Hardware and Software Laboratory Project 3 (Hardware)
SIMPLE Design Resources

Ver. 4.0: 04/15/2020

1 Introduction

This document is related to computer hardware design that is an assignment for Hardware and Software Laboratory Project 3. The computer that will be designed, SIMPLE (SIxteen-bit MicroProcessor for Laboratory Experiment), treats one word as 16 bits and has a relatively simple instruction set, but it is equipped with all of the basic functionalities that a computer should have. Consequently, correctly designing SIMPLE is an important first step towards designing more advanced computers.

In Chapter 2, this document details SIMPLE's basic architecture, that is to say, what kinds of register and memory it has as well as what kinds of formats and functions the instructions to be executed have. Following this, in Chapter 3 this document gives a small glimpse into the simplest way to design a machine based upon this architecture.

It is worth noting that students will not merely be following the design specifications and roadmap provided for them; rather, this assignment also necessitates that students apply original improvements and extensions while evaluating and considering their effectiveness. As a helpful reference, potential improvements and extensions to the instruction set architecture and the microarchitecture are included in Chapter 4.1.

2 SIMPLE's Architecture

A computer's architecture refers to a defined list of the various resources that it has access to, such as registers and memory, as well as the functions of the instructions that operate them. Fundamentally, architectures determine the logical structure and functions, they do not define how these things are actually implemented. As such, it is common practice to design various hardware based on a single architecture. However, no matter how it is designed, the crucial point is that the programs must behave in exactly the same manner. Architectures related to details of hardware designs are referred to as microarchitectures.

2.1 Main memory and register

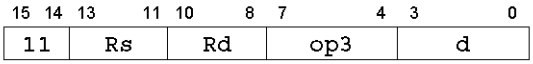
SIMPLE is a computer that uses 16 bits for one word. Its main memory and register are both 16 bits wide.

- Main memory

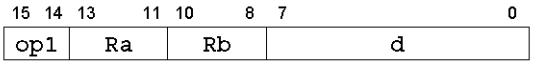
  Main memory addresses are 16 bits and each word is addressable. As such, address space size is 64 KW. Henceforth, accessing a word with address a will be denoted via *(a) (like in the C language).
- Register

  SIMPLE is equipped with eight general purpose registers denoted as r[0], r[1],..., r[7]. These are used for calculations related to operation sources/destinations as well as main memory addresses.
- Program counter

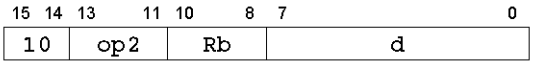
  Retains the address of instructions currently being executed. Denoted as PC.



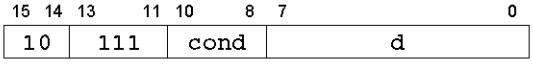
(a) Operation/input-output instruction format



(b) Load/store instruction format



(c) Load immediate/unconditional branch instruction format



(d) Conditional branch instruction format

Figure 1: SIMPLE's Instruction Formats

- Condition codes

  It consists of the four flags that retain the branching conditions for the result of operation instructions, S (sign), Z (zero), C (carry), and V (overflow), and they are configured as shown below.

  S: 1 if negative, 0 otherwise

  Z: 1 if zero, 0 otherwise

  C: 1 if there is a carry, 0 otherwise

  V: 1 if the operation result exceeds that which can be represented as a signed 16 bit, 0 otherwise

  It is worth noting that the explanation about the C value except for addition, subtraction, and comparison instructions is described later.

2.2 Instruction set architecture

Instruction format

SIMPLE's instructions are always fixed to one word (16 bits) in length. The four types of instruction formats can be seen in Figure 1. The meaning of each instruction format and field is as follows.

(a) Operation/input-output instruction format

- $I_{15:14}$ (op1).....Operation code (11) (opcode)
- $I_{13:11}$ (Rs).....Source register number
- $I_{10:8}$ (Rd).....Destination register number
- $I_{7:4}$ (op3).....Operation code (0000 to 1111)
- $I_{3:0}$ (d).....Shift number of digits

(b) Load/store instruction format

- $I_{15:14}$ (op1).....Operation code (00/01)
- $I_{13:11}$ (Ra).....Source/destination register number
- $I_{10:8}$ (Rb).....Base register number
- $I_{7:0}$ (d).....Displacement

(c) Load immediate/unconditional branch instruction format

- $I_{15:14}$ (op1).....Operation code (10)
- $I_{13:11}$ (op2).....Operation code (000 to 110)
- $I_{10:8}$ (Rb).....Source/destination/base register number
- $I_{7:0}$ (d).....Immediate or displacement

(d) Conditional branch instruction format

- $I_{15:14}$ (op1).....Operation code (10)
- $I_{13:11}$ (op2).....Operation code (111)
- $I_{10:8}$ (cond).....Branch condition
- $I_{7:0}$ (d).....Displacement

Operations/Input-output instructions

SIMPLE's operations/input-output instructions are shown in Table 1. For operation instructions, condition codes based on the result are configured.

