G2
Pipelined Execution

Annie, Oleks, ARK15 Course Team, see also Major Contributors

Version 1.1, September 28, 2015

Table of Contents

This is the second in a series of three G-assignments ("G" for "Godkendelse" and/or "Gruppeopgave") which you must pass in order to be eligible for the exam in the course Machine Architecture (ARK) at DIKU. We encourage pair programming, so please form groups of 2-3 students.

Sections 4.1—4.4 of [COD] set the stage for this assignment, and we will explore the sections 4.5—4.8 in excruciating detail. We assume that you've at least skimmed all of §§ 4.1—4.8, and are prepared to flip back and forth, mining for the details. You are also *strongly encouraged* to have read through Appendix B.1-B.3, B.5, B.7, and B.8.

Furthermore, this assignment assumes that you've already solved most of G1.

If you have any comments or corrections to the text, visit our public GitHub repository at https://github.com/onlineta/ark15.

Happy hacking :-)

1. Assignment

This is a short overview of your assignment. Flip back to this if you are ever in doubt about what you are doing.

Your task is to rewrite your simulator from G1 to be a *pipelined* simulator. This simulator **must** implement the instructions that you didn't implement in G1. We repeat: You **will not pass G2**, if you do not implement the instructions that didn't work in your G1 simulator.

Additionally, your new simulator should support as many of the following instructions as possible: `add`, `addu`, `addi`, `addiu`, `and`, `andi`, `beq`, `bne`, `j`, `jal`, `jr`, `lw`, `lui`, `nor`, `or`, `ori`, `sll`, `slt`, `sltu`, `slti`, `sltiu`, `srl`, `sub`, `subu`, `sw`, and halt when it sees a `syscall` instruction. If you start running out of time, prioritize these instructions: `add`, `addiu`, `beq`, `j`, `jal`, `jr`, `lw`, `lui`, `ori`, `sll`, `slt`, `srl`, `sub`, `sw`, and `syscall`.

The interface of the simulator should remain the same, but in addition to printing the number of instructions executed, it should *also* print the number of clock cycles elapsed.

```
Executed %zu instruction(s).
%zu cycle(s) elapsed.
pc = 0x%x
at = 0x%x
v0 = 0x%x
v1 = 0x%x
t0 = 0x%x
t1 = 0x%x
t2 = 0x%x
t3 = 0x%x
t4 = 0x%x
t5 = 0x%x
t6 = 0x%x
t7 = 0x%x
sp = 0x%x
ra = 0x%x
```

NB! Please follow this format precisely as your code will be subject to automated testing.

2. Introduction

So far, we have only counted the number of instructions that our simulator executes. Unwittingly, what we implemented in G1, was a *single-cycle* simulator: each instruction took one "clock cycle" to complete.

Equally important to CPU performance is the wall-clock duration of a clock cycle.

In an *edge-triggered clocking methodology*, all state changes occur at the edge of a clock cycle. In a single clock cycle, we can read a register, send the value through some *combinational logic*, and write a register. All writes to a *state element* (e.g. a register) which do not happen on a clock edge will have no effect. If all this sounds like black magic to you, see Section 4.2 (pages 248-251) and Appendix B.7 (pages B-48-50) in [COD5e].

Some MIPS32 instructions require more logic than others. Consider the `lw` instruction:

1. Read the instruction from instruction memory.
2. Decode base register, destination register, and immediate field; read the base register.
3. Add the value of the immediate field to the value of the base register.
4. Load the resulting address from data memory.
5. Store the loaded value in the destination register.

These stages are commonly given the following names:

1. **Instruction Fetch (IF)** — fetching an instruction from instruction memory.
2. **Instruction Decode and Register File Read (ID)** — (speaks for itself).
3. **Execution and Address Calculation (EX)** — using the ALU.

4. **Memory Access (MEM)** — accessing the data memory.
5. **Write Back (WB)** — writing results back to the register file.

Not all instructions need to go through all these stages. Most MIPS32 instructions omit the memory access stage altogether. But we cannot perform these instructions any faster if we have to stretch the duration of a clock cycle to accommodate instructions that utilize all five stages (e.g. `lw`): **In a single-cycle architecture, we cannot make the common case fast**.

One solution is to add intermediate state elements between the stages, and to advance all stages simultaneously in a single clock cycle. This technique is called "pipelining".

In a single-cycle architecture, all stages execute in sequence, and no stage executions overlap. The duration of a clock cycle is stretched to accommodate the execution of all 5 stages in a clock cycle:

```
+-----------------------------+-----------------------------+-----------------------------+
|            800ps            |            800ps            |            800ps            |
+-----------------------------+-----------------------------+-----------------------------+
| IF  | ID  | EX  | MEM | WB  |
                              | IF  | ID  | EX  | MEM | WB  |
                                                            | IF  | ID  | EX  | MEM | WB  |
```

In a pipelined architecture, all stages execute simultaneously in every clock cycle (provided they all have something to do). The duration of a clock cycle can then be reduced, as there is less combinational logic to accommodate in one clock cycle:

