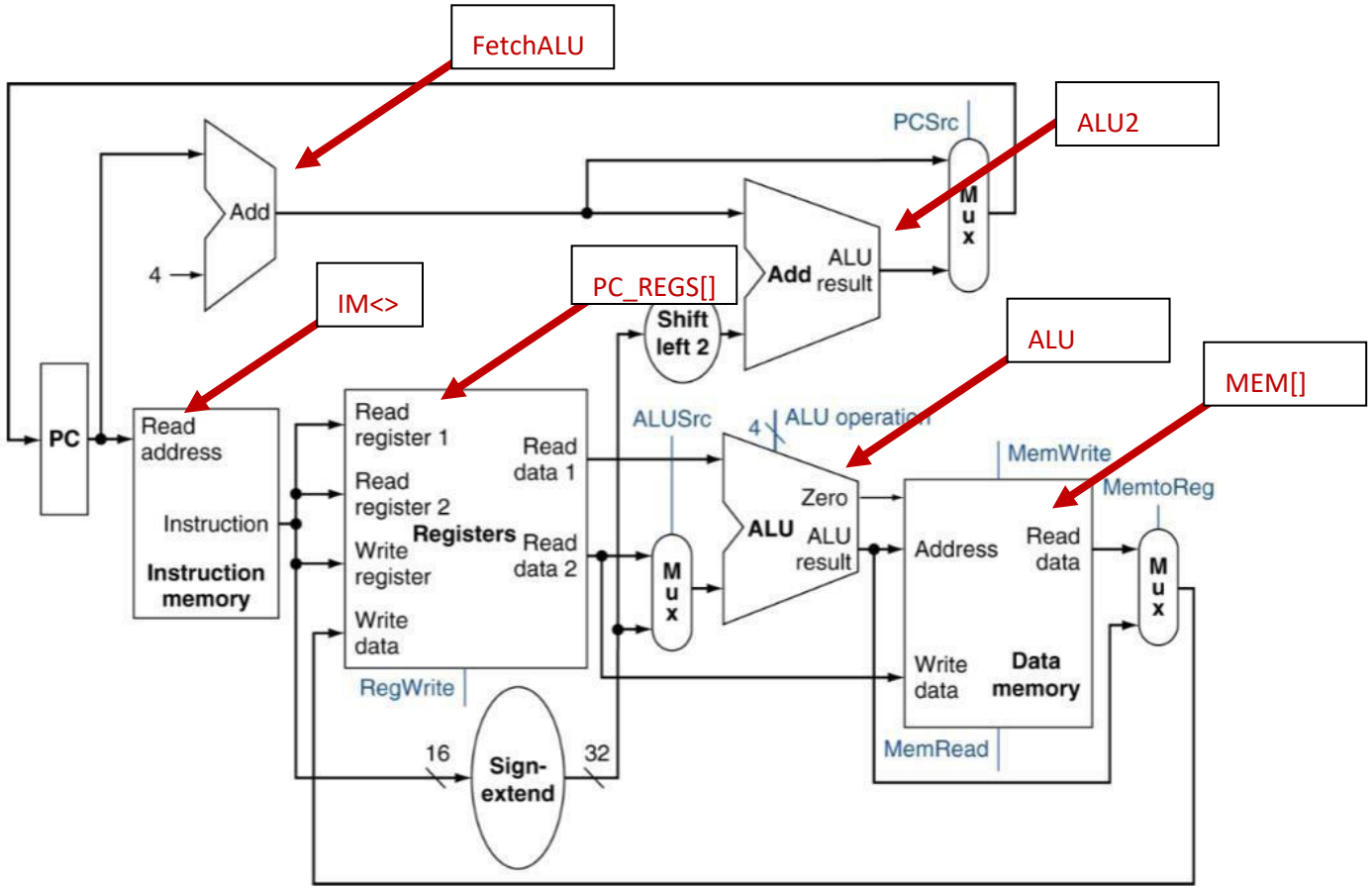
MIPS/Single Datapath Processor Learning Tool

Introduction

Many upcoming computer science students often have a difficult time understanding what is happening inside of a computer when a program is executed, since up until the first few semesters, they will have likely only used higher-level languages, such as C++, and perhaps a single semester with ISAs, such as MIPS, in an assembly language course. When diving even further down into abstraction, to what is happening on an architectural level, it can become very difficult for these newer students to understand, since requires a deep understanding of many lower-level concepts. However, I have decided to use their limited knowledge of MIPS and modular higher-level programming to our advantage and created a program that new CS students can look over in order to easily digest what is happening within a single datapath processor.