1. Arithmetic operations
   Results of addition (ADD:add) and subtraction (SUB:subtract) for registers Rd and Rs are stored in Rd and condition codes are configured. For condition code C, the carry is configured from the most significant bit.

2. Logical operations
   Results of bit-wise logical product (AND:and), bit-wise logical sum (OR:or), or exclusive logical sum (XOR:exclusive-or) for registers Rd and Rs are stored in Rd and condition codes are configured. However, condition code C will be 0 regardless of the result of the operation.

3. Comparison operations (CMP:compare)
   Register Rs is subtracted from register Rd, and condition codes are only configured based on the result. For condition code C, the carry is configured from the most significant bit.

4. Movement operations (MOV:move)
   Simply stores the value of register Rs in register Rd, creating condition codes based on the value of Rd. However, condition code C will be 0 regardless of the value of Rs.

5. Shift operations
   The value of register Rd is shifted in the manner shown below and stored in Rd. Condition codes are configured.

   - SLL (shift left logical)..... After shifting left, a 0 is put into the empty spot.

- SLR (shift left rotate)..... After shifting left, the bit column that was shifted out is put into the empty spot.
- SRL (shift right logical)..... After shifting right, a 0 is put into the empty spot.
- SRA (shift right arithmetic)... After shifting right, the value of the signed bit is put into the empty spot.

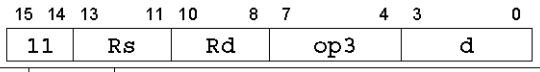
The number of digits shifted is immediate d (0 to 15). Condition code C is 0 both for SLR and when the number of digits shifted is 0. However, in all other cases it is set to the value of the last bit that was shifted out. Condition code V is always set to 0.

6. Input-output instruction

- IN (input)..... Stores a value input via a switch or some other device into register Rd.
- OUT (output).... Outputs the value of register Rs to 7SEG LED or some other device.
- HLT (halt)..... Halts SIMPLE.

Instructions denoted as “(reserved)” will not do anything aside from moving on to the next instruction.[1]

Table 1: SIMPLE’s Operations/Input-Output Instructions



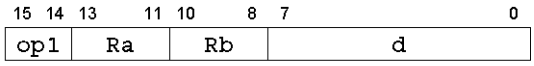
mnemonic	op3	function
ADD Rd,Rs	0000	r[Rd] = r[Rd] + r[Rs]
SUB Rd,Rs	0001	r[Rd] = r[Rd] - r[Rs]
AND Rd,Rs	0010	r[Rd] = r[Rd] & r[Rs]
OR Rd,Rs	0011	r[Rd] = r[Rd] | r[Rs]
XOR Rd,Rs	0100	r[Rd] = r[Rd] ^ r[Rs]
CMP Rd,Rs	0101	r[Rd] - r[Rs]
MOV Rd,Rs	0110	r[Rd] = r[Rs]
(reserved)	0111	
SLL Rd,d	1000	r[Rd] = shift_left_logical(r[Rd],d)
SLR Rd,d	1001	r[Rd] = shift_left_rotate(r[Rd],d)
SRL Rd,d	1010	r[Rd] = shift_right_logical(r[Rd],d)
SRA Rd,d	1011	r[Rd] = shift_right_arithmetic(r[Rd],d)
IN Rd	1100	r[Rd] = input
OUT Rs	1101	output = r[Rs]
(reserved)	1110	
HLT	1111	halt()

[1] In accordance with the policy “we cannot guarantee the behavior of undefined instructions,” it is okay for one to make these into don’t-care terms.

Load/store instruction

The functions of SIMPLE's load instruction (LD:load) and store instruction (ST:store) are shown in Table 2. The source/destination is register Ra designated in field Ra. The effective address is found by adding register Rb designated in field Rb by designation of the base register address and sign_ext(d), which is the code extension of field d.

Table 2: SIMPLE's Load/Store Instructions



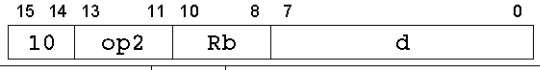
mnemonic	op1	function
LD Ra,d(Rb)	00	r[Ra] = *(r[Rb] + sign_ext(d))
ST Ra,d(Rb)	01	*(r[Rb] + sign_ext(d)) = r[Ra]

Load immediate/unconditional branch instructions

The functions of SIMPLE's load immediate instruction (LI:load immediate) and unconditional branch instruction (B:branch) are shown in Table 3.

- LI.....Stores immediate sign_ext(d) in register Rb.
- B.....Takes the code-extended value of d as the displacement and branches by designating the PC relative address.

