CS2100 Computer Organization
Tutorial 5: Control Path
(Week 6: 2 Oct – 6 Oct 2023)

For this tutorial, we will keep it simple and (like the textbook) just assume a MIPS processor that can handle only R-type, **`lw`**, **`sw`**, and **`beq`** instructions.

1. So far you have learnt C as well as MIPS programming. You are also starting to see the hardware implementation aspect of things. We will soon be starting on the journey to see how hardware implementations are done but whatever hardware does, it can be *described* in software. First, we need some preliminaries. There is a header file in C, called stdint.h, introduced by the C99 standard, that allows programmer to specify the exact bitwidths of integers. Among other things, it introduces the data type **`int8_t`**, **`int_16_t`**, **`int32_t`**, **`int64_t`**, **`uint8_t`**, **`uint_16_t`**, **`uint32_t`**, and **`uint64_t`** for signed 8, 16, 32, and 64 bit integers and *unsigned* 8, 16, 32, and 64 bit integers, respectively. In hardware description languages (like Verilog and VHDL – but not C), arbitrarily lengths are also supported. For the purpose of this tutorial, let's assume there is a **`int<N>_t`** and **`uint<N>_t`** type in C that also supports arbitrary length **`N`** signed and unsigned integers. As with **`int8_t`** etc, we shall trust that the compiler will "do the right thing" such as selecting the right assembly instructions to accomplish the operations required. Note that (just as a convention) **`N`** must be greater than 1. If **`N`** = 1, we will use **`bool`** (Boolean) data type instead.

   Going back to what is described earlier, we can use software (specifically, C in our case) to describe hardware and its function. So let's start with some of the components of the CPU. We can described instruction memory as an array:

```
uint32_t instruction_memory[1073741824];
```

   Yeah, the array size is huge but we are only doing a description here. Dealing with this large range in reality requires the help of hardware and the operating system, which you will gradually discover in your journey through SoC. Also, for convenience, even though we know that memory is an array of bytes, we will deal it with 32-bit at a time for simplicity since we don't need to deal with **`lb`** etc.

   Now:

   a) Write a C data structure and a function to describe the register file and its operation.

Answer:

```
int32_t _rf[32];

void RegFile(uint5_t RR1,
             uint5_t RR2,
             uint5_t WR,
             int32_t WD,
             int32_t *RD1,
             int32_t *RD2,
             bool RegWrite) {
```

```
                    // Because we need to send out multiple
               outputs,
                    // we will use passing by pointers.
                    *RD1 = (RR1 > 0)?_rf[RR1]:0;
                    *RD2 = (RR2 > 0)?_rf[RR2]:0;

                    if (RegWrite && WR)
                      _rf[WR] = WD;
               }
```

b) Write a C data structure and a function to describe the data memory and its operation.

Answer:

```
int32_t data_memory[1073741824];

int32_t ReadDataMemory(uint32_t address,
                       int32_t WrData,
                       bool MemRead,
                       bool MemWrite)
{
     // Nothing for Memory to do.
     If (!(MemRead || MemWrite) return 0;

     // We can do a sanity check here.
     if (MemRead && MemWrite) {
        // raise an error
     }

     if (MemRead) {
        return data_memory[address];
     }

     if (MemWrite) {
        data_memory[address] = WrData;
        return 0;
     }
 }
```

2. Next, let's deal with multiplexing (labelled as "MUX" in the slides). Ideally, we want to use some kind of template feature to describe it but C does not support templates (of course C++ does, but that's another story.) So show how two-way multiplexors ("MUX") of different bitwidths can be instantiated using the macro expansion facility of C. In other words, each call to the macro should yield a C function that implements a two-way multiplexing function (i.e., "if selection is 0, output is input0, and if selection is 1, output is input1) where the inputs and output are of bitwidth **N**.