Getting with the program…



***Components from this point forward will be represented in red and functions will be **bold().**

The above diagram represents the different elements of the single datapath processor, and the names associated with them, as they are represented in our program.

NOTE:

- We do not have a sign-extension unit, as we are dealing with decimal numbers, and we are not processing offsets in binary.
- For this same reason, we do not need a shift-left unit when processing branches or jumps, either.
- We also do not have muxes, nor do we have the logical AND gate used for executing branches, as their functionality can be represented with conditional statements, which are present within the functions themselves.

For an overview of the structure of the program, please refer to the presentation and source code. For a description of how to run the program, please refer to the README.md file.

A couple of examples…

Next, we will go through a series of examples a user can input, to explain in detail what is happening. We will begin by processing an arithmetic instruction, then a load/store instruction, a branch instruction, and lastly a jump instruction. We will walk through the process of what is happening, as it progresses through the program. Before we move on, it is important to note that the user may enter whatever values they want into the PC_REGS and MEM arrays (located at the top of the components.cpp file) that they would like to test. Upon running the program, the user is notified of the registers and memory available. Registers $r1-$r10 represent PC_REGS indices [0-9], and s$1-$s10 represent MEM indices [0-36] (memory is accessed through multiples of 4).

```
add $r1, $r2, $r3
```

```
int main() {
  std::ofstream file;
  file.open ("output.txt");
  int num_lines = 0;
      std::cout << "PC registers:      $r1-$r10 \n"
           << "Memory slots available: 10 \n"
           << "input operations into IM...\n";
  while (instruction != "stop") {
      std::cout <<"     ";
      getline (std::cin, instruction);
      if(instruction == "stop"){
          break;
      }
      num_lines++;
      IM.push_back(instruction);
      file << num_lines << "        " << instruction <<"\n";
      amt_instructions++;
    }
  file.close();
  bool first_inst = true;
  fetch(IM, amt_instructions, first_inst);
  std::cout << "NO INSTRUCTIONS LEFT, DISPLAYING MEMORY \n";
  printArry();
  std::cout << "\n\nDONE!";
  return 0;
}
```

Let's say the user would like to implement an add instruction. They would begin by typing the command in the command-line in very much the same way they would in a MIPS environment.

The instruction gets pushed into the IM vector. I decided to use a vector in order to not limit the number of instructions that the user can enter. Next, a count of the instructions (simulating the line-number, which is incremented for every instruction entered), is published to our output.txt file, as well as the instruction itself.

This file will be important if the user decides to process jumps or branches. Next, we call the **fetch()** function from the stages.cpp file.

```
void fetch (std::vector <std::string> instructions, int &amt_instructions,
            bool &first_inst) {
  std::string op1, op2, reg1, reg2, reg3;
  std::stringstream ss;
  if (first_inst) {
      std::cout<<"FETCHING FIRST...\n";
      ss << instructions.front();
      ss >> op1 >> reg1 >> reg2 >> reg3;
      first_inst=false;
      decode(op1, reg1, reg2, reg3);
      instructions.erase(instructions.begin());
      amt_instructions--;
      ss.clear();
      fetch(instructions, amt_instructions, first_inst);
  }else{
  std::cout << "FETCHING...\n";
  inFetchALU(instructions, amt_instructions, first_inst);
  }
}
```

Now that we are inside the **fetch()** function, we create a series of string objects which will represent the inputs by the user. Then, if it is the first instruction (indicated by bool first_inst passed in from **main()**), then we gather the stringstream from the front of the IM vector, and immediately start decoding, as the first instruction doesn't require the ALU to calculate it from the PC. After the instruction is processed in its entirety, it is erased from the array, and the stringstream is cleared to make way for the next instruction. Then, **fetch()** gets called recursively until there are no instructions left. After the first instruction has been processed, next time **fetch()** is called, the first_inst bool will be set to false, and the fetch ALU will now need to acquire all remaining instructions.

