CS2100 Computer Organization

Tutorial #5: Control

30 September – 4 October 2024

1. We will be using what was learnt in the datapath and control lectures for this tutorial. You are given the following 3 MIPS instructions (same as in the datapath tutorial):

   (a) `beq $1, $3, 12`

   (b) `lw $24, 0($15)`

   (c) `sub $25, $20, $5`

   For each of these instruction, fill in the corresponding elements of the tables given below. The first table contains the various datapath elements while second contains the control elements. Use the notation \$5 to represent register number 5, [\$5] to represent the content of register number 5 and Mem(X) to represent the memory data at address X. Also indicate the value of PC after the instruction is executed.

Registers File				ALU		Data Memory	
RR1	RR2	WR	WD	Opr1	Opr2	Address	Write Data

RegDst	RegWrite	ALUSrc	MemRead	MemWrite	MemToReg	Branch	ALUop	ALUcontrol

2. Given below are the resource latencies of the various hardware components (ps = picoseconds = 10^{-12} seconds)

Inst-Mem	Adder	MUX	ALU	Reg-File	Data-Mem	Control / ALU-control	Left-shift/ Sign-Extend/ AND
400ps	100ps	30ps	120ps	200ps	350ps	100ps	20ps

   Give the estimated latencies for the following MIPS instructions:

   (a) `SUB` instruction (e.g. `sub $25, $20, $5`)

   (b) `LW` instruction (e.g. `lw $24, 0($15)`)

   What do you think the cycle time should be for this particular processor implementation?

   Hint: First, you need to find out the critical path of an instruction, i.e. the path that takes the longest time to complete. Note that there could be several parallel paths that work more or less simultaneously.

3. We can use software (specifically, C in our case) to describe hardware and its functions. First, we need some preliminaries. There is a header file in C, called `stdint.h`, introduced by the *C99* standard, that allows programmers to specify the exact bit widths of integers. Among other things, it introduces the data types `int8_t, int16_t, int32_t, int64_t, uint8_t, uint16_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 for the operations required. Note that (just as a convention) `N` > 1. If `N` = 1, we will use the `bool` (Boolean) data type instead. Let us start with some of the components of the CPU. As an example, we can describe the instruction memory as an array:

   `uint32_t instruction_memory[1073741824];`

Yes, 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 with 32-bits 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.

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

4. We will now look at 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 (C++ does, but that's another story!). So show how two-way multiplexors (*MUX*) of different bit widths 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 bit width `N`.

5. 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`

(b) `ALUSrc`

(c) `MemRead`

(d) `ALUop`

6. Write a C program that will model `ALUcontrol`

```
uint4_t ALUcontrol(uint2_t _ALUop, uint6_t _funct);
```

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

7. 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 signal, and the outputs are the `ALUiszero` bit (passed by pointer) and the 32-bit result.

8. **For Fun** ☺: This question is all about some extra 'fun' learning ☺. You have heard about the *Big-Endian* and the *Little-Endian* formats for memory storage. Do you know where the terminology comes from?