Table 3: SIMPLE's Load Immediate/Unconditional Branching Instructions



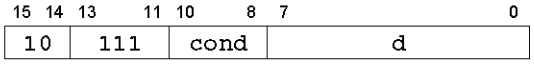
mnemonic	op2	function
LI Rb,d	000	r[Rb] = sign_ext(d)
(reserved)	001	
(reserved)	010	
(reserved)	011	
B d	100	PC = PC + 1 + sign_ext(d)
(reserved)	101	
(reserved)	110	
(Conditional branch instruction)	111	Refer to Table 4.

Conditional branch instruction

As shown in Table 4 below, SIMPLE's conditional branching instruction branches according to the PC relative address if the branching condition in field cond is TRUE. If the condition is FALSE, it simply moves to the next instruction. The branching condition for each instruction is detailed below.

- BE (branch on equal-to)..... Condition code Z is 1
- BLT (branch on less-than)..... .XOR (S ^ V) of condition codes S and V is 1
- BLE (branch on less-than or equal-to)... Z or (S ^ V) is 1
- BNE (branch on not-equal-to)..... Condition code Z is 0

Table 4: SIMPLE's Conditional Branching Instructions



mnemonic	cond	function
BE d	000	if (Z) PC = PC + 1 + sign_ext(d)
BLT d	001	if (S ^ V) PC = PC + 1 + sign_ext(d)
BLE d	010	if (Z || (S ^ V)) PC = PC + 1 + sign_ext(d)
BNE d	011	if (!Z) PC = PC + 1 + sign_ext(d)
(reserved)	100	
(reserved)	101	
(reserved)	110	
(reserved)	111	

3 Basic Design

As stated previously, it is possible to design various differing hardware based upon a single architecture. In this chapter, we will present the SIMPLE/B hardware, in which the SIMPLE CPU is divided into five phases, as a basic design example. It is worth noting that there are multiple places where the hardware presented can be improved, so it is greatly expected that students will make their own original design improvements after some thorough study without relying too heavily on the SIMPLE/B design.

As shown in Figure 2, SIMPLE/B is configured with five phases (p1 to p5) that are activated in order and an input-output device consisting of a controlling circuit that supplies a control signal for each phase, a main memory and a switch/LED/7SEG LED. The thick lines in the figure denote busses or selectors.

3.1 Controlling circuit

In SIMPLE/B, the signal detailed below is supplied externally.

1. clock
   System clock. Supplies the clock where Hi/Lo times are as close to 1:1 as possible by using an appropriate oscillation circuit. As shown in Figure 3, the controlling circuit activates each phase in order one by one in sync with the rising edge of the clock. Therefore, the period of the clock is fixed by the delay time of the phase where the delay time is the largest. In many cases, this is the value found by adding the delay time, etc. of the peripheral circuits to the memory access time.