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|200ps|200ps|200ps|200ps|200ps|200ps|200ps|200ps|200ps|200ps|200ps|200ps|200ps|200ps|200ps|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| IF  | ID  | EX  | MEM | WB  |
      | IF  | ID  | EX  | MEM | WB  |
            | IF  | ID  | EX  | MEM | WB  |
                  | IF  | ID  | EX  | MEM | WB  |
                        | IF  | ID  | EX  | MEM | WB  |
                              | IF  | ID  | EX  | MEM | WB  |
                                    | IF  | ID  | EX  | MEM | WB  |
                                          | IF  | ID  | EX  | MEM | WB  |
                                                | IF  | ID  | EX  | MEM | WB  |
                                                      | IF  | ID  | EX  | MEM | WB  |
                                                            | IF  | ID  | EX  | MEM | WB  |
```

Pipelining retains instruction *latency*: although it now takes up to 5 clock cycles to execute an instruction, the execution time of an instruction remains the same due to **a shorter clock cycle duration**.

Pipelining increases instruction *throughput*: pipeline start-up overhead aside, the number of clock cycles is roughly equal to the number of instructions. With a shorter clock cycle duration, more instructions get executed in the same wall-clock time-frame. In the example above, the single-cycle architecture only made it through 3 instructions in 2400ps, while the pipelined architecture made it through 11 instructions.

This exploitation of *parallelism* in a sequential instruction stream creates many opportunities for *hazards* to occur, as subsequent instructions may depend on the results of preceding instructions,

which have not finished executing yet. Forwarding data and stalling the pipeline are just some of the ways such hazards are resolved.

3. Rewriting Your Simulator

Firstly, we need to conceptually split the execution of an instruction into the execution of the 5 pipeline stages. Each stage advances an instruction to the next pipeline stage (or stalls the pipeline).

MODELLING CONCEPT

One way to simulate an instruction pipeline is to have a function for every pipeline stage, and to call the stage functions in order *from end to start* of the pipeline. For instance, we could name these functions `interp_wb`, `interp_mem`, `interp_ex`, `interp_id`, and lastly, `interp_if`. The execution of these five functions (in that order), constitutes a clock cycle.

Mental exercise: Why shouldn't we execute the stage functions in order from start to end?

With this modelling concept, "advancing an instruction" to the next pipeline stage involves passing on everything necessary to execute the immediately following, *and* any subsequent pipeline stages for the instruction. Data is passed via the 5 so-called *pipeline registers*:

1. **IF/ID**: Data from the IF stage to the ID stage (+EX+MEM+WB).
2. **ID/EX**: Data from the ID stage to the EX stage (+MEM+WB).
3. **EX/MEM**: Data from the EX stage to the MEM stage (+WB).
4. **MEM/WB**: Data from the MEM stage to the WB stage.

Mental exercise: Why don't we also have a WB/IF pipeline register?

With these pipeline registers, the old registers (which we called `regs`) will from now on be referred to as *programmer-visible registers*.

MODELLING CONCEPT

A C-struct is a collection of named fields. So is a pipeline register.

We can model the pipeline registers using static C-structs which we'll call `if_id`, `id_ex`, `ex_mem`, and `mem_wb`.

Each stage function then reads from its respective pipeline register, and writes to its subsequent pipeline register. For instance, `interp_id` reads from `if_id` and writes to `id_ex`. As with `mem`, `regs`, and `PC`, let's keep the pipeline registers static, declared at the top of our `sim.c`.

3.1. Getting Started

We assume that you have correctly solved most of G1.

Recursively copy your solution for the first assignment to get started on the second:

```
~$ cd ark
~/ark$ mkdir 2nd
~/ark$ cp -r 1st/* 2nd/
```

Download the handout archive from Absalon and place it in the `~/ark` folder. Unpack the archive, to add/overwrite the new or updated handout files:

```
~/ark$ tar xvf g2-handout-v1.0.tar.gz
```

Your old assembly files are likely to *not* work with the pipelined simulator, until you are completely done with the assignment.

EXERCISE

Break your simulator:

1. Declare a variable `cycles` alongside your `instr_cnt`.
2. Define a non-zero macro `SAW_SYSCALL` at the top of your file.
3. Write a function stub, `cycle` above your `interp`. `cycle` should return an `int` indicating how the cycle went. For now, let it just return the non-zero value `SAW_SYSCALL`.
4. Replace the loop body in your `interp` function with a call to `cycle`. Make sure to break out of the loop if `cycle` returns a non-zero value (as with `interp_inst` in G1). If `cycle` returned `SAW_SYSCALL`, `interp` should return successfully.
5. Count up the new variable `cycles` instead of `instr_cnt` in your `interp` loop. We will count up `instr_cnt` elsewhere.

CORRECTING EXERCISE

There was a mistake in the G1 assignment text. The original text said that `SP` should be initialized to poin to the 4th last byte in `mem`. This is not correct.

MIPS convention has it, that `SP` denotes the most recently used (data) memory address. None of the data memory is initially in use, so `SP` should initially be set to the 1st byte past `mem` (the stack grows downwards).

Correct this in your implementation.

TESTING EXERCISE

For any valid configuration and ELF file, your (broken) simulator should exit with the value 0. Use `echo $?` to print the exit code of the last command executed in your terminal.

3.2. Troubleshooting

Before we get too far off with our pipeline, we would like to take the time to give you some advice on troubleshooting your implementation. We strongly encourage you to skip this section until you e.g. hit a so-called "segmentation fault", or get tangled up in all the different "control bits".

Perhaps the first thing you should do is read the Tips about control bits.

In general, we recommend that you try to test your implementation in a stage-by-stage manner. Print the values of the different pipeline registers as instructions progress through the pipeline. Check that things are set (and unset!) properly as you progress. You might also find the function `getchar()` (defined in `<stdio.h>`) useful in your `cycle` or `interp` to "pause" the simulator until you hit e.g. enter.

This is what we might call printf-style debugging. If you are looking for something faster, GDB, The GNU Project Debugger, might what you're looking for. If nothing else, it is very useful for catching segmentation faults.

To use GDB with your implementation, you will need to add an additional compilation flag to your `Makefile`. You need to tell GCC to compile for debugging with GDB. To do this, specify the `-g` option when you compile your `sim.c`:

Makefile

```
sim: mips32.h elf.o sim.c
  $(CC) $(CFLAGS) -g -o sim elf.o sim.c
```

3.2.1. Catching Segmentation Faults

Segmentation faults are caused by memory writes to, or reads from invalid memory addresses. This typically indicates trouble with `lw`, `sw`, branching, or jumping instructions, or your forwarding implementation (if you got that far).

Before you start, check your assembly program. Check that you are not using something too far off the stack pointer for your `lw` or `sw` instructions, if you have any.

Start `gdb` by specifying your (compiled for GDB) executable:

```
~/ark/2nd$ gdb ./sim
GNU gdb (GDB) ...
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
...
Reading symbols from ./sim...done.
(gdb)
```

This is the GDB prompt where you can enter GDB commands. One GDB command you can enter is to run the program with some chosen set of arguments:

```
(gdb) run default.cfg asm/sw-lw.elf
Starting program: /home/archimedes/ark/2nd/sim default.cfg asm/sw-lw.elf

Program received signal SIGSEGV, Segmentation fault.
0x000000000040145f in interp_mem () at sim.c:126
126     SET_BIGWORD(mem, ex_mem.alu_res, ex_mem.rt_value);
(gdb)
```

GDB is telling us a lot more than the raw command-line did! The segmentation fault happens on line 126, which (in this case) is part of `interp_mem`.

Your program has not finished running. For GDB, a segmentation fault is like a breakpoint. You can ask GDB for the value of different local or global variables at this point. For instance, what is the value of `ex_mem.alu_res`, in hexadecimal notation?

