SE201 - 2016 - PR1 Assignment

General Instructions

- You can download all required source files, that are provided as starting points for the following exercises, from the course website.
- Each group should submit a zip (or tar.gz) file containing all the necessary source code files via the course website.
- Each group should submit a report, written in French or English, in the PDF file format via the course website.
- Each student has to be member of a group, where groups should generally consist of 3 students.

1 Two's Complement Arithmetic

Aims: *Understand and manipulate the two's complement number representation and the associated arithmetic circuits.*

1.1 Unsigned Addition

For this exercise we will assume that binary numbers are represented as strings of characters `'0'` and `'1'`. Download the file `addition.c` from the course website and complete missing parts of the program (look for `TODO` in the code). You can use the provided logic functions `and`, `or`, and `xor` that are already defined in the source code.

- Implement the function `half_adder`, which should behave like a single-bit half adder as discussed in the lecture.

```
void half_adder(char a, char b, char *s, char *c)
{
  *c = and(a, b);
  *s = xor(a, b);
}
```

- Implement the function `full_adder`, which should use the `half_adder` function, in order to realize a single-bit full adder as discussed in the lecture.

```
void full_adder(char a, char b, char c_in, char *s, char *c)
{
  char tmp_s, tmp_c1, tmp_c2;
  half_adder(a, b, &tmp_s, &tmp_c1);
  half_adder(tmp_s, c_in, s, &tmp_c2);
  *c = or(tmp_c1, tmp_c2);
}
```

- Implement the function `addition`, using the `full_adder` function, in order to perform an addition of two `N`-bit numbers (where `N` is a constant defined in the code).

  ```
  void addition(char *a, char *b, char *s)
  {
    char c = '0';
    for(unsigned int i = 0; i < N; i++)
    {
      full_adder(a[N - 1 - i], b[N - 1 - i], c, &s[N - 1 - i], &c);
    }
  }
  ```

- Thoroughly comment your code, i.e., explain what the code is doing, define what the various functions expect as inputs, describe the functions' output, and mention error conditions.
- Thoroughly test your code using various inputs. Did you find *interesting* inputs? Describe them in the report and explain what happens.

Example: Calling the program with the command line arguments `"00111"` and `"00101"` should print the following output on the screen:

```
brandner@kairon:~> ./addition 00111 00101
addition:
a:   00111
b: + 00101
c:   01100
```

Hint: Test your code with varying bit-widths, i.e., test your code with varying values for `N`.

1.2 Signed Addition and Subtraction

For this exercise, we use the same string representation for binary numbers. However, using the functions implemented before, we will perform signed additions and subtractions.

- Add a function `addition_signed`, that uses the `addition` function, in order to perform additions on signed numbers. Perform an overflow check on the result you obtain. In case of an overflow, signal an error on `stderr` using `fprintf` and terminate the program using `exit(2)`.

  ```
  void addition_signed(char *a, char *b, char *s)
  {
    // do regular unsigned addition
    addition(a, b, s);

    // check for overflow -- sign-bit is at index 0
    if (a[0] == b[0] && b[0] != s[0])
  ```

```
    {
      fprintf(stderr, "%s:␣overflow␣detected:␣%s\n", program, s);
      exit(2);
    }
  }
```

- Add another function `subtraction_minus_one` that computes $A - B - 1$ using the `addition` function (you can ignore overflows).

  ```
  void subtraction_minus_one(char *a, char *b, char *s)
  {
    char c = '0';
    for(unsigned int i = 0; i < N; i++)
    {
      full_adder(a[N - 1 - i], not(b[N - 1 - i]), c, &s[N - 1 - i], &c);
    }
  }
  ```

- Thoroughly comment your code, i.e., explain what the code is doing, define what the various functions expect as inputs, describe the functions' output, and mention error conditions.
- Thoroughly test your code using various inputs. Again try to find *interesting* inputs and describe them in the report.

Example: Calling the program with the command line arguments `"11111"` and `"10000"` should signal an overflow:

```
brandner@kairon:~> ./addition 11111 10000
./addition: overflow detected: 01111
```

Calling it with the arguments `"11111"` and `"11111"` should print the following result:

```
brandner@kairon:~> ./addition 11111 11111
addition:
a:   11111
b: + 11111
c:   11110
signed addition:
a:   11111
b: + 11111
c:   11110
subtraction:
a:   11111
b: - 11111 - 1
c:   11111
```

1.3 Unsigned Multiplication

For this exercise we use the regular C data types `uint32_t` (a 32-bit unsigned number) and `uint64_t` (a 64-bit unsigned number) in order to implement a multi-cycle multiplier as a C function. Download the file `multiplication.c` from the course website and complete missing parts of the program (look for `TODO` in the code). You can use the normal C operators to implement the function (+, −, <<, >>). However, the multiplication operator (*) is not allowed.

- Implement the `multiplication` function following to the example of the multi-cycle multiplier discussed in the lecture.

  ```
  uint64_t multiplication(uint32_t a, uint32_t b)
  {
    uint64_t result = 0;
    for(unsigned int i  = 0; i < 32; i++)
    {
      if (b & 1 == 1)
      {
        result = result + a;
      }
      b = b >> 1;
      a = a << 1;
    }

    return result;
  }
  ```

- Improve the code of the `multiplication` function in order to minimize the number of iterations needed.
- Thoroughly comment your code, i.e., explain what the code is doing, define what the various functions expect as inputs, describe the functions' output, and mention error conditions.
- Thoroughly test your code using various inputs. Again try to find *interesting* inputs and describe them in the report.

