)`
- `1111111 1 11 xxxxxx for` **`HALT`** `(no operand)`

- A collection of test masm programs are included with this assignment, and you must show that you can assemble, run and provide correct output for each of these tests. The source code for each test will be placed on the class web site

- The attached examples shown below should serve as a template for the work flow of this assignment.

**cp ~bill/cs305/mcc .**

- Re-write the mic1 microcode and build a new promfile source that includes support for the required new instructions. You must comment this microcode to show your additions
- Translate your new microcode source into a 32 bit microcode .dat file using the **mcc** program found in ~bill/cs305:

  $ ~bill/cs305/**mcc** my_new_microcode.mc > prom_new.dat
- Re-write the masm assembler to include support for your new macro instructions, providing comments in the code and the flex specification file to highlight your new code
- Build your new **masm** program using something like the Makefile at:

  ~bill/cs305/NandRshiftExample/**Makefile_nand_rshift**

  This will build a new masm program from your flex specification changes and your changes to the mic1symasm.c files. This new masm should now be able to recognize and generate mic1 machine code for the newly implemented instructions (MULT, DIV, RSHIFT and HALT).
- Use your new **masm** assembler to assemble the supplied assembly test programs into object files that can be run with the mic1 emulator. Test code available on-line at:

  http://www.cs.uml.edu/~bill/cs305/assignment_4_help_dir/

  or directly on mercury at:

  ~bill/cs305
- Run the test object files in the mic1 emulator using your new microcode and using the embedded de-bugger to display your results for each test

• Your submitted directory should contain:

- Your write-up first (make sure you discuss all the steps you took to get this assignment completed)
- Your commented microcode source
- Your commented masm sources (.c and .ll)
- Your masm Makefile
- The output of mic1 runs for each test program

# Extra Multiply/Divide Info

Multiplication - overflow

Overflow is when you add two positives and get a negative or when you add two negatives and get a positive. Put that together with knowing the number is positive if the most significant bit is 0, and it's negative if the most significant bit is 1. So look at the negative flag, expecting one result if the two numbers are positive and the opposite if they are negative.

Division and "unsigned" remainders

The remainder is specified as being an unsigned number to remove any ambiguity when the two numbers being operated on (the dividend and divisor) are of a different sign, meaning the quotient is negative.

If the dividend and divisor are both negative or both positive, the result is positive and there's no ambiguity about the remainder. For example, 10 divided by 4 is 2 with a remainder of 2, and you get that same result for -10 divided by -4.

But what if it's -10 divided by 4 (or 10 divided by -4)? Should the result be -2 with a remainder of -2 or should it be -3 with a remainder of 2? That's the ambiguity the assignment is resolving. The assignment is telling you that the remainder must be unsigned (i.e. non-negative) So the answer for this should be -3 with a remainder of 2.

(Double-checking the math, -10 = (4 * (-3)) + 2.)

## Original Microcode (~bill/cs305/promfile.txt)

```
70:a := ac;                              { 1111101 = SWAP }
71:ac := sp;
72:sp := a; goto 0;
73:alu := tir; if n then goto 76;
74:a := band(ir, smask);                 { 1111110 = INSP }
75:sp := sp + a; goto 0;
76:tir := tir + tir; if n then goto 80;
77:a := band(ir, smask);                 { 11111110 = DESP }
78:a := inv(a);
79:a := a + 1; goto 75;
80:rd; wr;                               { 11111111 = HALT }
```

## Extended Microcode

```
70:a := ac;                              { 1111101 = SWAP }
71:ac := sp;
72:sp := a; goto 0;
73:alu := tir; if n then goto 76;
74:a := band(ir, smask);                 { 1111110 = INSP }
75:sp := sp + a; goto 0;
76:tir := tir + tir; if n then goto 80;
77:a := band(ir, smask);                 { 11111110 = DESP }
78:a := inv(a);
79:a := a + 1; goto 75;
80:tir := tir + tir; if n then goto 97;{ 1111 1111 1x = HALT }
81:alu := tir + tir; if n then goto 89;{ 1111 1111 01 =RSHIFT}
82:mar := sp; a := sp + 1; rd;          { 1111 1111 00 = NAND }
83:rd;
84:mar := a; b := mbr; rd;
85:rd;
86:c := mbr;
87:a := band(b, c);
88:ac := inv(a); goto 0;
89:a := lshift(1);                       { 1111 1111 01 = RSHIFT}
90:a := lshift(a + 1);
91:a := lshift(a + 1);
92:a := a + 1;
93:b := band(ir, a);
94:b := b + (-1); if n then goto 96;
95:ac := rshift(ac); goto 94;
96:goto 0;
97:rd; wr;                               { 1111 1111 1x = HALT }
```