```
(gdb) print/x ex_mem.alu_res
$1 = 0xfffffffc
(gdb)
```

Or how does it look in binary notation?

```
(gdb) print/t ex_mem.alu_res
$2 = 11111111111111111111111111111100
(gdb)
```

You can even ask GDB to print out whole structs:

```
(gdb) print if_id
$3 = {inst = 0, next_pc = 4194340}
(gdb) print ex_mem
$4 = {mem_read = false, mem_write = true, reg_write = false,
      mem_to_reg = false, branch = false, bzero = false,
      rt = 0 '\000', rt_value = 3, reg_dst = 0 '\000',
      alu_res = 4294967292, branch_target = 4194316}
(gdb)
```

So it looks like what is wrong with our program is that `ex_mem.alu_res` is not computed correctly, but where does this really go wrong? You could now go ahead with printf-style debugging, knowing what to look for, or you could continue with GDB-style debugging.

3.2.2. Debugging with GDB

(Start up GDB again to walk your way to the segmentation fault.)

To set a breakpoint with GDB, use the GDB command `break` (before you run your program).

You can break on entry to a function in your C file:

```
(gdb) break cycle
Breakpoint 1 at 0x4023fc: file sim.c, line 560.
```

Or break when a line in your C file is hit:

```
(gdb) break 319
Breakpoint 2 at 0x4019e4: file sim.c, line 319.
```

After you have set your breakpoints, run the program:

```
(gdb) run default.cfg asm/sw-lw.elf
Starting program: /home/archimedes/2nd/sim default.cfg asm/sw-lw.elf

Breakpoint 1, cycle () at sim.c:560
560   int retval = 0;
(gdb)
```

After you've examined the values you want to examine using `print`, you can instruct GDB to continue until the next breakpoint is met:

```
(gdb) continue
Continuing.

Breakpoint 2, interp_if () at sim.c:319
319   if_id.inst = GET_BIGWORD(mem, PC);
(gdb)
```

In this case, it looks like the `interp_if` breakpoint is rightfully reached before the `cycle` breakpoint is reached again (`interp_if` in this C file was on lines 318-323).

You can also step though the program one C line at a time from here:

```
(gdb) next
320   PC += 4;
(gdb) print/x PC
$1 = 0x400018
(gdb)
```

To exit GDB, use the `quit` command.

If you are looking for more GDB commands, we recommend this GDB cheat sheet.

3.2.3. GDB scripts

It can get a little tedious to set breakpoints and run your program every time you compile your program anew. You can use a GDB script to get this work done for you.

A GDB script is a file that contains a list of GDB commands. For instance, something like this `gdb.script` file might be useful:

gdb.script

```
break cycle
run default.cfg asm/sw-lw.elf
continue
print/x if_id
```

This script sets a breakpoint at the function `cycle`, runs the simulator, continues the first time the breakpoint is hit (the very first cycle), and on next hit of the breakpoint, prints the `if_id` register, in hex.

To run GDB with this script, use the `-x` option:

```
$ gdb -x gdb.script ./sim
...
Reading symbols from ./sim...done.
Breakpoint 1 at 0x4023fc: file sim.c, line 560.

Breakpoint 1, cycle () at sim.c:560
560   int retval = 0;

Breakpoint 1, cycle () at sim.c:560
560   int retval = 0;
$1 = {inst = 0xafa8fffc, next_pc = 0x40001c}
(gdb)
```

3.2.4. Tips about control bits

1. Check that you `break` out of all your cases in `interp_control`.
2. Remember to set `mem_to_reg` *every time* you set `reg_write` to `true`. Otherwise, `lw` can creep in on your R-type instructions, and vice-versa.
3. Remember to set the `branch` control bit to `false` for all instructions other than `beq`, `bne`. Otherwise, you might branch off to odd places.
4. Remember to set the `jump` control bit to `false` for all instructions other than `j`, `jal`, `jr`. Otherwise, you might jump off to odd places.
5. Check the `alu_src` for all instructions that pass through the `alu`.
6. Remember to set the `mem_read` and `mem_write` control bits for *all* instructions. This will prove useful in G3.

Go back to Troubleshooting if you are still having trouble.

3.3. Implementing `lw` and `sw`

We will start by implementing the `lw` and `sw` instructions. We have already discussed how `lw` does something in every pipeline stage. `sw` is similar, except that it does nothing in the WB stage. Implementing `lw` and `sw` will get us started across the board, with something to do in every pipeline stage.

Despite the fact that we will call the stage functions in order *from end to start* of the pipeline, it is certainly most convenient to implement the functions in order *from start to end*:

1. Instruction Fetch (IF)
2. Instruction Decode and Register File Read (ID)
3. Execution and Address Calculation (EX)
4. Memory Access (MEM)
5. Write Back (WB)

3.3.1. Instruction Fetch (IF)

Instruction Fetch, or IF, is the first pipeline stage. In the IF stage, we read the instruction addressed by the PC from memory, and increment the PC. We save the instruction that was read in the IF/ID register.

EXERCISE

1. Declare a static C struct, `if_id`, near the top of `sim.c` (just below your existing static variable declarations):

