CS2100 Computer Organization

Tutorial #4: Datapath – Draft Answer

16 – 20 September 2024

1. An ISA has 16-bit instructions and 5-bit addresses. There are two classes of instructions: class A instructions have one address, while class B instructions have two addresses. Both classes exist and the encoding space for the opcode is completely utilized. Please answer the questions below.

   (a) What is the minimum total number of instructions?

   (b) What is the maximum total number of instructions?

   (Past year's exam question)

   Answer:

   Class A instructions are in the format: PPPPPPPPPPP AAAAA. Here, PPPPPPPPPPP represents the 11-bits for the opcode and AAAAA the 5 bits for the address.

   Class B instructions are in the format: JJJJJJ AAAAA AAAAA. Here, JJJJJJ represents the 6-bits for the opcode and AAAAA the 5 bits for the address.

   (a) To get the minimum total number of instructions, we should give the least number of addresses to the instruction format that has the highest number of bits for the opcode. So, we give the bit pattern 111111 (6-bits) to A and the rest of the $2^6 - 1$ bit-patterns to B.
   Hence, class A has: 1 x $2^5 = 32$ opcodes, while class B has 63.
   Total = 32 + 63 = 95 opcodes.

   (b) To get the maximum total number of instructions, we should give the highest number of addresses to the instruction format that has the highest number of bits for the opcode. So, we give the bit-pattern 000000 (6-bits) to B and the rest of the $2^6 - 1$ bit-patterns to A.
   Hence, class A has: $2^6 - 1 * 2^5 = 63 * 32 = 2016$ opcodes, while class B has 1.
   Total = 2016 + 1 = 2017 opcodes.

2. You have seen how the **blt** ("branch less than") instruction can be implemented in the lecture slides. As we know, MIPS assembly also allows for *pseudo-instructions* which the assembler will expand to multiple instructions to implement the functionality. Show how the following pseudo-instructions are implemented using real MIPS instructions:

   (a) `bgt $r1, $r2, L` ("branch greater than")

   (b) `bge $r1, $r2, L` ("branch greater than or equal")

   (c) `ble $r1, $r2, L` ("branch less than or equal")

   (d) `li $r, imm` ("load immediate" where the immediate can be any length up to 32-bits))

   (e) `nop` ("no operation", i.e. a null operation)

   Answer:

   (a) `bgt $r1, $r2, L` ≡ `blt $r2, $r1, L`
   or:

   ```
   slt $at, $r2, $r1
   bne $at, $zero, L
   ```

   And this is why we need to reserve a register \$at (\$1) for "assembly temporary". Using any other register will work also – provided you are careful that the register used doesn't contain a value assigned before this pair of instruction and then used after it – because this pair of instruction will overwrite that register.

(b) From the lecture notes: if ($r1 ≥ $r2) ⇒ if (!($r1 < $r2))
or:

```
slt $at, $r1, $r2
beq $at, $zero, L
```

The clever bit here is that while `bne` is used to test if a condition is TRUE, we can use `beq` to test if a condition is FALSE, because `slt $at, $r2, $r1` will set $at to 1 if $r2 < $r1 but 0 otherwise, and `bne` will take the branch if $at is 1, i.e., the `slt` comparison was true, while `beq` will take the branch if the `slt` comparison turns out to be false.

(c) Similar to the previous answer.
if ($r1 ≤ $r2) ⇒ if ($r2 ≥ $r1) ⇒ if (!($r2 < $r1))
or:

```
slt $at, $r2, $r1
beq $at, $zero, L
```

(d) Since it is a constant, the assembly can determine at assembly time whether it will need more than 16 bits to store it. If the constant is less than 16 bits, then it can use:

```
ori $r, $zero, imm
```

If it is longer, it will need the `lui-ori` pair given in the lecture. There is something else worth pointing out here. If you read the MIPS instruction sheet specification carefully, you will find that the immediate arguments for the arithmetic instructions are "SignExtImm", i.e., sign extended immediates, while those for the logical instructions like ori are "ZeroExtImm", i.e., zero extended immediates. This gives different results. The assembler will parse the immediate in "li" (in its human written form) and then decide accordingly.

