MIPS CPU Simulator

Jacob Grunwald and Christy Roney

Introduction:

We think the best introduction here would be to say that our code doesn't run either of the programs we were given. Right now the working theory is that there is something wrong with the branch instructions because that is about as far as we got with debugging this. What this really means is that we can get some code to run correctly, like the beginning of program 1, but only by running our code single cycle. So unfortunately, this means that out of the code we were given for the project, we were able to get only 15 cycles to run correctly on program 1 before things broke.

The goal of this project was actually to help teach us about how five stage MIPS processor with an instruction and data cache and simulated memory controller work, which we do have a much better understanding of now. I think we both really underestimated how much work this was going to be, but we did still start during spring break. We worked consistently throughout the weeks leading up to this due date, but it took us a very long time to understand exactly how to implement this, and I don't think we are alone in thinking this way. Regardless of the final result of our (failed) attempt at creating the MIPS simulator, we both learned a great deal about how the processor works and the pipeline and cache in general.

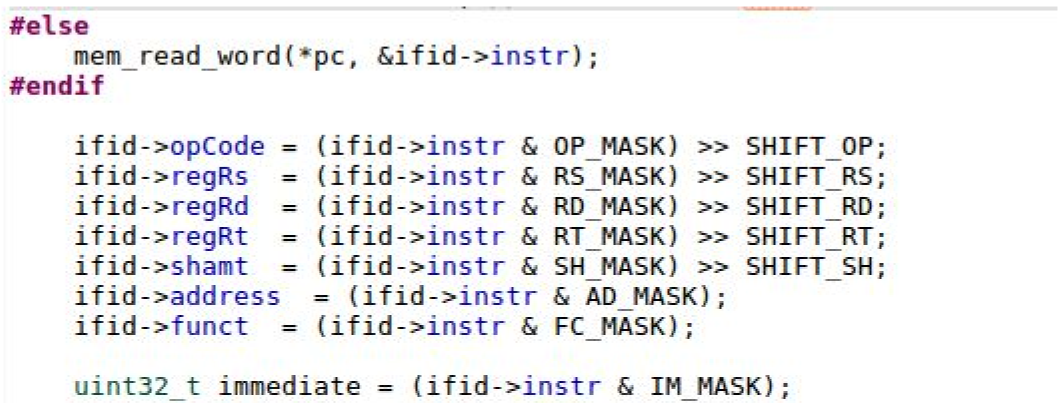
Since we do not have any remotely accurate simulation data aside from the no cache cycle count from the first program, which is still off by about 200 cycles, we will be focusing on the theory behind what we implemented and why we made the choices we did. Where appropriate we will also include snippets of our code to explain and demonstrate our design attempts. We will also include locations we are unsure of the validity of the code and comment on why we think it isn't working.

Finally, since our pipeline is not functioning correctly, we were unable to test our caching, so we will provide our best attempt at answering the questions about the cache simulation. We will primarily be drawing from the theory we have learned in class to inform our answers on these questions.

MIPS CPU Five Stage Pipeline Breakdown:

We will start out by breaking down each stage of the five stage pipeline in detail and describing the requirements to simulate it.

So the next couple sections can be better understood, some background on how we ran our code is needed. We ran our pipeline in reverse, meaning our write back stage ran first and our fetch stage ran last. This was to prevent the need to use shadow registers. We realized that if we initialized our control registers, (IF/ID, ID/EX, EX/MEM, MEM/WB) were initialized to an all zero state then we would essentially be running a nop in all of the registers until they were updated in their respective functions. This meant that on the very first cycle, all control registers would be running nops until the fetch stage was called and the first instruction was pulled into the IF/ID register. In retrospect, this may have cause problems for us, as one of the problems with our simulation seems to be the lack of discrete clocked changes to each register and the pc. More on this later though.