```
struct preg_if_id {
  uint32_t inst;
  // ...
};
static struct preg_if_id if_id;
```

2. Write a function `interp_if()`:
   a. Use the macro `GET_BIGWORD` (defined in `mips32.h`) to get the instruction addressed by `PC` from `mem`.
   b. Save the instruction in `if_id.inst`.
   c. Increment `PC` by 4.
   d. Count up `instr_cnt`.
   e. Call `interp_if` from `cycle`.

Note that `interp_if` cannot fail, and so should return `void`.

3.3.2. Instruction Decode and Register File Read (ID)

Instruction Decode and Register File Read, or ID, is the second stage of the pipeline. In the ID stage, we decompose the instruction into its constituent fields, set up the control signals for subsequent pipeline stages, and read in the necessary registers, among other things.

As with the IF stage, we start out with a simple implementation, focusing for now on just the `lw` and `sw` instructions. Both are I-type instructions, so we are interested in the `opcode`, `rs`, `rt`, and `imm` fields of the instruction passed in the IF/ID pipeline register. For the subsequent pipeline stages we will need:

1. The value of the register addressed by the `rs` field.
2. The value of the sign-extended `imm` field.
3. To signal to the EX stage that it should calculate the sum of the above values.

Furthermore, if it is a `lw` instruction, we will need:

4. To signal to the MEM stage that it should read the memory address computed in the EX stage.

5. To signal to the WB stage that it should store the value loaded in the MEM stage in register `rt`.

If it is a `sw` instruction, we will need:

4. The value of the register addressed by the `rt` field.
5. To signal to the MEM stage that it should write the value to the memory address computed in the EX stage.
6. To signal to the WB stage that it should do nothing.

We will use "control bits" to signal to subsequent pipeline stages what they should and should not do.

MODELLING CONCEPT

A "bit" is a value that is either asserted or deasserted. Bits are basic units in hardware, but hard to deal with in software. It is easier to model "control bits" using booleans, i.e. values which are either `true` or `false`. Underneath the covers, a boolean usually takes up one byte of memory.

You will need to include `<stdbool.h>` at the top of your `sim.c` to get the standard C boolean type `bool`.

HACK

To signal to the EX stage what it should do, [COD5e] goes to great lengths to define a so-called "ALU control unit" (see pages 259-261, as well as Figure 4.18 on page 266).

We will take a shortcut and use a `funct` field in our ID/EX pipeline register. This field will be needed later for R-type instructions. Since this field otherwise goes unused for the instructions `lw`, `sw`, `beq`, and `bne` instructions, we can use it to "spoof" an ALU control unit.

For `lw` and `sw` instructions, the `funct` field should be set to `FUNCT_ADD` (defined in `mips32.h`).

EXERCISE

1. Declare a static C struct, `id_ex`, just below the declaration of `if_id`:

```
struct preg_id_ex {
  bool mem_read;
  bool mem_write;
  bool reg_write;
  // ...
};
static struct preg_id_ex id_ex;
```

2. The struct already has the following control bits defined:

Table 1. Control bits in the ID/EX pipeline register

Control bit	Destination Stage	Intent
`mem_read`	MEM	Whether we should read from memory
`mem_write`	MEM	Whether we should write to memory
`reg_write`	WB	Whether we should write back to a register

3. Add the following fields to the struct:

Table 2. Other fields in the ID/EX pipeline register

Field	Meaning	Where from?
`rt`	Value of the `rt` field	Use `GET_RT` on `if_id.inst`
`rs_value`	Value of the `rs` register	Lookup `rs` in `regs` array
`rt_value`	Value of the `rt` register	Lookup `rt` in `regs` array
`sign_ext_imm`	Sign extended immediate	`SIGN_EXTEND` and `GET_IMM` on `if_id.inst`
`funct`	ALU operation to perform in EX	Set by `interp_control` (below)

EXERCISE

Define a function `interp_control()`:

1. Switch on the `opcode` field of the instruction in the IF/ID register.
2. In the case of `OPCODE_LW` or `OPCODE_SW`, set `mem_read`, `reg_write`, and `mem_write` appropriately.
3. For either case above, set `id_ex.funct` to `FUNCT_ADD`. (See also HACK above.)
4. For the default case, return a suitable error value.

EXERCISE

Write a function `interp_id()` just below `interp_control`:

1. Call `interp_control` at the bottom of your `interp_id` to set the control bits and funct field of the ID/EX register. Make sure to check the return value of `interp_control`.
2. Set the other fields of the ID/EX register before calling `interp_control`.
3. Call `interp_id` *before* calling `interp_if` in `cycle`.

Note that `interp_id` *can* fail, and so should return `int`. Make sure to check this return value in `cycle`.

It is also important to note, that `interp_id` will be called before any calls to `interp_if`. What will the instruction in the IF/ID register look like the first time `interp_id` runs? Since the IF/ID pipeline register is *statically allocated*, the initial value of all the fields in the IF/ID register, including the instruction, will be set to 0.

This zero instruction has a name: `nop`, or "no operation". We will need to support this instruction before we can do any testing. (At present, `interp_control` simply fails!) To implement `nop`, set all control bits to `false` in `interp_control`.

EXERCISE

Add a check at the top of `interp_control` if `if_id.inst` is 0.

If it is, set all control bits to `false` and return successfully from the function.

Mental exercise: Why is this both *necessary* and *sufficient* to implement `nop` instructions?

The last thing we need to do before we can test our progress, is to change `cycle` to not *always* return `SAW_SYSCALL`.

EXERCISE

Update the return value of `cycle`. If none of the pipeline stages failed, return 0.

TESTING EXERCISE

Check that `interp_if` and `interp_id` work as intended with the provided `sw-lw.S`. Note, you don't support R-type instructions *yet*. `sw-lw.S` has a `syscall` instruction. For now, the intended behaviour is that your simulator fails and complains about an unknown opcode.

3.3.3. Execution and Address Calculation (EX)

Execution and Address Calculation, or EX, is the third pipeline stage. In the EX stage, the ALU performs its operation. Unlike the suggestion in [COD5e], we didn't construct an ALU control unit. Instead, we spoofed the `funct` field to be `FUNCT_ADD` for the `lw` and `sw` instructions.

EXERCISE

1. Declare a static C struct, `ex_mem`, just below the declaration of `id_ex`.
2. Add the control bits `mem_read`, `mem_write`, and `reg_write` to `ex_mem`.
3. Add also the following fields to the struct:

Table 3. Other fields in the EX/MEM pipeline register

Field	Meaning	Where from?
`rt`	Value of the `rt` field	The ID/EX register
`rt_value`	Value of the `rt` register	The ID/EX register
`alu_res`	Result of ALU operation	Set by `alu` (below)

The control bits above, as well as the `rt` and `rt_value` fields are not modified during the EX stage: They are first needed in the MEM and WB stages.

EXERCISE

1. Define a function `alu()` returning an `int`. `alu` will be *very* similar to `interp_r` in G1. As with `interp_r`, `alu` can fail if the `id_ex.funct` field has some unknown value.
2. Switch on `id_ex.funct`:
   a. Add support for `FUNCT_ADD`, adding `id_ex.sign_ext_imm` to `id_ex.rs_value`, and storing the result of the calculation in `ex_mem.alu_res`.
   b. In the default case, return a suitable error value.

HACK

Add support for funct value 0 in `alu`: simply break out of the switch-case. The funct value of a `nop` instruction is 0. The funct value of an `sll` instruction is also 0, but `sll` is an R-type instruction, which we will implement later.

EXERCISE

1. Define a function `interp_ex()` below `alu`:
   a. "Forward" the control bits and the `rt` and `rt_value` fields from `id_ex` to `ex_mem`.
   b. Call the `alu` function, so it can perform the ALU operation and set the `alu_res` field in the EX/MEM register.
   c. Call `interp_ex` *before* calling `interp_id` in `cycle`.

`alu` *can* fail, and if it does, pass on the error value as the return value of `interp_ex`. Make sure to check the return value in `cycle`.

TESTING EXERCISE

Check that `interp_if`, `interp_id`, and `interp_ex` work as intended with the provided `sw-lw.S`.

3.3.4. Memory Access (MEM)

Memory Access, or MEM, is the fourth stage of the pipeline.

In the MEM stage, we actually get to read from, and write to memory.

EXERCISE

1. Declare a static C struct, `mem_wb`, just below the declaration of `ex_mem`.
2. Add a `reg_write` control bit, and an `rt` field to the struct.
3. Add a field `read_data` to the struct, where we will store the data read for a `lw` instruction.

EXERCISE

Define a function `interp_mem()`:

1. Forward the `reg_write` control bit and the `rt` field through the MEM stage.
2. If `ex_mem.mem_read` is set, use the macro `GET_BIGWORD` to read from `mem` the word addressed by `ex_mem.alu_res`. Store the result in `read_data`.
3. If `ex_mem.mem_write` is set, use the macro `SET_BIGWORD` to write the value in `ex_mem.rt_value` to `mem` at the address stored in `ex_mem.alu_res`.
4. Call `interp_mem` *before* calling `interp_ex` in `cycle`.

`interp_mem` cannot fail, and so should return `void`.

TESTING EXERCISE

Check that `interp_if`, `interp_id`, `interp_ex`, and `interp_mem` work as intended with the provided `sw-lw.S`.

3.3.5. Write Back (WB)

Write Back, or WB, is the fifth and final stage of the pipeline.

In the WB stage, results from the previous stages are finally written back to the register file.

If you have followed along in the optional exercises, and taken a look at the handed out `sw-lw.S`, you might have noticed that each `sw` and `lw` instruction was followed by 4 `nop` instructions. This was done to allow the `sw` and `lw` instructions to reach the WB stage before an instruction that depends on their result enter the ID stage. We will fix this later by optionally forwarding data to the earlier stages in the pipeline, but first, let's finish the WB stage.

EXERCISE

Define a function `interp_wb()`:

1. If `mem_wb.reg_write` is set, store the value in `mem_wb.read_data` in the register addressed by `mem_wb.rt`, unless `mem_wb.rt` is register 0.
2. Call `interp_wb` *before* calling `interp_mem` in `cycle`.

`interp_wb` cannot fail, and so should return `void`.

TESTING EXERCISE

Verify that your interpreter works as intended with the provided `sw-lw.S`. Provided that register `t0` has the initial value x, you should see the value x in register `t1` if everything works as intended.

If you are having trouble, test your implementation stage-by-stage.

3.4. Implementing R-type Instructions

R-type instructions always write to a register, but never use the memory.

EXERCISE

Add a case for `OPCODE_R` in `interp_control`:

1. Set `mem_read`, `mem_write`, and `reg_write` appropriately.
2. Use `GET_FUNCT` on `if_id.inst` to set the `funct` field in the ID/EX register.

`lw` and `sw` were I-type instructions. This means that the ALU took its arguments from the `rs` register and the `imm` field of the instruction. For an R-type instruction, the ALU should take the `rs` and `rt` registers as its arguments.