But let's continue on with our ALU instruction, which has now been passed to **decode()**.

```
void decode (std::string operation, std::string reg1, std::string reg2,
             std::string reg3) {
  std::cout <<"  DECODING...\n";
  std::string regs[3] = {reg1, reg2, reg3};

  if ((operation == "add" && operation[3] == '\0') ||
      (operation == "sub" && operation[3] == '\0') ||
      (operation == "mul" && operation[3] == '\0') ||
      (operation == "and" && operation[3] == '\0') ||
      (operation == "sll" && operation[3] == '\0') ||
      (operation == "srl" && operation[3] == '\0') ||
      (operation == "or"  && operation[3] == '\0') ||
      (operation == "ncp")) {
    opcode = "000000";
  } else if (operation == "addi") {
    opcode = "001000";
  } else if (operation == "lw"){
    opcode = "010111";
  } else if (operation == "beq") {
    opcode = "000100";
  } else if (operation == "j") {
    opcode = "000010";
  } else if (operation =="sw"){
    opcode = "101011";
  } else {
    std::cout << "ERR: please enter a valid command...\n";
    return;
  }
  int i = 0;
  int first = 1;
  bool is_valid = true;
  RFRead (regs, operation, i, first, is_valid);
  if(is_valid){
  execute(opcode, operation, first);
  }else{
    return;
  }
}
```

Now we are in **decode()**. The entirety of the user's input has been segmented into separate strings (representing the operation and registers entered) which have now been passed to the **decode()** function. The program then goes through a series of conditional statements to find out what kind of instruction the program is currently executing. Once it is found, the program assigns its corresponding opcode. If no valid operation is found, then an error message is thrown, and the program simply moves on to the next instruction.

Next, the rest of the strings (not including the operation), are passed onto the **RFRead()** function, which reads what registers the user has entered.

```
void RFRead(std::string regs[], std::string operation, int &i, int &first,
            bool &is_valid) {
    if (i ==0) {
       std::cout <<"    READING REG FILE...\n";
    }
    if (operation == "nop"){
        return;
    }
    else if ((regs[i] == "" || i >= 3) && (operation == "add" ||
                                           operation == "sub" ||
                                           operation == "mul" ||
                                           operation == "and" ||
                                           operation == "or")) {
      return;
    } else if ((regs[i] == "" || i >= 2) && ((operation == "sll") ||
                                             operation == "srl" ||
                                             operation == "addi")) {
      ALU[i] = stoi(regs[i]);
      return;
    } else if ((regs[i] == "" || i >= 1) && operation == "lw") {
      ALU[i] = stoi(regs[i]);
      return;
    } else if ((regs[i] == "" || i >= 1) && operation == "sw") {
      ALU[i] = stoi(regs[i]);
      return;
    } else if ((regs[i] == "" || i >= 2) && operation == "beq") {
      ALU[i] = stoi(regs[i]);
      return;
    }
    else if ((regs[i] == "" || i >= 0) && operation == "j") {
      ALU[i] = stoi(regs[i]);
      return;
    }

  for (int j = 0; j < 10; j++) {
    if (regs[i] == "$r1," || regs[i] == "$r1") {
      ALU[i] = PC_REGS[0];
      if (i == 0) {
        first = 1;
      }
      break;
    } else  if (regs[i] == "$r2," || regs[i] == "$r2") {
      ALU[i] = PC_REGS[1];
      if (i == 0) {
        first = 2;
      }
      break;
    } else  if (regs[i] == "$r3," || regs[i] == "$r3") {
      ALU[i] = PC_REGS[2];
      if (i == 0) {
        first = 3;
      }
      break;
    } else  if (regs[i] == "$r4," || regs[i] == "$r4") {
      ALU[i] = PC_REGS[3];
      if (i == 0) {
        first = 4;
      }
      break;
    } else  if (regs[i] == "$r5," || regs[i] == "$r5") {
      ALU[i] = PC_REGS[4];
      if (i == 0) {
        first = 5;
      }
      break;
    } else  if (regs[i] == "$r6," || regs[i] == "$r6") {
      ALU[i] = PC_REGS[5];
      }
      break;
    } else  if (regs[i] == "$r8," || regs[i] == "$r8") {
      ALU[i] = PC_REGS[7];
      if (i == 0) {
        first = 8;
      }
      break;
    } else  if (regs[i] == "$r9," || regs[i] == "$r9") {
      ALU[i] = PC_REGS[8];
      if (i == 0) {
        first = 9;
      }
      break;
    } else if (regs[i] == "$r10," || regs[i] == "$r10") {
      ALU[i] = PC_REGS[9];
      if (i == 0) {
        first = 10;
      }
      break;
    } else {
      std::cout << "ERR: invalid registers... \n";
      is_valid = false;
      return;
    }
  }
  i++;
  RFRead(regs, operation, i, first, is_valid);
  return;
}
```