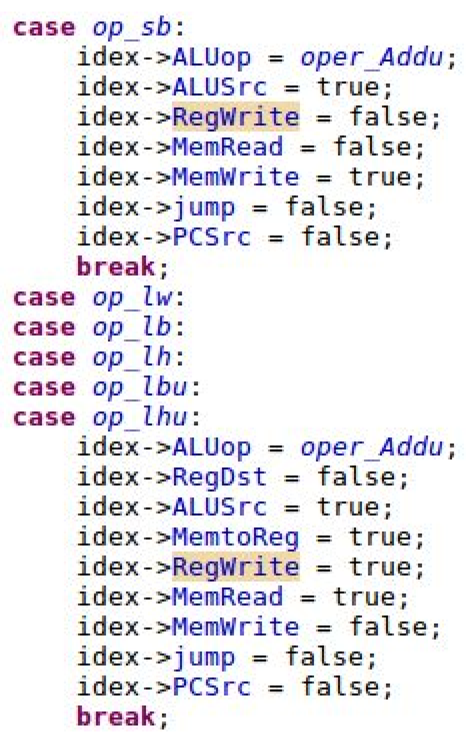
```
#else
    mem_read_word(*pc, &ifid->instr);
#endif

    ifid->opCode = (ifid->instr & OP_MASK) >> SHIFT_OP;
    ifid->regRs  = (ifid->instr & RS_MASK) >> SHIFT_RS;
    ifid->regRd  = (ifid->instr & RD_MASK) >> SHIFT_RD;
    ifid->regRt  = (ifid->instr & RT_MASK) >> SHIFT_RT;
    ifid->shamt  = (ifid->instr & SH_MASK) >> SHIFT_SH;
    ifid->address  = (ifid->instr & AD_MASK);
    ifid->funct  = (ifid->instr & FC_MASK);

    uint32_t immediate = (ifid->instr & IM_MASK);
```

Snippet from Fetch Function

Fetch Stage:

The beginning of any instruction being executed starts with the fetch stage. When code has been compiled it is stored in memory where it can be accessed via the memory controller. The fetch stage pulls assembly instructions from memory and then stores them in the IF/ID register. These values are pulled from memory using the program counter (pc) which keeps track of what line in the code source is running or will be run next. We both feel that this stage was the easiest to implement as it only needed to fetch the data from memory, which means from the i-cache if it is found there, or from main memory. It's worth noting here that programs with many loops would likely benefit more from the i-cache than those with fewer loops because of increased occurrences of hits on cache blocks pulled in due to spatial locality and increased use of temporal locality due to the nature of loops. On the flip side, programs with many more function calls and jumps that are not due to loops could have increased memory access times because the cache could be pulling in instructions that aren't going to be used due to jumping. This would result in a larger number of cache misses overall and a decreased CPI for these programs.

To simulate the fetch stage, we needed to first read in the data from memory. Planning to implement the cache later, we developed a main memory controller that we could use to pull this information from, of course, main memory. After pulling the instruction we needed to break it down into its component parts. We figured that it would be easiest to break down every instruction into all possible combinations of fields that MIPS instructions use. We reasoned that we will be doing all of the actual execution in the execution stage, so we can strategically ignore the values we wouldn't need in execution even though we still have them stored in the control register. For example, if we were running "addu a0, a0, a2" we wouldn't need the immed field, because this is an unsigned addition of two registers. So, although the immed field was still generated, in our ALU we ignore the immed and only use the rs, rt, and rd registers.



```
case op_sb:
    idex->ALUop = oper_Addu;
    idex->ALUSrc = true;
    idex->RegWrite = false;
    idex->MemRead = false;
    idex->MemWrite = true;
    idex->jump = false;
    idex->PCSrc = false;
    break;
case op_lw:
case op_lb:
case op_lh:
case op_lbu:
case op_lhu:
    idex->ALUop = oper_Addu;
    idex->RegDst = false;
    idex->ALUSrc = true;
    idex->MemtoReg = true;
    idex->RegWrite = true;
    idex->MemRead = true;
    idex->MemWrite = false;
    idex->jump = false;
    idex->PCSrc = false;
    break;
```

Snippet from Decode Function

Decode Stage:

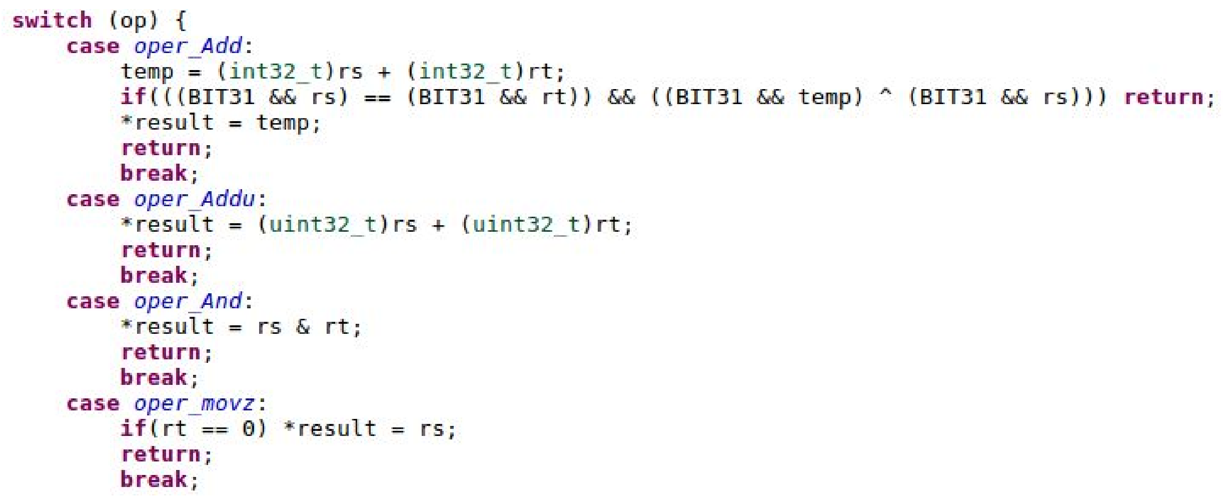
In the decode stage we needed to set up all of the select elements for the rest of the pipeline. This included things like register destination, which would tell the processor which register is being written to when necessary, the program counter source, which tells the processor where to get the next pc value from, and register write, which would tell the processor whether or not to write to the registers. We also set some quality of life things here like our instr which would hold the raw data instruction to help us locate where in code memory our simulator was at.

The decode stage is also responsible for reading data from the register file to prepare for execution as well as doing branch predictions and hazard checks. We think that another of the errors we made in our project are in this stage. We were unable to correctly implement hazard stalls. Specifically, between lines 17 and 18 of program one, we needed a stall and could not make one generate. We think that because of this most of our program counts are off and we may be executing instructions out of order or with incorrect data or partially filled registers. In either case, this is a problem in the decode section. We think that again, this error boils down to poor design choices on our part. For whatever reason, we decided we were going to do hazard checking in a separate function, which made it difficult to debug what was going wrong with our code.

Load data hazard detection and forwarding to the ALU should also happen here. Load data hazard detection should occur whenever a value is being loaded into a register from memory. Because this cannot be forwarded like other instructions, we need to input a stall. Forwarding to the ALU should happen whenever an instruction requires the result of the previous instruction. In these cases, the data will be forwarded from the EX/MEM register to the ID/EX register. In our code, this occurs in the hazard_detection function. Specifically, we check to see if the EX/MEM register is writing to either rs or rt and ID/EX is reading from them. If this is the case, then the value from EX/MEM is forwarded to the ID/EX registers so that the ALU can execute correctly.

When forwarding does happen we also need to recheck the validity of our branches. Because we determine a preliminary branch prediction before we actually do any forwarding or hazard detection, we rerun our branch detection code to update whether or not a branch is taken after the forwarding has been completed. We think this could be another failure point in our code because there is a chance that in setting the branch path twice, we could be overwriting an action that has already partially taken place.

We think that some of our logic in determining where stalls and perhaps forwarding happens may be flawed. It seems that we get forwarding and stalls in most of the places we need them, but cannot tell for sure without stepping through all of the code cycle by cycle, which isn’t feasible because of the thousands of cycles per program.



```
switch (op) {
    case oper_Add:
        temp = (int32_t)rs + (int32_t)rt;
        if(((BIT31 && rs) == (BIT31 && rt)) && ((BIT31 && temp) ^ (BIT31 && rs))) return;
        *result = temp;
        return;
        break;
    case oper_Addu:
        *result = (uint32_t)rs + (uint32_t)rt;
        return;
        break;
    case oper_And:
        *result = rs & rt;
        return;
        break;
    case oper_movz:
        if(rt == 0) *result = rs;
        return;
        break;
```