## Original Flex Scanner (~bill/cs305/MasmSrc/mic1symasm.ll)

```
#define LODD 1
#define STOD 2
#define ADDD 3
#define SUBD 4
#define JPOS 5
#define JZER 6
#define JUMP 7
#define LOCO 8
#define LODL 9
#define STOL 10
#define ADDL 11
#define SUBL 12
#define JNEG 13
#define JNZE 14
#define CALL 15
#define PSHI 16
#define POPI 17
#define PUSH 18
#define POP 19
#define RETN 20
#define SWAP 21
#define INSP 22
#define DESP 23
#define HALT 24
#define INTEG 25
#define JUNK 26
#define LABEL 27
#define LOC 28
#define STR 29

%%
[Ll][Oo][Dd][Dd]    return(LODD);

[Ss][Tt][Oo][Dd]    return(STOD);

[Aa][Dd][Dd][Dd]    return(ADDD);

[Ii][Nn][Ss][Pp]    return(INSP);

[Dd][Ee][Ss][Pp]    return(DESP);

[Hh][Aa][Ll][Tt]    return(HALT);
                :
                :
\".+\"              return(STR);

-?[0-9][0-9]*       return(INTEG);

[A-Za-z][0-9A-Za-z]*: return(LABEL);

\.LOC               return(LOC);

;.*\n ;

" "    |
"\t"   |
"\r"   |
"\n"   ;

[^ \t\r\n]* return(JUNK);
```

## Original MASM Assembler (~bill/cs305/MasmSrc/mic1symasm.c)

```
p1 = fopen("/tmp/passone", "w+");
unlink("/tmp/passone");
while(tok=yylex()){
switch(tok){
       case 1: switch(tok=yylex()){
          case INTEG:
                 str_12(yytext);
                 fprintf(p1,"%d 0000%s\n", pc, cstr_12);
                 break;
          case LABEL:
                 fprintf(p1,"%d U0000000000000000 %s\n", pc, yytext);
                 break;
          default:
                 fprintf(stderr,"Bad operand after LODD is %s on line
                 %d\n", yytext, pc);
                 exit(1);
            }

          break;
       case 2: switch(tok=yylex()){
          case INTEG:
                 str_12(yytext);
                 fprintf(p1,"%d 0001%s\n", pc, cstr_12);
                 break;
          case LABEL:
                 fprintf(p1,"%d U0001000000000000 %s\n", pc, yytext);
                 break;
          default:
                 fprintf(stderr,"Bad operand after STOD is %s on line
                 %d\n", yytext, pc);
                 exit(1);
          }
       break;
       .
       .
       .
       case 23: if((tok=yylex()) != INTEG){ /* DESP */
                    fprintf(stderr,"Bad operand after DESP is
                    %s\n",yytext);
                    exit(1);
                 }
                 str_8(yytext);
                 fprintf(p1,"%d 11111110%s\n", pc, cstr_8);
                 break;

        case 24: fprintf(p1,"%d 1111111100000000\n",pc); /* HALT */
                 break;

        case 25: str_16(yytext); /* INTEG */
                 fprintf(p1,"%d %s\n", pc, cstr_16);
                 break;

        case 27: if (label_pc == pc){ /* LABEL */
                   fprintf(p1,"%d U0000000000000000 %s\n", pc, yytext);
                   break;
                 }
                 search_sym_table(yytext);
                 update_sym_table(yytext);
                 label_pc = pc;
                 pc--;
                 break;
```

```
      case 28: if((tok=yylex()) != INTEG){ /* LOC */
                  fprintf(stderr,"Bad operand after .LOC is
               %s\n",yytext);
                  exit(1);
               }
               if((temp = ((unsigned short)atoi(yytext) )) < pc){
                 fprintf(stderr,"Bad operand after .LOC is %s, TOO
                 SMALL !\n",yytext);
                 exit(1);
               }
               pc = temp - 1;
               break;

   case 29: i=1; /* STR */
               do{
                  if(*(yytext+i) == '\"'){
                    bstr_16(0);
                    fprintf(p1,"%d %s\n", pc, binstr_16);
                    break;
                  }
                  temp = (unsigned short)*(yytext+i++);
                  if(*(yytext+i) != '\"'){
                  temp = (temp | ((unsigned short)*(yytext+i) << 8));
                  }
                  bstr_16(temp);
                  fprintf(p1,"%d %s\n", pc, binstr_16);
               }while(*(yytext+i++) != '\"' && ++pc);
               break;
                                                    /* JUNK */
      case 26:  fprintf(stderr,"Unrecognized token is %s\n",yytext);
                exit(26);

/* default */
      default:  fprintf(stderr,"Default case, unrecoverable error\n");
                exit(26);
}
```