(e) "Why would you want to execute an instruction that does nothing at all? Isn't it a waste?" Good question. It turns out that at times it is useful or even necessary to fill in (or "zero out") part of the instruction memory or instruction stream. We will see this later on in the module. All instruction sets would actually cater to this in some ways. If you look at the MIPS instruction set encoding, you will discover that 32 bits of zeroes encodes the following instruction:

```
sll $zero, $zero, 0
```

While there are other instructions which also does nothing at all, this `nop` is particularly convenient because the encoding turns out to be 32 bits of zeroes, which corresponds to the notion of "zeroing out" in the data section.

3. Convert the following MIPS instructions to its corresponding hexadecimal equivalent.

(a) `beq $1, $3, 12`
(b) `lw $24, 0($15)`
(c) `sub $25, $20, $5`

Answer:

(a) `beq` is an I-type instruction, so we have the format as:

31 ... 26	25 ... 21	20 ... 16	15 ... 0
opcode	rs	rt	immediate

Looking at the encoding and the MIPS reference data sheet, we get:
`beq` has opcode 0x4 ⇒ 000100
`$1` is `$rs` ⇒ 00001
`$3` is `$rt` ⇒ 00011
`12` ⇒ 00000000 00001100
The 32-bit value is: 0001 0000 0010 0011 0000 0000 0000 1100
Value in Hexadecimal is: 0x1023000C

(b) `lw` is an I-type instruction

Looking at the encoding and the MIPS reference data sheet, we get:

`lw` has opcode 0x23 ⇒ 100011

`$15` is `$rs` ⇒ 01111

`$24` is `$rt` ⇒ 11000

`0` ⇒ 00000000 00000000

The 32-bit value is: 1000 1101 1111 1000 0000 0000 0000 0000

Value in Hexadecimal is: 0x8DF80000

(c) `sub` is a R-type instruction, so we have the format as:

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

Looking at the encoding and the MIPS reference data sheet, we get:

`sub` has opcode 0x0 ⇒ 000000

`$20` is `$rs` ⇒ 10100

`$5` is `$rt` ⇒ 00101

`$25` is `$rd` ⇒ 11001

shmat is 0 ⇒ 00000

funct is 0x22 ⇒ 100010

The 32-bit value is: 0000 0010 1000 0101 1100 1000 0010 0010

Value in Hexadecimal is: 0x0285C822

4. Draw the MIPS datapath for the following instruction. Make sure that you specify the necessary bits on any lines you draw.

   `addi $15, $14, -50`

   Answer: You can use the slides in the lectures as a good starting point for this. The key is to make sure that you do have a sign-extend element as the numbers are only 16-bit and the ALU takes in 32-bit values. One solution to this is provided in Figure 4. Only the elements necessary to this datapath are shown here.