Answer:

You will need to use C macro processor's *token pasting* facility, i.e. **##**. See below how it works with respect to the parameter N to the macro MUX.

```
#define MUX(N) \
        int##N##_t mux##N (int##N##_t in0,    \
                           int##N##_t in1,    \
                           bool ctrl)         \
        {                                     \
            if (ctrl) return in1;             \
            else return in0;                  \
        }
```

Note the use of "****". Macro expansion is strictly not part of the C language. It is more string processing, and has a different syntax and semantics to C itself. The "****" at the end of the line is to say "continuing on the next line". Note also that the last line does not require a "****".

Now, if in the code, there is this line:

```
MUX(8);
```

It will expand to:

```
int8_t mux(bool ctrl, int8_t in0,
                      int8_t in1)

{
    if (ctrl) return in1;
    else return in0;
}
```

Because "**MUX(8)**" is textually expanded to this function definition, we should not put a ";" If you do, the expanded code will end up with "};" at the last line. The C compiler is ok with it. So it is up to you – are you happier with the statement looking "weird" before or after expansion, though both do not affect correctness? Macro expansion gives us a new degree of freedom in writing our programs by treating our code as text, do some text processing, before sending it to the compiler proper.

And by the way, there is an alternative: you can use C's conditional operator:

```
#define MUX(N) \
        int##N##_t mux##N(int##N##_t in0, \
                          int##N##_t in1, \
                          bool ctrl)      \
        {                                 \
            return (ctrl?in1:in0);        \
        }
```

C's conditional operator works like if-then-else but is an expression - it returns a result. So it can be used to form compound expressions. See: https://www.geeksforgeeks.org/conditional-or-ternary-operator-in-c/.

3. Using C's **`struct`** and **`union`** data types, define a data type that will hold an instruction (let's call it "**`struct insn`**") of R-, I-, and J-type along with its sub-fields.

Answer:

```
struct insn {
   uint6_t  opcode;
   uint5_t  rs, rt, rd, shamt;
   uint6_t  funct;
   uint16_t immed;
   uint26_t target;
};
```

4. Main control is to be implemented by this function (we use pointers to pass multiple values out):

```
void Control(uint6_t opcode,
             bool    *_RegDst,
             bool    *_Branch,
             bool    *_MemRead,
             bool    *_MemtoReg,
             uint2_t *_ALUOp,
             bool    *_MemWrite,
             bool    *_ALUSrc,
             bool    *_RegWrite);
```

Provide the C code for computing the following signals that would be in this function:

a) **_RegDst**

Answer:

RegDst determines whether **Rd** or **Rt** is to be used as the destination register. An easy criteria for doing this is to observer that for all R-type instructions, **Rd** is the destination register (indicated by an output of 1) while all others may only involve **Rt**. "But what if the instruction doesn't write to **Rt**?" No problem. That is handled by the **RegWrite** signal. If **RegWrite** is correct, **RegDst** will be safely ignored for those instructions that do not write to registers.

```
if (!opcode) *_RegDst = 1;
else *_RegDst = 0;
```

b) **_ALUSrc**

Answer: (Colin pointed out the bug.)

ALUSrc determines if the ALU is to receive the second output of reading the register file or the immediate of the instruction as the second operand. **rt** should be the second ALU operand for R-type instructions (opcode = 0) as well as **beq** (opcode = 4).

```
if (!opcode || opcode == 4) *_ALUSrc = 0;
else *_ALUSrc = 1;
```

c) **_MemRead**

Answer:

MemRead is easy - just return 1 if it is any of the load instructions.

```
if (opcode == 0x23)
   *_MemRead = 1;
else *_MemRead = 0;
```

d) **_ALUOp**

Answer:

R-type instructions have a diverse range of operations required of the ALU. What exactly to do will have to be decided by the funct code. However, other non-R-type instructions may require the ALU to do work too. And these will not have funct codes. These will have to be decoded by main control and the ALU instructed accordingly. Note that we are only handling (some) R-type, **lw**, **sw**, and **beq** in this exercise. Otherwise, 2 bits for ALUOp may not be enough. For example, I-type instructions requiring more complicated ALU operations will require more bits.

```
switch (opcode) {
    case 0:
         *_ALUOp = 2;
         break;
    case 0x23:      // lw
    case 0x2b:      // sw
         *_ALUOp = 0;
         break;
    case 0x4:       // beq
         *_ALUOp = 1;
         break;
}
```