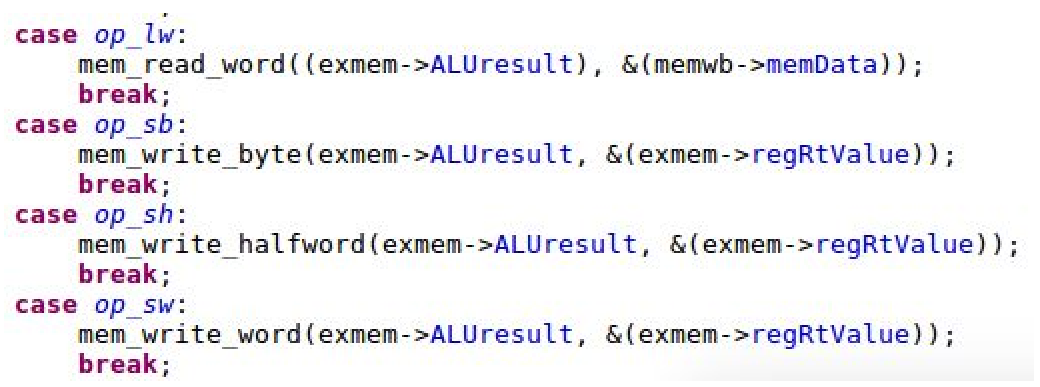
Snippet from ALU Function

Execute Stage:

I thought of the execute stage as the most important stage of the pipeline for a long time. Basically all the way up until I implemented an actual pipeline here. Now I realize that while it is important, it is also relatively straightforward. Basically the execute stage calls the ALU to do any of the math that needs done. That’s it. So the rs value that goes into the ALU comes from the ID/EX register every time. The rt value is chosen either from an immediate if ALU source is high, or from the rt value pulled from the ID/EX register otherwise. We initialize the ALU result to whatever is in the rd register from ID/EX just in case the operation being run doesn’t really require the ALU like branch instructions. The rt, rs and rd registers are then passed into the ALU and provided there is an operation the ALU can actually perform like an add, or addu, or addui, or sub, or anything like that, then the ALU will perform it and store the result in the ALU result field of the control register struct. If no operation is performed then the ALU result field just holds the value from rd.

To simulate the execute stage, we needed to define in the ALU what each operation would be doing. For example, the addu operation will cast rs and rt to unsigned thirty two bit integers then add them together before storing their sum in the ALU result field.

Of all of our pipeline stages, this is the one we are most confident in. Because this stage is so simple, we figured there’s no way we could have messed this one up. Provide instructions are entering the execute stage correctly from other stages, we are confident that they are exiting the execute stage correctly as well.



```
case op_lw:
    mem_read_word((exmem->ALUresult), &(memwb->memData));
    break;
case op_sb:
    mem_write_byte(exmem->ALUresult, &(exmem->regRtValue));
    break;
case op_sh:
    mem_write_halfword(exmem->ALUresult, &(exmem->regRtValue));
    break;
case op_sw:
    mem_write_word(exmem->ALUresult, &(exmem->regRtValue));
    break;
```

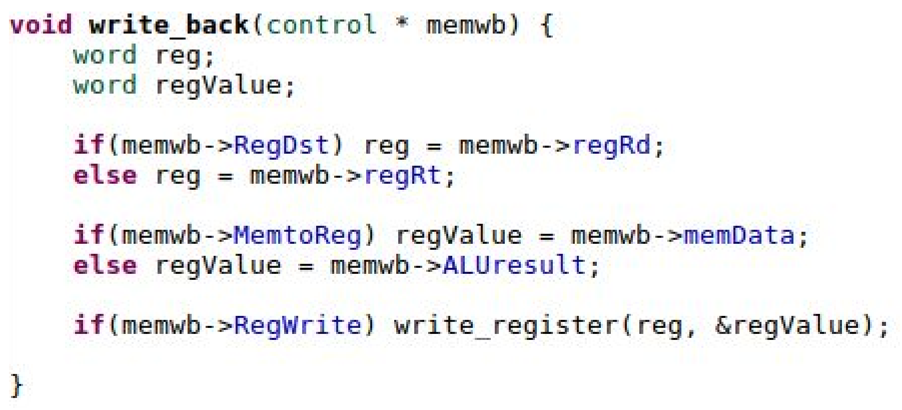
Snippet from Memory Access Function

Memory Access Stage:

Now that we are writing out descriptions of each stage and reviewing the work we put into each, it seems that the most difficult stages really are the first two. Once we get past there, things become relatively simple. Or at least that’s how we saw things. It could be that our errors in our simulation are coming from oversimplifications of the last stages. We really hope not though. That would be disappointing.