`id_ex` already contains `rs_value`, `rt_value`, and `sign_ext_imm`. All we need to do is signal to the EX stage whether the ALU should use `id_ex.rt_value` or `id_ex.sign_ext_imm` as its second operand. We will use a control bit, `alu_src`, to signal this.

EXERCISE

1. Add the control bit `alu_src` to `id_ex`.
2. In `interp_control`, set `alu_src` to `false` for an R-type instruction, and `true` for a `lw` or `sw` instruction.
3. In `alu`, use `alu_src` to choose a suitable second operand for the ALU operation.

For both `lw` and R-type instructions, we want to write back to a register in the WB stage. In the case of `lw`, we want to write the value read in the MEM stage. In case of an R-type instruction, we want to write the ALU result obtained in the EX stage. We will use a control bit, `mem_to_reg`, to signal this.

For a `lw` instruction, the write back destination register is addressed by the `rt` field. For an R-type instruction, it is addressed by the `rd` field. We will use a pipeline register field called `reg_dst` to send the destination register address to the WB stage.

EXERCISE

1. Add the field `alu_res` to `mem_wb`, and forward `alu_res` through the MEM stage.
2. Add the control bit `mem_to_reg` to `id_ex`, `ex_mem`, and `mem_wb`. Forward `mem_to_reg` the same way you forwarded `reg_write`.
3. In `interp_control`, set `mem_to_reg` to `false` for R-type and `sw` instructions, and `true` for `lw` instructions.
4. Add the field `reg_dst` to `id_ex`, `ex_mem`, and `mem_wb`. This will hold the destination register. Forward `reg_dst` the same way you forwarded `reg_write`.
5. In `interp_control`, use `GET_RD` to set the `reg_dst` field for an R-type instruction, and `GET_RT` for a `lw` instruction.
6. Remove your existing implementation of the WB stage in `interp_wb` and start anew: If `reg_write` is not set, or `reg_dst` is zero, exit the function without doing anything. Otherwise, if `mem_to_reg` is set, write the value in `read_data` to the destination register (designated by `reg_dst`). If `mem_to_reg` is not set, write the value in `alu_res` to the destination register.

EXERCISE

Add support for `FUNCT_SYSCALL` in `alu`.

`alu` should return `SAW_SYSCALL` when it sees a `syscall` instruction.

TESTING EXERCISE

Check that your interpreter works as intended with the provided `add.S`. Provided that register `t0` has the initial value x, you should see the value x in register `t1` if everything works as intended.

If you are having trouble, check your implementation stage-by-stage.

Check that your interpreter *still* works as intended with `sw-lw.S`.

EXERCISE

Based on your implementation of `interp_r` in G1, add support in `alu` for the following instructions: `addu`, `and`, `nor`, `or`, `sll`, `slt`, `sltu`, `srl`, `sub` and `subu`. (We will handle `jr` later.)

To implement `sll` and `srl`, you will need to add a `shamt` field to the ID/EX pipeline register and read it off of the instruction in the ID stage using the `GET_SHAMT` macro.

Remember to remove the 0-case for `nop` in `alu`. This will now be handled by the `FUNCT_SLL` case. The `sll` instruction has funct value 0. The `nop` case now also requires no special handling in `interp_control`.

3.5. Implementing `beq` and `bne`

`beq` and `bne` are I-type instructions that neither use the memory, nor write to registers. We will focus on explaining `beq`, leaving `bne` as an exercise.

EXERCISE

1. Add a case for `OPCODE_BEQ` in `interp_control`.
2. Set `mem_read`, `mem_write`, and `reg_write` appropriately.

We want to branch on `beq` if the two operand registers are equal. R[rs] and R[rt] are equal if R[rs]-R[rt]==0. We can use the ALU to subtract R[rs] from R[rt]. This means that `beq` behaves a bit like an R-type instruction:

EXERCISE

In `interp_control`, set `alu_src` for `OPCODE_BEQ` the same way as you would do with an R-type instruction.

As with `lw` and `sw`, we can "spoof" an ALU opcode via the `funct` field in `id_ex`:

HACK

Set the `funct` field to `FUNCT_SUB` for `OPCODE_BEQ` in `interp_control`.

To perform a branch instruction, we need to compute a branch target address. The branch target address is relative to the address of the instruction immediately following the branch instruction. [COD5e] suggests that the branch target address should be computed in the EX stage (see e.g. the upper half of the EX stage in Figure 4.33 on page 287), and so we need to forward the incremented PC through the ID stage:

EXERCISE

1. Add a field `next_pc` to `if_id`.
2. Set `if_id.next_pc` to the incremented `PC` in `interp_if`.
3. Add a field `next_pc` to `id_ex`, and forward it through the ID stage.

Compute the branch target address in the EX stage:

EXERCISE

1. Add a field `branch_target` to `ex_mem`.
2. Set the branch target address in `interp_ex` using `next_pc` and `sign_ext_imm`. Remember to bit-shift `sign_ext_imm`!

Similarly, [COD5e] suggests that branching should be detected in the MEM stage (see e.g. the MEM stage in Figure 4.33 on page 287). We need to signal the MEM stage in case we see a `beq` instruction:

EXERCISE

1. Add a control bit `branch` to `id_ex`.
2. Set the `branch` control bit appropriately for all instructions in `interp_control`.
3. Add a control bit `branch` to `ex_mem`, and forward it through the EX stage.

Although [COD5e] suggests that branching should be implemented in the MEM stage, implementing it in `interp_mem` can get messy with our choice of executing pipeline stages in order from end to start of the pipeline. We can take a more "executive" approach, and check if we need to perform branching at the end of `clock`, once all the stage functions have executed. This is "the end of a clock cycle".

EXERCISE

At the end of your `cycle` function (after the call to `interp_if`), check if the `ex_mem.branch` control bit is set and also `ex_mem.alu_res` is 0. If they are, set `PC` to `ex_mem.branch_target`.

By always reading the next couple of instructions after a `beq`, we've implemented an **assume branch not taken** branch-prediction strategy. This means that if we *do* have to branch, some of the pipeline stages will have to drop what they were doing. This is called *flushing the pipeline*.

At the same time, modern MIPS processors implement a branch delay slot: The instruction immediately following a branch instruction is always executed. Modern MIPS processors detect and perform branching in the ID stage, avoiding flushing altogether.

We detect whether we need to branch after the EX stage, meaning that the IF/ID and ID/EX pipeline

registers need to be flushed. However, when implementing a branch delay slot, we *only* need to flush the IF/ID register.

Flushing a pipeline stage means turning its operation into a `nop`.

EXERCISE

If we detect that we need to branch at the end of `cycle`, set `if_id.inst` to 0 to flush the IF/ID pipeline register, in addition to updating `PC`. Remember to decrement the `instr_cnt`!

OPTIONAL EXERCISE

Avoid having to flush the IF/ID register. (You will need to change the entire approach to `beq`.)

TESTING EXERCISE

Check that your implementation of `beq` works as intended with the provided `beq-true-nopsled.S` and `beq-false-nopsled.S`.

Check that your interpreter *still* works as intended with `sw-lw.S` and `add.S`.

EXERCISE

Add support for `bne`.

3.6. Implementing `j`, `jal`, and `jr`

[COD5e] does not discuss a pipelined implementation of `j`, `jal`, and `jr`. This is left as an exercise for the reader. You can however, find a discussion of the single-cycle implementation on p. 270. In particular, see Figure 4.24 on p. 271.

Jumps also use a branch delay slot. Unlike branches however, we do not need to wait around until the end of the EX stage to detect if we should jump or not. Jumps can take place right after the end of the ID stage, once the jump instruction is decoded.

EXERCISE

1. Add a control bit `jump` to `id_ex`.
2. Add a field `jump_target` to `id_ex`.

We can use `id_ex.jump` and `id_ex.jump_target` to implement `j`, `jal`, and `jr`.

3.6.1. Implementing `j`

`j` and `jal` are J-type instructions.

`j` neither uses the memory, nor writes to a register, nor branches. (It jumps!).

EXERCISE

1. Add a case for `OPCODE_J` in `interp_control`:
   a. Set `mem_read`, `mem_write`, `reg_write`, `branch`, and `jump` control bits appropriately.
   b. Use `GET_ADDR` to get the address field of `if_id.inst`.