5. Write a C function that will model ALUControl:

```
uint4_t ALUControl(uint2_t _ALUOp,
                   uint6_t _funct);
```

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

Answer:

In our 2-level design, **ALUControl** will decode the **funct** code of R-type instructions to control the ALU. However, for non-R-type instructions, the

decoding of the ALU operation is done by main control and passed along as the **ALUOp**. But even for R-type, the small table above is not enough to implement all the **funct** codes in the MIPS instruction set. Again, we are keeping it simple for pedagogical reasons.

```
uint4_t ALUControl(uint2_t _ALUOp, uint6_t _funct)
{
     if (_ALUOp == 2) { // R-type instruction.
                        // Need to decode funct code.
        switch (_funct) {
           case 0x20:        // add
           case 0x21:        // addu - see Note 1.
                return 2;    // ALU to perform Add
           case 0x22:        // sub
           case 0x23:        // subu - see Note 1.
                return 6;    // ALU to perform Sub
           case 0x24:
                return 0;    // ALU to perform And
           case 0x25:
                return 1;    // ALU to perform Or
           case 0x27:        // nor
                return 0xC; // Nor
           case 0x2a:
                return 7;    // slt - see Note 2.

           default:
                // raise an error
        }
     } else { // Non R-type
       if (_ALUOp == 0)
          return 2; // Ask ALU to add

       if (_ALUOp == 1)
          return 6; // Ask ALU to sub

     }
}
```

Note 1: Signed numbers are represented in two's complement. Unsigned numbers are taken to be a binary value "as is". In two's complement arithmetic (as presented in class), subtraction is achieved using addition and addition is just basic binary addition – with the additional check for overflow. In true processors, there would be a overflow bit to indicate this (just like we have a isZero bit). Unsigned addition is just basic binary addition without overflow checking. Subtraction is a bit more complex but we can assume that it is the same. Since our ALU does not output any indication of overflow, the two are therefore equivalent.

Note 2: "**slt**" may seem to be like "subtract two numbers and check if it is less than zero." Unfortunately, the checking is not the same for signed and unsigned due to the sign bit. Take for example, two 4-bit binary numbers $a = 1111_2$ and $b = 0001_2$. Interpreted as signed numbers in two's complement, a represents -1, and hence a signed comparison would say that a is less than b. However, as unsigned numbers, a represents 15, and 15 is greater than 1, hence a is greater than b. Since

our table of ALU operations does not include an unsigned "**slt**", it is left undecoded.

6. Write a C function that will model the behavior of the ALU having the following function prototype:

```
int32_t ALU(int32_t in0, int32_t in1,
            uint4_t ALUControl, bool *ALUiszero);
```

where **in0** and **in1** are the 32-bit inputs, **ALUControl** is the 4-bit ALU control signals, and the outputs are the **ALUiszero** bit (passed by pointer) and the 32-bit result.

Answer:

Mostly straightforward with a few points to take note of listed after the code below:

```
int32_t ALU(int32_t in0, int32_t in1,
            uint4_t ALUControl, bool *ALUiszero)
{
    int32_t result;

    switch (ALUControl) {
       case 0:
          result = in0 & in1;
          break;
       case 1:
          result = in0 | in1;
          break;
       case 2:
          result = in0 + in1;
          break;
       case 6:
          result = in0 - in1;
          break;
       case 7:
          result = (int32_t)(in0 < in1);
          break;
       case 12:
          result = ~(in0 | in1);
          break;
    }

    *ALUiszero = (result == 0);
    return(result);
}
```

Note 1: Whether the second operand is from register or immediate is resolved prior to the ALU.

Note 2: We have to distinguish between the logical AND and OR, and the bitwise AND and OR.

Note 2: a NOR b is defined as NOT(a OR b). However, we cannot use the logical NOT operator (i.e., "!") because we want it to be bitwise. Hence, we need to use C's bitwise complement operator (i.e., "~"). You will study these things in detail in the next part of the course.