This chunk of code represents the register file. It parses through recursively, until it finds all of the registers/values that the user has entered. However, certain instructions require a different number of registers/values. For example, add requires three (a register to write to, and the other two to add), while lw requires only two (the register to write to, and the offset to acquire from memory). This is handled by a variable which tracks the number of times that the function has been ran through, as well as the operation, which has also been passed to the **RFRead()** function. A series of conditional statements at the top ensures that the program has iterated through the correct number of times. If an incorrect register has been entered, then an error is called, and the is_valid bool which has been passed to **RFRead()** is flagged as false. If it has been flagged false, then when we return to the **decode()** function, then we simply return, and go to the next function after throwing an error message. Otherwise, after all of the valid registers/values have all been found, then the results are stored in the ALU array as inputs, and we return back into the **decode()** function, and we immediately call **execute()**.

```
void execute(std::string opcode, std::string operation, int pos) {

  std::cout <<"       EXECUTING...\n";
  controlUnit(opcode, operation, pos);
  return;
}

void controlUnit(std::string opcode, std::string operation, int pos) {
  std::cout <<"         IN CONTROL UNIT...\n";
  if (opcode == "000000") {
    std::cout <<"           TO ALU...\n";
    inALU(operation, pos);
  } else if (opcode == "001000"){
    std::cout <<"           TO ALU...\n";
    inALU(operation, pos);
  } else if (opcode == "010111"){
    std::cout <<"           TO ALU...\n";
    inALU(operation, pos);
  } else if (opcode == "101011") {
    std::cout <<"           TO ALU...\n";
    inALU(operation, pos);
  } else if (opcode == "000100"){
    std::cout <<"           TO ALU...\n";
    inALU(operation, pos);
  } else if (opcode == "000010"){
    std::cout <<"           TO ALU...\n";
    inALU(operation, pos);
  }
  return;
}
```

execute() is merely called to inform the user that we are currently at that stage in the process. Other than that, it immediately calls **controlUnit()** inside the components file.

Inside the control unit, we gather the opcode that we assigned from our **decode()** function, and direct it to the next component, the ALU. The rest of the control unit's functionality will be simulated through the operation, which will be passed from function to function from this point forward.

```
void inALU(std::string operation, int pos) {
  bool zero = false;
  std::cout << "            IN ALU...\n";
  if (operation == "nop"){
      writeBack(PC_REGS, ALU[0], pos, operation);
  }
  else if(operation =="add") {
    ALU[0] = ALU[1] + ALU[2];
    writeBack(PC_REGS, ALU[0], pos, operation);
  } else if (operation == "sub") {
    ALU[0] = ALU[1] - ALU[2];
    writeBack(PC_REGS, ALU[0], pos, operation);
  } else if (operation =="mul") {
    ALU[0] = ALU[1] * ALU[2];
    writeBack(PC_REGS, ALU[0], pos, operation);
  } else if (operation =="and") {
    ALU[0] = (ALU[1] & ALU[2]);
    writeBack(PC_REGS, ALU[0], pos, operation);
  } else if (operation =="or") {
    ALU[0] = (ALU[1] | ALU[2]);
    writeBack(PC_REGS, ALU[0], pos, operation);
  } else if (operation =="sll") {
    ALU[0] = (ALU[1] << ALU[2]);
    writeBack(PC_REGS, ALU[0], pos, operation);
  } else if (operation =="srl") {
    ALU[0] = (ALU[1] >> ALU[2]);
    writeBack(PC_REGS, ALU[0], pos, operation);
  } else if (operation == "addi") {
    ALU[0] = (ALU[1] + ALU[2]);
    writeBack(PC_REGS, ALU[0], pos, operation);
  } else if (operation == "lw") {
    inMem(MEM, pos, ALU[1], operation);
    return;
  } else if (operation == "sw") {
    inMem(MEM, ALU[0], ALU[1], operation);
    return;
  } else if (operation == "beq") {
    zero = (ALU[1]-ALU[0]);
    if (zero == 0){
    std::cout << "              $zero REGISTER TRUE, REGISTERS EQUAL...\n";
    bool zero = true;
    inALU2(ALU[2]);
    } else {
      return;
    }
    return;
  } else if (operation == "j") {
    inALU2(ALU[0]);
  }
  return;
}
```

Now, we are in the main ALU. As stated previously, our operation gets passed in, and this allow us to know which operation to execute *within* the ALU. In this case, the user has entered the add instruction. At which point, it meets the corresponding conditional statement, and the add operation is executed. The two inputs (represented by indices 1 and 2) are added together, and stored in it's output (index 0), which are then finally written back into the register file. **writeback()** gets called to accomplish this.

```
void writeBack(int PC_REGS[], int result, int pos, std::string operation) {
  if (operation == "nop") {
     std::cout <<"               NO OPERATION...\n";
     return;sn
  }else{
      std::cout <<"               WRITING BACK TO REG FILE...\n"
                << "                          "
                << result << " result stored in $r" << pos <<  "\n";
      PC_REGS[pos-1] = result;
      return;
  }
  return;
}
```