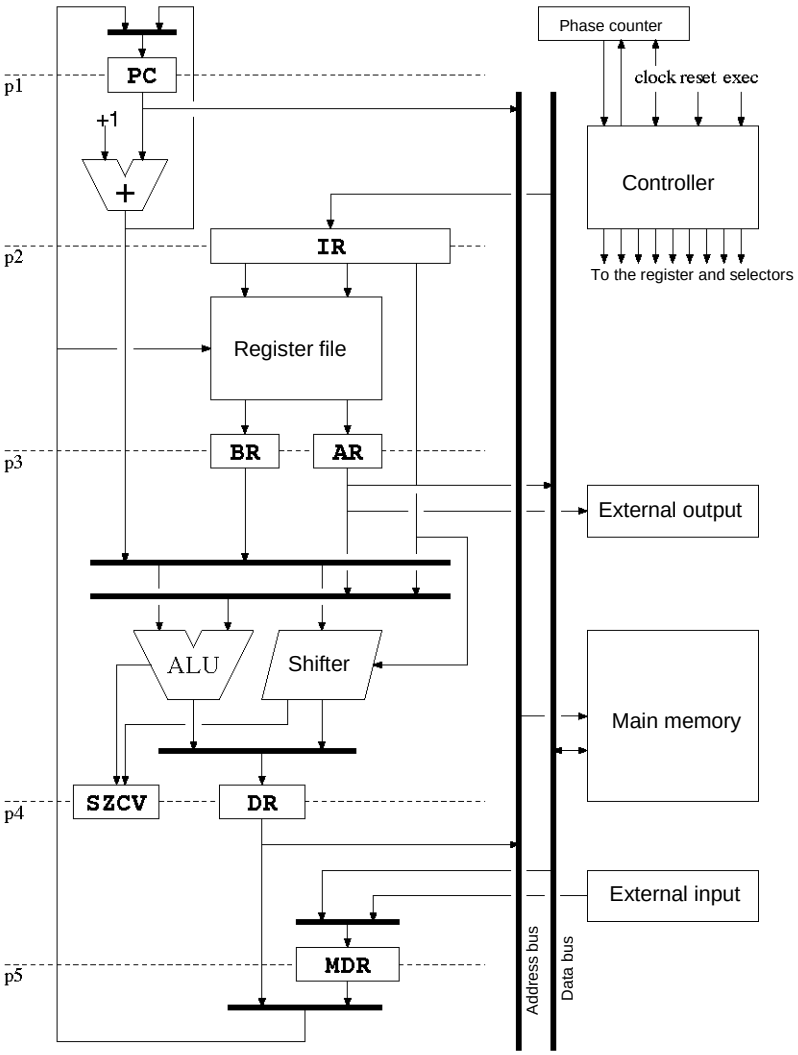


Figure 2: Block Diagram of SIMPLE/B

2. reset

   Reset signal. Uses a push switch to supply 1 when pressed and 0 when released[2]. It is desirable to have it so that it supplies 1 when powering up. When reset becomes 1, SIMPLE/B clears the PC, etc. and transitions to a suitable initial state.

3. exec

   Start/Stop signal. Uses a push switch to supply 1 when pressed and 0 when released[2]. When SIMPLE/B is in a stop state and exec changes from 0 to 1, SIMPLE/B will begin executing instructions. When SIMPLE/B is running and exec changes from 0 to 1, it will complete the instruction currently being executed and cease activity.

[2] [reset] and [exec] can both be made with negative logic.

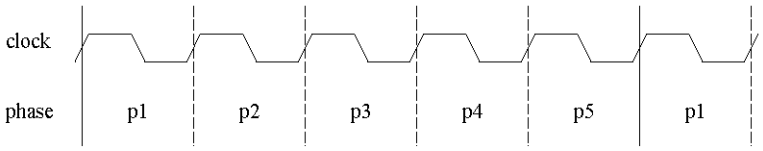


Figure 3: SIMPLE/B's Phases

3.2 Phases p1 to p5

SIMPLE/B executes a single instruction while activating five phases (p1 to p5) in sequence.

1. p1 (instruction fetch)

   Fetches the instruction of the address retained in the PC (program counter) from the main memory and stores it in the IR (instruction register), while adding 1 to the PC.

2. p2 (register readout)

   Reads the values of the general purpose register specified in the Ra/Rs field and Rb/Rd field of the instruction retained in the IR, storing them in registers AR and BR.

3. p3 (operation)

   The ALU or shift circuit performs the operation determined by the instruction (including address calculations and simple data transfer) and stores the result in register DR (data register). Condition codes S, Z, and C are set during operation instructions. There are times when AR and BR, as well as the PC and the instruction's immediate are used as the operation source.

4. p4 (main memory access)

   For load/store instructions, the value of DR is taken as an address and used to access the main memory. For load instructions, the read value is stored in register MDR (memory data register), and for store instructions the value retained by AR is written. Also, for input-output instructions, the value of the input switch is stored in MDR, and the value of AR is displayed to the 7SEG LED.

5. p5 (register writing)

   For instructions that require writing to a general purpose register, the value of DR or MDR is written to the general purpose register specified in the instruction's Ra, Rb, or Rs field. Also, for branching instructions, the value of DR is written to the PC as the branch destination address.

3.3 Main memory and input-output device

The capacity of the main memory will be made to be 64 KW, the same amount as SIMPLE's address space. Also, address conversions will not be performed; addresses determined by instructions will be used as main memory addresses without any changes being made. This main memory will be configured in the RAM loaded on the FPGA. However, due to the type of FPGA used in implementing SIMPLE, it is possible that size restrictions on the RAM loaded on the FPGA will render the main memory unable to hold 64 KW. (For a CycloneIV EP4CE30 on a PowerMedusa MU500-RX board, 33 KW is the maximum value.)

The RAM loaded onto the FPGA works in sync with a clock. Addresses for writing to and reading from the main memory are designated by selecting the value of the PC or DR and outputting it to the address bus. Data read from the main memory is output to the data bus

from the main memory if the clock signal is input. For this, IR or MDR can be brought in as necessary. Data to be written to the main memory is output from AR to the data bus. If the write enable signal is 1 when the clock signal is input, the data is written to the main memory.

Each student can choose their own preferred input-output interfaces that will deal with IN instructions and OUT instructions. The following input-output interfaces exist for the PowerMedusa MU500-RX/RK/7SEG boards.

- Input
  - Push switches: 20
  - Rotary switches: 2
  - 8-bit DIP switches: 2
- Output
  - LED: 8 + 64
  - 7SEG LED: 8 + 64
  - Buzzers: 1

These can be used not only for input-output instructions, but also to monitor and control the internal state of the processor as well. For the switches, on top of the aforementioned reset and exec signals, one should also assign a control signal to use for debugging. Display systems should show bus and register values as well as their value history. The use of rotary switches is also one way to change what is displayed.