Example: Calling the program with the command line arguments `5` and `3` should print the following result:

```
brandner@kairon:~> ./multiplication 5 3
5 * 3 = 15
```

Hint: For your first implementation of the multiplication, use a **`for`**-loop counting to 32.

1.4 Multiplication and Overflow

For this exercise no code is required, please include your answers to the questions from below in the report.

- Choose two 5-bit numbers and describe how the multi-cycle multiplier operates on these two numbers, e.g., indicate the values of the multiplicand, multiplier, and product registers for each cycle.
- Under which circumstances can overflow occur when the result is stored as a signed 5-bit values? How could you check for these circumstances?
- Can you also construct an example illustrating on overflow for a 5-bit multiplication result using unsigned numbers?

2 Processor Design

Aims: *Explain and understand the instruction set of a processor and its implementation using a simple pipeline.*

2.1 Instruction Set Architecture

Describe the instruction set and the binary representation of these instruction for a simple processor. Your processor should respect the following list of characteristics:

- All instructions should be encoded in 16 bits. Apart from the instruction width, you are free to define the binary format yourself.
- Your processor should have 8 registers, i.e., encoding a register operand requires 3-bits. Assume that each register is 32-bit wide.
- The `PC` of your processor should be 32-bit wide.
- Your processor has separate instruction and data memories.
- Define at least three different arithmetic/logic instructions operating on 3 register operands (reading 2 registers and writing 1).
- Define an instruction that copies a signed 11-bit immediate value into a register.
- Define an instruction to read a 32-bit value from data memory (load). The instruction should take two register operands (reading 1 and writing 1). The address used to access the memory is simply the value of the register read.
- Define an instruction to write a 32-bit value to data memory (store). The instruction should take two register operands (reading 2). The address used to access the memory is simply the value of one of the registers read.
- Define a conditional branch instruction with 2 register operands (reading 2) and a signed 8-bit immediate operand. The branch is taken when the two register operands are *not* equal. The new `PC` value is computed as follows: $\mathtt{PC}_{\mathtt{new}} = \mathtt{PC}_{\mathtt{old}} + \mathtt{imm}$. Untaken branches simply continue straight.

Using the instruction set you just defined, please complete the following exercises and include your replies in the report:

- Describe each instruction of your processor. Explain what the instruction is doing, how it can be written in human readable form (assembly), and how it is encoded in binary form.
- Group instructions into binary formats, similar to the I-, J-, and R-format discussed for MIPS in the lecture. Illustrate the formats using figures in your report.
- Provide a sequence of instructions in assembly form that allows to load the constant 2097152 (2^{21}) into a register using the instructions of your processor.

```
li r1 = 1
li r2 = 21
shl r1 = r1 << r2
```

- Translate the following C-code to corresponding instructions of your processor. Assume that variables a, s, and i are kept in processor registers.

```
  li r1 = 0x200 // a
  li r2 = 0     // s
  li r3 = 0     // i
  li r4 = 4
  li r5 = 1
  li r6 = 10
loop:
  ld r7 = [r1]        // *a
  add r2 = r2 + r7    // s = s + *a
  add r1 = r1 + r4    // a++
  add r3 = r3 + r5    // i++

  br r3 == r6: loop   // for i != 10
```

```
int *a = 0x200;
int s = 0;
for(int i = 0; i != 10; i++)
{
  s = s + *a;
  a++;
}
```

Hint: The pointer increment a++ should simply add 4 to the pointer's value stored in a register, i.e., we assume that variables of type **int** are 32-bit wide.

2.2 Pipelining

Now define the pipeline of your processor, while respecting the following characteristics:

- Your processor should have three pipeline stages: instruction fetch (IF), instruction decode (ID), and execute (EX).
- For arithmetic and branch instructions the three pipeline stages correspond, except for minor differences, to the pipeline stages of the MIPS processor discussed in the lecture.
- Memory accesses are, however, different. Since no address computation is needed (the address is simply the value of a register operand), the memory access can be executed in the EX stage.
- Taken branches flush the two instructions in the IF and ID stage.

- Assume that the processor registers are written at the beginning of the `EX` stage and read at the end of the `ID` stage, i.e., values written in the `EX` stage are immediately available in the `ID` stage.

Using the instruction set of your processor from the previous exercise and your pipeline design, please complete the following exercises and include your replies in the report:

- Draw a diagram of your processor's design. Use registers, pipeline registers, multiplexers, ALUs, . . . , as you need them. You can ignore the logic needed to flushing the `IF` and `ID` stages for conditional branches.
- Which kinds of hazards (data, control, or structural) can you encounter for your processor? Explain under which circumstances these hazards occur. How are these hazards resolved?
- Does your processor need *forwarding* (as discussed in the lecture)? Explain why it is needed or why not.
- Draw a pipeline diagram for the last two iterations of the assembly program corresponding to the **`for`**-loop of the previous exercise. Highlight hazards in the diagram and count the penalties.