## Extended Flex Scanner

```
#define LODD 1
#define STOD 2
#define ADDD 3
#define SUBD 4
#define JPOS 5
#define JZER 6
#define JUMP 7
#define LOCO 8
#define LODL 9
#define STOL 10
#define ADDL 11
#define SUBL 12
#define JNEG 13
#define JNZE 14
#define CALL 15
#define PSHI 16
#define POPI 17
#define PUSH 18
#define POP 19
#define RETN 20
#define SWAP 21
#define INSP 22
#define DESP 23
#define HALT 24
#define INTEG 25
#define JUNK 26
#define LABEL 27
#define LOC 28
#define STR 29
#define MULT 30
#define RSHIFT 31
#define DIV 32

%%

[Ll][Oo][Dd][Dd] return(LODD);
[Ss][Tt][Oo][Dd] return(STOD);
[Aa][Dd][Dd][Dd] return(ADDD);
:
:
[Hh][Aa][Ll][Tt] return(HALT);
[Mm][Uu][Ll][Tt] return(MULT);
[Rr][Ss][Hh][Ii][Ff][Tt] return(RSHIFT);
[Dd][Ii][Vv] return(DIV);

\".+\" return(STR);

-?[0-9][0-9]* return(INTEG);

[A-Za-z][0-9A-Za-z]*: return(LABEL);

\.LOC return(LOC);

;.*\n ;

" " |
"\t" |
"\r" |
"\n" ;

[^ \t\r\n]* return(JUNK);
```

## Extended MASM Assembler

```
case DESP: if((tok=yylex()) != INTEG){
              fprintf(stderr,"Bad operand after DESP is %s\n",yytext);
              exit(1);
          }
          str_8(yytext);
          fprintf(p1,"%d 11111110%s\n", pc, cstr_8);
          break;

case HALT: fprintf(p1,"%d 1111111111000000\n",pc);
          break;

case MULT: if((tok=yylex()) != INTEG){
              fprintf(stderr,"Bad operand after MULT is %s\n",yytext);
              exit(1);
           }
           str_6(yytext);
           fprintf(p1,"%d 1111111100%s\n", pc, cstr_6);
           break;

case RSHIFT: if((tok=yylex()) != INTEG){
           fprintf(stderr,"Bad operand after RSHIFT is %s\n",yytext);
           exit(1);
           }
           str_6(yytext);
           fprintf(p1,"%d 1111111101%s\n", pc, cstr_6);
           break;
case DIV: fprintf(p1,"%d 1111111110000000\n",pc);
           break;

case INTEG:  str_16(yytext);
            fprintf(p1,"%d %s\n", pc, cstr_16);
            break;

case LABEL: if (label_pc == pc){ /* for < lbx: lby: > */
              fprintf(p1,"%d U0000000000000000 %s\n", pc, yytext);
              break;
              }
              search_sym_table(yytext);
              update_sym_table(yytext);
              label_pc = pc;
              pc--;
              break;
```