4 Original Improvements and Extensions

The basic architecture of SIMPLE laid out in Chapter 2 was intentionally made to be an extremely simple way of constructing a functioning processor. On that note, this section will provide some hints for the assignment of improvements and extensions for that architecture.

4.1 Improving the instruction set architecture

This section contains some hints for how to go about extending the instruction set architecture below.

1. Enhancing the immediate operand
2. Enhancing input-output instructions
3. Adding “branch register” and “branch and link” instructions
4. Consolidating conditional branching into one instruction
5. Support for interrupts via board input
6. Adding complex operation instructions
7. Adding conditional operation instructions

The architecture shown here is merely an example. Students are free to make original modifications, and are even encouraged to do so. Also, when making large-scale extensions to the basic instruction set, it is possible to encounter a situation where there are not enough bits. In such cases, it is permitted for students to remove some instructions from the basic instruction set to incorporate these extensions.

4.1.1 Enhancing the immediate operand

In the basic architecture, the operands of operation instructions are all general purpose registers. For example, to add 1 to register r[0], it would require two instructions as shown below.

```
LI      R1,1
ADD     R0, R1
```

As such, one might consider an extension for the operation instructions in Table 1 (aside from the shifts that do not use the d field) that makes them operate on the immediate designated in register Rd and the d field.

Some hints are listed below.

- You will need to extend the instruction set while taking into account the conversion between r[Rd] + r[Rs] and r[Rd] + sign_ext(d). (perhaps using 1 bit within d as a flag)
- Of course, upon adding a functional unit, a possible means of extension would be to make it perform the operation r[Rd] + r[Rs] + sign_ext(d).

4.1.2 Enhancing input-output instructions

In the basic architecture, there are IN instructions in the form of input from the board as well as OUT instructions in the form of output to the board. In the basic architecture, the input and output target of these instructions is fixed to a specific 16-bit input/output. However, the PowerMedusa EC6S board has the input and output targets with 16 bits or more, and it is better for program creators to be able to select and use input/output among these targets. Therefore, one could consider an extension that makes use of the unused fields Rs and d in IN instructions and fields Rd and d in OUT instructions to change the input/output destination by using those fields.

Some hints are listed below.

- The 8-digit 7SEG LED can display data in data of 32 bits. You can use one bit in the field to switch between higher order and lower order for the 8-digit 7SEG LED.
- If all of the output of the 7SEG board is to be used, one will need at least a 5-bit field to select eight 8-digit 7SEG LEDs and 64 LEDs. To do so, one will need to use both the Rd and d fields.
- The register file can read two register values at the same time, so it might be interesting to extend it so that one can output two register values with a single OUT instruction.

4.1.3 Adding “branch register” and “branch and link” instructions

When a function is called, most compilers will output code that performs the following actions.

1. Stores the next PC after the instruction that called the function to the register and jumps to the beginning of the function.
2. Executes the function.
3. Writes the value stored to the register in Step 1 to the PC and begins execution from the next instruction following the one that called the function.

As seen above, the reason that the return address is stored to the register is that the line of code that calls the function might not be the only line that does so.

In order to support function calls, normal processors have what is known as branch and link (BAL) instructions to carry out the purpose of Step 1, and branch register (BR) instructions to carry out the purpose of Step 3. If one were to extend the architecture to implement these instructions, they would be able to implement functions that can be called from any location. These are instructions that must be implemented for practical programming on the processor.

Some hints are listed below.

- A new data path will be necessary to save the PC to the register.
- There are ways to implement this in which the register to which the BAL saves the PC as well as the register that the BR reads from can be freely specified. Conversely, there are implementations of this where these registers are fixed. For what it’s worth, the MIPS architecture restricts the register to which the BAL equivalent saves the PC to register 31. It can be an interesting exercise to consider the pros and cons of fixing the register in this manner.
- When writing programs in which the called function calls yet another function, it is necessary to save the content of the resister storing the return location from the current function to the main memory, etc. before calling the second function. Other register values will also need to be saved to the main memory, etc. as necessary. (For details, refer to compiler experiments.)
- For situations like the aforementioned layered function calls, it might be interesting to consider ways to speed up the function calls/returns in cases where they are not in such deep layers. For example, one might consider decreasing the amount needing to be saved to the main memory by adding the stack that saves the return addresses and the registers to the processor.