   c. Use the address field and `if_id.next_pc` to set the `id_ex.jump_target` field. Recall that jumps use pseudodirect addressing. See also G1.
2. Set the `jump` control bit appropriately for all other opcodes in `interp_control`.
3. At the end of `cycle` (after branch detection) check if the `id_ex.jump` control bit is set. If it is, set `PC` to `id_ex.jump_target`.

Mental exercise: Why should we perform jump detection *after* branch detection?

Since we jump after the end of the ID stage and do not flush the IF/ID pipeline register, our jumps implement a branch delay slot.

TESTING EXERCISE

Check that your implementation of `j` works as intended with the provided `j-nopsled.S`.

Check that your interpreter *still* works as intended with `sw-lw.S`, `add.S`, `beq-true.S`, and `beq-false.S`.

3.6.2. Implementing `jal`

Similarly to `j`, `jal` neither uses the memory, nor branches (it jumps!), but it does write to a register: the `ra` register.

We can implement this by internally turning the `jal` instruction into an R-type `add` instruction, with the operand values `0` and the value of the `PC` register.

EXERCISE

1. Add a case for `OPCODE_JAL` in `interp_control`. Do as you did with `OPCODE_J`, but:
   a. Set `reg_write` to `true`.
   b. Set `funct` to `FUNCT_ADD`.
   c. Overwrite `rs_value` and `rt_value` appropriately. (See also point 2.)
   d. Set `reg_dst` to 31, corresponding to the RA register. (Be careful not to use the `RA` macro!)
2. Make sure that you still call `interp_control` at the *bottom* of `interp_id`, to ensure that `rs_value` and `rt_value` are *overwritten* by the above case.

Mental exercise: What should we set `rs_value` and `rt_value` to?

TESTING EXERCISE

Check that your implementation of `jal` works as intended with the provided `jal-nopsled.S`, i.e. check the RA register printed by `show_status`.

Check that your interpreter *still* works as intended with `sw-lw.S`, `add.S`, `beq-true-nopsled.S`, `beq-false-nopsled.S`, and `j-nopsled.S`.

3.6.3. Implementing `jr`

EXERCISE

In the case for `OPCODE_R` in `interp_control`, check if the funct value of the instruction is `FUNCT_JR`. If it is, do as you did with `OPCODE_J`, but set `id_ex.jump_target` to the value of the `rs` register.

In `alu`, handle the case for `FUNCT_JR`. (Do nothing. You just handled `FUNCT_JR` in the ID stage.)

TESTING EXERCISE

Check that your implementation `jr` works as intended with the provided `jal-j-jr-nopsled.S`. You will need working implementations of `jal` and `j`.

Check that your interpreter *still* works as intended with `sw-lw.S`, `add.S`, `beq-true-nopsled.S`, `beq-false-nopsled.S`, `j-nopsled.S`, `jal-nopsled.S`.

3.7. Optional: Forwarding/Hazards

This part is optional, but if you have time to spare, you are encouraged to complete it: it is likely to be part of G3.

It is time to drop the "nop sled". We needed all those `nop` instructions to make sure that the instructions we were testing made it far enough through the pipeline, but this wastes clock cycles.

The values computed in the EX stage, or loaded from memory in the MEM stage, can be forwarded to earlier instructions already in the pipeline, if those instructions need them. See also § 4.7 on pp. 303–316 in [COD5e].

Similar to branching and jumping, it makes sense to implement forwarding at the end of `cycle`, once all the pipeline registers have been set.

EXERCISE

Define a function stub `forward()`, and call it at the end of `cycle`.

3.7.1. EX hazards

The first data hazard occurs when the data we need for the EX stage in the following clock cycle has only just passed the EX stage. This is called an *EX hazard*. See also the pseudo-code on p. 308 in [COD5e].

EXERCISE

1. Add a field `rs` to `id_ex` and set it appropriately in `interp_id`.
2. In `forward`, if `ex_mem.reg_write` is set, and the EX/MEM destination register is equal to either `id_ex.rs` or `id_ex.rt`, then forward the ALU result to the appropriate field(s) in ID/EX.

You may need to forward both to `id_ex.rs_value` and `id_ex.rt_value`. You should not forward if the EX/MEM destination register is register 0.

TESTING EXERCISE

Check that your implementation works as intended with the provided `ex-hazard.S`.

Check that your interpreter *still* works as intended with all the `*-nopsled.S` files.

3.7.2. MEM hazards

The second hazard is that the data we need for the EX stage in the following clock cycle has only just passed the MEM stage. This is called a *MEM hazard*.

Here we have to be careful that an EX hazard is not occurring at the same time as a MEM hazard. In this case, the EX hazard has precedence. See also the pseudo-code on p. 311 in [COD5e].

EXERCISE

Handle MEM hazards in `forward`:

1. If `mem_wb.reg_write` is not set, there is no MEM hazard.
2. If the MEM/WB destination register is equal to `id_ex.rs`, and no EX hazard is competing to forward a value to `id_ex.rs_value`, then forward either the ALU result or the data read from memory to `id_ex.rs_value`. (Check the `mem_wb.mem_to_reg` flag.) If this sounds confusing, see also the pseudo-code on p. 311 in [COD5e].
3. You will need to handle another case in addition to those discussed in [COD5e]. A `jal` instruction might be on it's way to the WB stage, when we hit a `jr` instruction in the ID stage. In this case, the `mem_wb.alu_res` should be forwarded to `id_ex.jump_target` instead of `id_ex.rs_value`.
4. Similarly, for `id_ex.rt`.
5. Make sure to still handle EX hazards as before.

You may need to forward both to `id_ex.rs_value` and `id_ex.rt_value`. You should not forward if the MEM/WB destination register is register 0.

TESTING EXERCISE

Check that your implementation works as intended with the provided `mem-hazard.S` and `mem_to_reg-hazard.S`.

Check that your interpreter *still* works as intended with all the `*-nopsled.S` files, as well as `ex-hazard.S`.

3.7.3. Load-use hazards

The last hazard is something we cannot fix with forwarding. A data hazard occurs when the result of a load instruction is needed in the EX stage. The MEM stage has to get a chance to execute, and so we have to *stall* the pipeline, and insert a `nop` in the EX stage. See also the pseudo-code on p. 314 in [COD5e].

Here, we cannot take an executive approach. To stall the pipeline we need to insert a `nop` without modifying the program counter, so we do not "lose" instructions.

EXERCISE

In `interp_if`, right after you load the instruction from memory, if `id_ex.mem_read` is set, check if the `rs` or `rt` field of the loaded instruction is equal to `id_ex.rt` (the destination register for a `lw` instruction). If it is, you should stall the pipeline and return from `interp_if` immediately (i.e. without updating `PC` or `instr_cnt`).

TESTING EXERCISE

Check that your implementation works as intended with the provided `lw-use-hazard.S`.

Check that your interpreter *still* works as intended with all the `*-nopsled.S` files, as well as `ex-hazard.S`, `mem-hazard.S`, and `mem_to_reg-hazard.S`.

TESTING EXERCISE

Check that your implementation works as intended with the provided `sw-lw.S`, `add.S`, `beq-true.S`, `beq-false.S`, `j.S`, `jal.S`, `jal-j-jr.S`.

Check that your interpreter *still* works as intended with all the `*-nopsled.S` files.

Mental exercise: Why do we still need 1 `nop` instruction before a `syscall` instruction?

Mental exercise: Why do we always need 2 `nop` instructions after a `syscall` instruction?

3.8. Finalé

EXERCISE

Add support for the remaining I-type instructions. That is, `addi`, `addiu`, `andi`, `lui`, `ori`, `slti`, and `sltiu`. If you run out of time, we can make due with `addiu`, `lui`, and `ori`.

4. Submitting Your Solution

Follow these steps to submit your solution.

4.1. Finalize Your Solution

Clean up your code, remove superfluous code, and add comments for the non-trivial parts.

Write a **short** report (`g2-report.txt` or `g2-report.pdf`) documenting your solution. Discuss what works, what doesn't, if anything. Discuss the design decisions you have had to make, if any. To back your claims, test with the handed out test programs, and add your own. Discuss your tests in your report.

Your report should be sufficient to get a good idea of the extent and quality of your implementation. **Your code will only be used to verify the claims you make in your report**.

4.2. Package Your Code

Use the `tar` command-line utility to package your code:

```
~/ark$ tar cvzf g2-code.tar.gz 2nd
```

4.3. Submit on Absalon

Submit **two files** on Absalon:

1. Your report (`g2-report.txt` or `g2-report.pdf`)
2. Your archive (`g2-code.tar.gz`)

Remember to **mark your team members** on Absalon.

5. Optional: Interpreting C

Do as in G1, but add 3 `nop` instructions after `syscall` in `_start.S`:

~/ark/2nd/c/_start.S

```
.globl _start
_start:
  jal main
  syscall
  nop # nop the ID stage
  nop # nop the IF stage (never reached, due to inverse pipeline order)
```

Mental exercise: Why don't we need a `nop` before the `syscall` here?

TESTING EXERCISE

Test that your simulator works with good old `universe.c`.

6. References

1. [COD5e] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. Elsevier. 5th edition.

7. Major Contributors

This text was made possible by the hard and enduring work of the entire ARK15 Course Team, and in particular the following members of the team:

- Annie Jane Pinder <anpi@di.ku.dk>
- Oleksandr Shturmov <oleks@oleks.info>

A special thanks to Phillip Alexander Roschnowski <roschnowski@gmail.com> for the meticulous proof-reading.

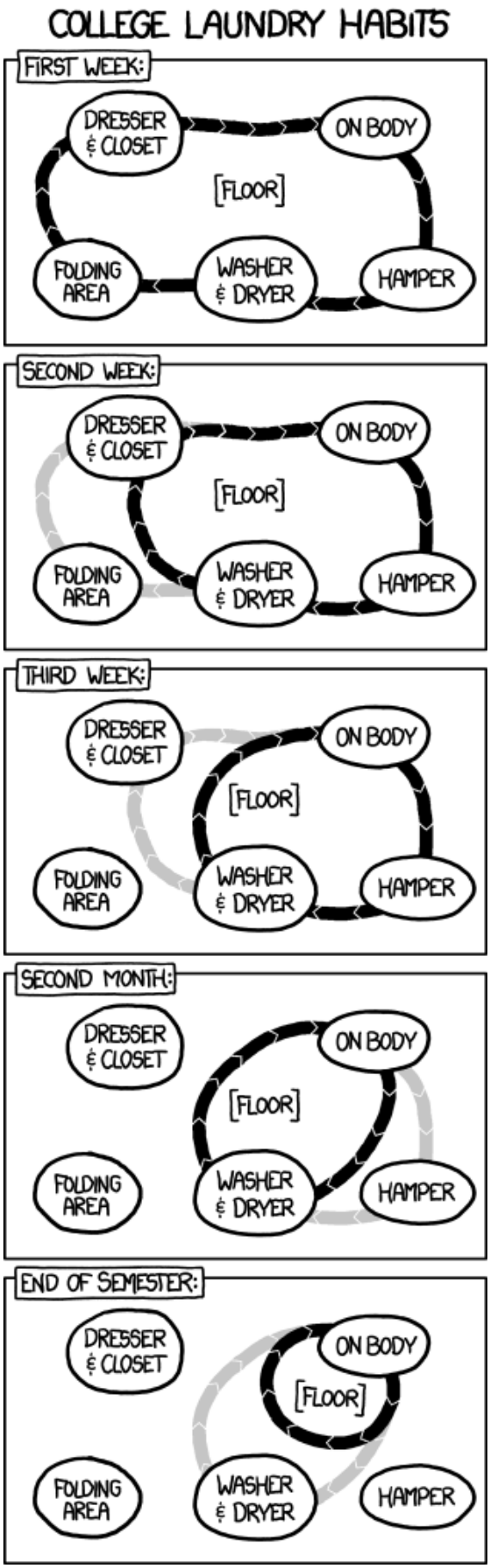

Figure 1. XKCD: Laundry (source: http://xkcd.com/1066/).