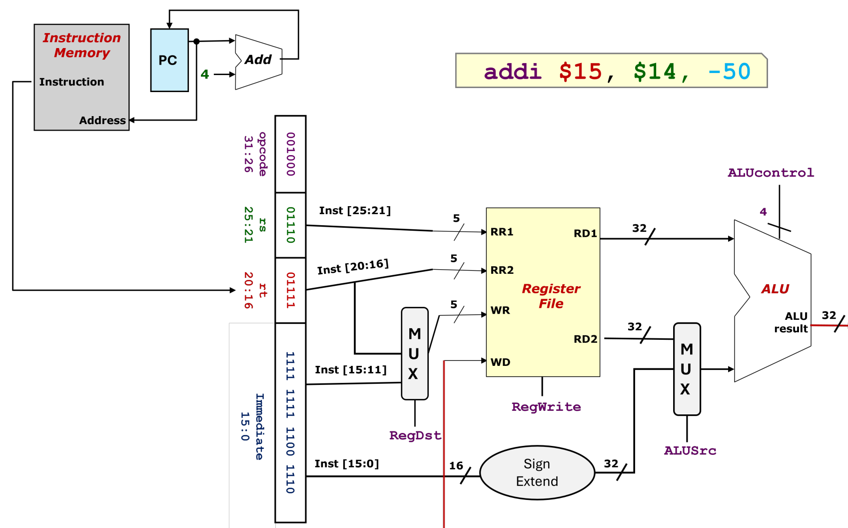

Figure 1: Datapath for question 4

5. **For Fun** ☺: This question is all about some extra 'fun' learning, if you choose to ☺. We have heard a lot about GPUs (Graphics Processing Unit) in the news off late, especially in relation to the advancements in Machine Learning and AI. Given you are learning how to build a simple CPU, how are GPUs different to CPUs from an architecture perspective?

Answer: The surprising answer to this is that not by much. A lot of what you learn is exactly what goes into a GPU as well ☺. Both CPUs and GPUs work on the same fetch-decode-execute cycle. While CPUs are typically designed to handle many complex and diverse tasks, GPUs are optimized to handle simpler workloads, but those that can be run in parallel. Figure 5 shows a good high level difference between a CPU and a GPU architecture. In the figure, you can see that while a CPU has a more H/W per core, the GPU has more cores that share some of this H/W. This typically leads to what is called as "Single Instruction Multiple Threads' execution, where you have a single datapath for the fetch and decode steps that is shared by many cores. This allows the same instruction to execute on many cores, but on different data, thus leading to a very high degree of parallelism.

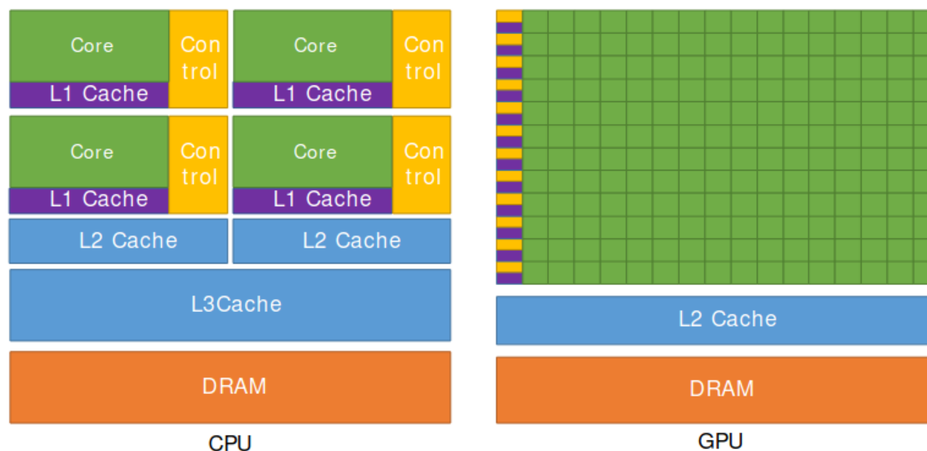

Figure 2: High level difference between CPU and GPU architecture (source: nVidia CUDA C++ Programming Guide. Available at: `https://docs.nvidia.com/cuda/archive/11.2.0/pdf/CUDA_C_Programming_Guide.pdf`)

Still want to know more? Here are some articles that go into these differences in much more detail:

`https://blogs.nvidia.com/blog/whats-the-difference-between-a-cpu-and-a-gpu/`
`https://www.intel.com/content/www/us/en/products/docs/processors/cpu-vs-gpu.html`
`https://www.run.ai/guides/multi-gpu/cpu-vs-gpu`
`https://aws.amazon.com/compare/the-difference-between-gpus-cpus/`

Happy reading ☺.