4.1.4 Consolidating conditional branching into one instruction

In the basic architecture, conditional branching makes use of the condition codes set by the result of operations. With this configuration, it is possible to expand the potential scope of the branching by enlarging the d field of the instruction’s bit stream shown in Table 4. However, with this configuration, one fundamentally needs to execute a comparison instruction before carrying out the conditional branch. This makes one consider the potential

of performing the comparison and branching in a single instruction. Naturally, one must provide space in the instruction's bit stream for the part that specifies the two registers being compared. As such, this decreases the number of bits available to be used for the d field. As a result, the potential scope of the branching is decreased, but this is not a problem if one leaves the conditional branching instruction from the basic architecture.

Some hints are listed below.

- As seen in the format in Table 4, there are not enough bits to specify two registers. One can make use of the "(reserved)" sections found in Table 3.
- The basic architecture is a proper architecture with a lot of deep thought and planning. If one writes programs in a certain manner, it is possible to implement conditional branching without the use of comparison instructions. It might be interesting to consider things along that line of thought.
- Table 4 also contains "(reserved)" sections. One can use these to increase the branching conditions for conditional branching instructions.

4.1.5 Support for interrupts via board input

The basic architecture does not have support for processing interrupts from board input. It cannot do anything like "executing a specific routine when a certain button on the board is pressed"[3]. As such, one might consider a potential extension that enables interrupting according to board input. The board input can be used to trigger specific tasks to be processed.

Fundamentally, when interrupting occurs, it follows the pattern of: "suspension of current processing" → "execution of interruption processing" → "resuming of previous processing."

1. Completes the instruction currently being executed, updates the register values and condition codes, and decides the PC of the next instruction to be executed.
2. Saves the condition codes, PC value, and all of the values in the register file to the memory.
3. Feeds the first memory address of the instruction line of the interruption processing to the PC.
4. Reads the instructions according to the PC and performs the interruption processing.
5. Writes back the register values, condition codes, and PC value saved in Step 2 once the interruption processing is finished.
6. Reads instructions according to the PC value, continuing the execution of the processing that was occurring in Step 1.

Some hints are listed below.

- Fundamentally, the call/return processing required for interrupting can be thought of as a superset of the function call/return processing. Thus, one should first implement BAL and BR instructions.

[3] It is not necessarily impossible to implement this with IN instructions only, but these instructions can only read values in the moment they are executed, so the IN instruction would need to be executed while the button is being pressed. Consequently, one would need step-by-step execution of these IN instructions or a loop that continuously monitors input, so they would be exceedingly difficult to use.

- The processing for saving/returning of the registers, condition codes, and PC can be implemented through the hardware, but one can simplify the hardware by simply implementing this as a part of the interruption processing.
- The first memory address of the instruction line of the interruption processing can be fixed, but if one makes it so that it can be specified in a register, then one will be able to switch between a few processes for interrupts.
  - The register that specifies this address can be part of a general purpose register, or one can arrange for a dedicated register.
  - If one goes for the dedicated register route, then it will be necessary to have dedicated instructions to operate this register.
- Implementation would be easier with a PC, register file, and condition codes to use specifically for interrupts along with the ability to switch back and forth upon start/finish of interruption processing. However, this would certainly increase hardware costs.
- It would be good to be able to support multiple interrupt signals instead of just one. Of course, in this case, one would need to prepare multiple interruption processes and have the ability to select between them based on the signal.
- It would be good to also implement instructions that initiate interruptions. This can double as a way to test the ability to process interrupts.
- Most normal processor designs treat interrupt processing instructions in the same manner as input-output instructions; they are considered privileged instructions and cannot be executed by normal programs. However, considering the relatively small intended scope of SIMPLE, there is not a need to go out of one's way to implement such functionality.
- Implementing timer interrupts would be interesting as it would allow for one to write programs that process in real time. However, the PowerMedusa EC6S board does not have a timer, so one can assume a fixed clock frequency and initiate timer interrupts from the clock number.

4.1.6 Adding complex instructions

Many processors that specialize in processing sounds and images (digital signal processors, etc.) are equipped with instructions that can perform multiple operations with a single instruction. This enables them to process sounds and images more efficiently. An example of such an instruction is the multiply accumulate (MAC:Multiply ACcumulate) instruction below.

r[c] = r[c] + r[a] * r[b]

When programs were written in assembly, in cases where the instruction lines below appeared frequently, it would be interesting to have an extension that introduces an instruction that accomplishes the same behavior as r[c] = r[c] op2 (r[a] op1 r[b]). (op1 and op2 denote operations.)

op1 r[a], r[b]
op2 r[c], r[a]

Some hints are listed below.

- The number of register values that will have to be read upon execution will increase, so one will either need to increase the number of read ports for register files or perform the readings across multiple cycles.
- One can come up with configurations that use multiple functional units and that utilize multi-cycle execution to use only a single functional unit.
- The implementation of intricate complex instructions can cause the frequency of the system clock to sharply decrease due to the circuits becoming more complicated. Also, in many cases one is unable to attain performance improvements that counterbalance the extensions made to the hardware. Thoroughly considering these factors can be very interesting.
- If time allows, one can try implementing both types of configurations; implementing multiple functional units and performing multi-cycle execution and then compare the two.