The memory access stage will handle all of the loads and stores for the processor. In the case of a load, it will check the cache for the data first, and if it does not find it, will go to main memory to pull it into the cache. If using early start functionality, the cache will continue operation after the data it needed is fully brought into the cache. In this instance, it means that after six clock cycles, the pipeline will continue, and as the cache continues to read data into itself, there will be one cycle bubbles inserted into the pipeline to make sure that the data needed for each instruction is present when it is needed.

When handling writes, if using a write back cache, the data will be stored primarily in the cache, and will only be written to main memory before it is updated again. A write through cache will be slightly slower to respond due to the writing to main memory and writing to the cache, but has the benefit that reads from memory after that will have correct data.



```c
void write_back(control * memwb) {
    word reg;
    word regValue;

    if(memwb->RegDst) reg = memwb->regRd;
    else reg = memwb->regRt;

    if(memwb->MemtoReg) regValue = memwb->memData;
    else regValue = memwb->ALUresult;

    if(memwb->RegWrite) write_register(reg, &regValue);

}
```

Snippet from Write Back Function

Write Back Stage:

The write back stage is very simple in the simulation. It will write to registers if the register write flag is high. This means that if the instruction requires a value to be written to the registers, then the registers will actually be written at this point.

We are also fairly confident that this stage works. We tested our memory controller a good amount and found it to be reliable. This on top of our accurate readings of instructions and data from main memory make us confident that the issues in our simulator are elsewhere.

To simulate the write back stage, we checked if the MEM/WB register had received the register write flag from the decode stage and if it did, we wrote either the retrieved memory data or the ALU result to the appropriate register for the instruction.

Cache Breakdown:

All of our cache schemes support an early start function on instruction cache fills along with a single deep write buffer on write through and write back caches to improve performance. On write back, our write buffer holds the data to be written to occur until the new cache block has been filled. After the new cache block has been loaded into the data cache, the write buffer then will write the data in the write buffer to main memory. The cache contains a block size of 4 with a read miss penalty of 8 and a cycle penalty of 2. A write to memory will cause main memory to be busy, not able to support a memory access, of 6 clock cycles on the first word of a block and 2 clock cycles for subsequent blocks. Also, our cache memory becomes the instruction and data memories of the pipeline and contain separate instruction and data caches (split caches).

When writing to the cache (write-allocation) for a write-back, we wait to write dirty data until eviction. For a hit, data should be written to the cache, the valid bit and data bit should be set to 1, and the tag needs to be set as well. On a miss for an invalid tag, our code evicts the block currently taking up space needed for this data. Specifically, if the dirty bit is 0, then the valid bit should be set to 0. If the dirty bit is 1, then our code writes the whole evicted block to the write buffer, then it sets the cache valid bit to 0 and the dirty bit to 0. Also, on a miss for an invalid tag, our code allocates cache line for data, loads the block from memory, writes to the cache, allows the instruction to continue and then the writes the evicted block in write buffer to main memory. Plus, the cache block's valid bit and dirty bit need to be set to 1 and the tag needs to be filled. On a write through, our code writes to memory and the cache at the same time. On a hit, write data to cache and write data to memory. On a miss for an invalid tag, our code evicts current block from cache and we don't need to write it to memory. Plus, our code loads the correct block from memory into the cache, performs the write, sets the dirty bit, and writes the updated word to main memory (not the whole block).

When reading from the write-allocate cache, for write-back, on a hit, our code returns the specified word. On a miss for an invalid tag, if the block is dirty, our code evicts the block to the write buffer and sets the valid and dirty bits to 0. If the block is clean, we just set our valid and dirty bits to 0. Then, we load the correct block from main memory. Evictions don't need to write to memory on write-through because they were written before, when they were modified.