At this point, we finally take the result from the ALU's output, and we write it back into PC_REGS. The correct position in the array is found via the pos variable, which has been passed down, all the way from **decode()** (then called first), and the change is indicated to the user. Finally, the function returns, and all of the rest do the same, up until the point that it hits the fetch ALU, at which point it will fetch the next instruction (if there is one).

lw $r2, 36

In the case of a load instruction, all of the same above steps are accomplished up until the point that the register and offset is read from the ALU. Rather than **writeback()** being called, we instead call the **inMem()** function.

```
void inMem(int MEM[], int reg_pos, int mem_pos, std::string op) {

  std::cout <<"              IN MEM...\n";

  if (op != "sw" && mem_pos%4 == 0){
  int result = MEM[mem_pos/4];
  writeBack(PC_REGS, result, reg_pos, op);
  return;
  } else if (op == "sw" && mem_pos%4 == 0) {
    std::cout << "                          "
         << reg_pos << " result stored to $s" << (mem_pos/4)+1 <<"\n";
    MEM[mem_pos/4] = reg_pos;
  }
  else {
    std::cout << "ERR: incorrect memory address...";
  }
  return;
}
```

From this point, the program ensures that it isn't **inMem()** in order to write to it (in the case of a sw instruction), and it accomplishes this with a conditional statement, which checks the operation. In the same conditional statement, it also ensures that a valid memory address has been entered. It accomplishes this by ensuring that the memory position (variable mem_pos) is divisible by 4 (mod%4 == 0). If the conditions are met, then result acquires the correct value from memory, and writes it back into the corresponding register, by calling **writeback()**.

sw $r2, 36

A store instruction accomplishes all of the same things a load instruction does, except when it is called to **inMem()**, and it hits the conditional statements, it meets the second conditional's requirements. At which point, it immediately gets written into memory.

beq $r2, $r2, 1

```
  } else if (operation == "beq") {
    zero = (ALU[1]-ALU[0]);
    if (zero == 0){
    std::cout << "              $zero REGISTER TRUE, REGISTERS EQUAL...\n";
    bool zero = true;
    inALU2(ALU[2]);
    } else {
      return;
    }
    return;
  } else if (operation == "j") {
    inALU2(ALU[0]);
  }
  return;
}
```

A branch instruction effectively does all of the same things an ALU instruction does, up until it hits the main ALU, at which point, it does the comparison of the two registers. If the register values are equal, then subtracting both of them should give us a result of zero. We can use the zero bool to represent the main ALU's $zero register. If the two values equal zero, the zero bool is set to true, the user is informed that the two registers are equal, and the second ALU is called to calculate where to jump. If they are not equal, then the function merely returns.

```
void inALU2(int target) {

  std::cout << "               IN FETCH ALU (processing jump)...\n";

  std::string mytarget = std::to_string(target);
  bool found = false;

  std::string line;
  std::string line2;
  int amt = 0;
  std::ifstream countlines;
  countlines.open("output.txt");

  while (getline(countlines, line2)){
    amt++;
  }

  std::fstream file; file.open("output.txt", std::ios::in | std::ios::out);
  countlines.close();
  countlines.open("output.txt");
    while(!found && amt > 0){
      file >> line;
      if(line == mytarget) {
        std::cout << "                    JUMP FOUND (check output.txt)...\n";
        file << "HERE:";
        found = true;
        return;
      } amt--;
        getline(file, line2);
    }
    std::cout << "ERR: Jump not found...\n";
}
```

When the second ALU gets called, we are now going to utilize our output.txt file. We open the file, and we go through a simple while loop in order to figure out the number of lines that the program has written to it. Then, the file gets iterated through a second time, with the target and, through a file input stream, finds the line number it is currently on. With each iteration, the line number variable also gets decremented, in order to ensure we have not exceeded the size of the file.

When the line number matches the target line that the user has entered, then "HERE:" gets written onto that line of the output file, the found bool is set to true, and we break out of the loop. **NOTE**: Since the file pointer may not exceed the size of the file, we can only make jumps backward, not forward. If the jump exceeds the size of the file, then a Jump not found error is prompted to the user, and the function ends.

```
j  2
```

Jump instructions are handled the same way as a branch is, however, instead of making the comparison, the target is immediately thrown into the second ALU, the file is parsed, the target is either found, in which case “HERE:” gets written to the target line, or the error message is thrown.