4.1.7 Adding conditional operation instructions

Certain processors exist that are equipped with conditional execution instructions whose execution is contingent on the status of some condition code. Such instructions are useful for cutting down on pipeline bubbles caused by branching in pipelined processors (omitting a detailed explanation for the sake of brevity). SIMPLE/B is not pipelined, so there is no particular significance in adding conditionally executed instructions. However, based on the same line of thinking, one might consider implementing instructions in which the operation performed changes based upon the status of a condition code.

To help explain this, let us examine an example. Suppose that a program contains an if-else statement in which the if portion adds variables A and B, and the else portion subtracts A and B. If instructions are implemented that will perform addition if the condition code is the same as the if portion's condition and subtract when it is not, then this drastically reduces the number of instructions required to express this if-else statement. If we have programs like the one detailed above, then adding these conditional operation instructions becomes extremely intriguing.

Some hints are listed below.

- These instructions will be even more limited in their potential uses than the complex instructions described in Section 4.1.6. One should probably brainstorm other programs that they could be useful for so that one does not implement instructions that can only be used in one program.
- In their report, students should demonstrate the application examples of these instructions for multiple programs and explain their usefulness.
- It would also be intriguing to discuss in one's report just how much one can benefit from pipelined processors that have conditional execution.

4.2 Improving the microarchitecture

4.2.1 Parallel execution of phases

One possible way to improve upon SIMPLE/B would be to shorten the instruction cycle by executing phases in parallel.

1. p1/p5 parallel execution (Fig. 4 (a))

The instruction cycle becomes a 4-step cycle. This can be implemented quite easily with a little bit of tinkering with the process that handles setting the branch destination address in the PC.

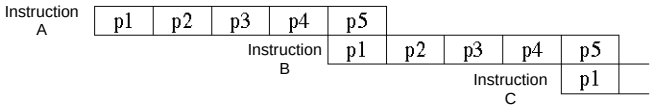
2. p1/p3 and p2/p5 parallel execution (Fig. 4 (b))
   The instruction cycle becomes a 3-step cycle.

   It takes a lot of tinkering with the branch instruction execution to be able to always execute p1/p3 in parallel (hint: do not use p3 for the branch address calculation). As an alternative method, one might consider simply not using parallel execution for branching instructions.

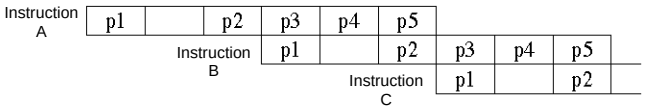
   In order to execute p2/p5 in parallel, one must take steps to deal with situations in which an instruction that references a general purpose register is encountered immediately after an instruction that updates the same register.

3. p1/p3/p5 and p2/p4 parallel execution (Fig. 4 (c))
   The instruction cycle becomes a 2-step cycle. One will need to do some further tinkering to the data reliance solution devised in Step 2.

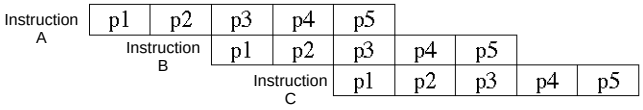
4. p1/p2/p3/p4/p5 parallel execution (Fig. 4 (d))
   The instruction cycle becomes a 1-step cycle, referred to as pipelining. Implementing this would be very difficult, but it is worth giving it a shot due to the fact that many processors used these days are pipelined.



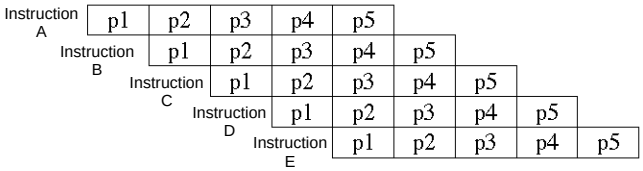
(a) p1/p5 parallel execution



(b) p1/p3 and p2/p5 parallel execution



(c) p1/p3/p5 and p2/p4 parallel execution



(d) p1/p2/p3/p4/p5 parallel execution

Figure 4: Parallel Execution of Phases

4.2.2 Parallel execution of instructions (superscalar execution)

Current processors for PCs detect sets of instructions that can be executed in parallel among instructions in the program and execute them in parallel. This is called superscalar

execution, and processors that can do this are called superscalar processors. Current processors for PCs combine superscalar execution and out-of-order execution (execution that makes it possible to execute instructions in order different from the program order), but implementing this configuration would naturally require hardware of a much larger scale. As such, if one wishes to implement superscalar execution, it would be better to combine it with in-order execution (in which instructions are only executed in the program order). This combination of superscalar execution and in-order execution is even used in Intel's initial Pentium series.

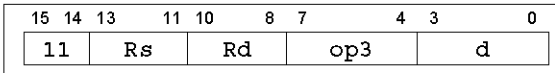
When combining superscalar execution and in-order execution, the processor reads consecutive instructions on the program all at once from the main memory and executes them in parallel so long as the instructions have no data interdependencies (in which the execution result of an instruction is used by a different instruction). The probability of three or more consecutive instructions on the program having no data interdependencies is quite low, thus most processors that combine in-order execution with superscalar execution limit the number of instructions executed in parallel to two. Some hints regarding two-instruction parallel execution are shared below.

- Reading the main memory for multiple instructions generally is performed by making the unit of reads from the main memory larger than the bit length of the instructions and then extracting the individual instructions from the read data. However, if one uses multi-port RAM to add support for parallel execution of load instructions (discussed later), multiple instruction addresses can be sent to the RAM to read multiple instructions.
- Whether or not parallel execution of multiple instructions is possible depends on p2 or equivalent. Specifically, so long as the first instruction's Rd (Ra, Rb) and the second instruction's Rb, Rs, Rb, Ra do not coincide, parallel execution is possible. If it is determined at this point that parallel execution is not possible, only the first instruction will be sent to p3. As for the second instruction, one might consider an approach where it is discarded and then re-read by the main memory or where it is stored in the registers and only one instruction is read at the next reading of instructions.
- In order to execute two instructions in parallel, the register file will have a 4 read/2 write configuration.
- When executing two instructions in parallel, the condition code will be updated in p3 by the second instruction.
- In order to perform p4's main memory access in parallel, the main memory should be composed of multi-port RAM. However, the FPGA used in this experiment is prepared with only up to 2 read/1 write RAM, so it cannot execute stores in parallel. One can include the judgment as to whether two instructions to be executed in parallel are both store instructions in the parallel execution judgment made in p2.

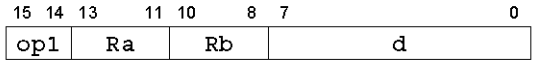
5 Related Materials

- Experiment webpage: http://www.lab3.kuis.kyoto-u.ac.jp/~takase/le3a/This page contains detailed materials on the experiments. Also, various announcements are posted there.

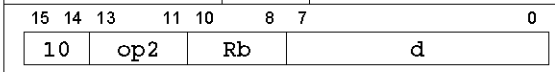
Appendix A: Instruction Set for the Basic Architecture



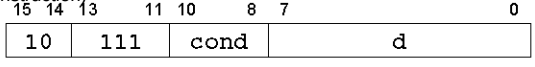
mnemonic	op3	function
ADD Rd,Rs	0000	r[Rd] = r[Rd] + r[Rs]
SUB Rd,Rs	0001	r[Rd] = r[Rd] - r[Rs]
AND Rd,Rs	0010	r[Rd] = r[Rd] & r[Rs]
OR Rd,Rs	0011	r[Rd] = r[Rd] | r[Rs]
XOR Rd,Rs	0100	r[Rd] = r[Rd] ^ r[Rs]
CMP Rd,Rs	0101	r[Rd] - r[Rs]
MOV Rd,Rs	0110	r[Rd] = r[Rs]
(reserved)	0111	
SLL Rd,d	1000	r[Rd] = shift_left_logical(r[Rd],d)
SLR Rd,d	1001	r[Rd] = shift_left_rotate(r[Rd],d)
SRL Rd,d	1010	r[Rd] = shift_right_logical(r[Rd],d)
SRA Rd,d	1011	r[Rd] = shift_right_arithmetic(r[Rd],d)
IN Rd,d	1100	r[Rd] = input
OUT Rs	1101	output = r[Rs]
(reserved)	1110	
HLT	1111	halt()



mnemonic	op1	function
LD Ra,d(Rb)	00	r[Ra] = *(r[Rb] + sign_ext(d))
ST Ra,d(Rb)	01	*(r[Rb] + sign_ext(d)) = r[Ra]



mnemonic	op2	function
LI Rb,d	000	r[Rb] = sign_ext(d)
(reserved)	001	
(reserved)	010	
(reserved)	011	
B d	100	PC = PC + 1 + sign_ext(d)
(reserved)	101	
(reserved)	110	
(Conditional branch instruction)	111	



mnemonic	cond	function
BE d	000	if (Z) PC = PC + 1 + sign_ext(d)
BLT d	001	if (S ^ V) PC = PC + 1 + sign_ext(d)
BLE d	010	if (Z || (S ^ V)) PC = PC + 1 + sign_ext(d)
BNE d	011	if (!Z) PC = PC + 1 + sign_ext(d)
(reserved)	100	
(reserved)	101	
(reserved)	110	
(reserved)	111	