1   Why Pipelining?

The datapath design that we implemented for Project 1 was, in fact, grossly inefficient. By focusing on increasing throughput, a pipelined processor can get more instructions done per clock cycle. In the real world, that means higher performance, lower power draw, and most importantly, happy customers!

2   Project Requirements

In this project, you will make a pipelined processor that implements the Conte-200 ISA. There will be five stages in your pipeline:

1. **IF** - Instruction Fetch
2. **ID/RR** - Instruction Decode/Register Read
3. **EX** - Execute (ALU operations)
4. **MEM** - Memory (both reads and writes with memory)
5. **WB** - Writeback (writing to registers)

Before you move on, read Appendix A: Conte-200 Instruction Set Architecture to understand the ISA that you will be implementing. We provide you with a Brandonsim file with the some of the structure laid out.

3   Building the Pipeline

First, you will have to build the hardware to support all of your instructions. You will have to make each stage such that it can accommodate the actions of all instructions passing through it. Use the book (Ch. 5) to get an idea of what the pipeline looks like and to understand the function of each stage before you start building your circuits.

1. IF Stage

The IF stage is responsible for:

- Getting the instruction from I-MEM at location PC
- Updating the PC

For normal sequential execution, we would update the PC by incrementing it by 1. Notice, however, that this may not be the case when executing a SKP, CALL, RET, or GOTO instruction. Hence, you will likely need to multiplex which value is used to update the PC.

2. ID/RR Stage

The ID/RR stage is responsible for:

- Decoding the instruction
- Reading the appropriate registers
- Resolving any CALL, RET, SKP, or GOTO instructions

Please look at Appendix A: Conte-200 Instruction Set Architecture in order to understand the instruction formats! You will have a dual ported register file (DPRF), which allows you to read from two registers and write one register all at the same time. As you will notice, the TAs have been very kind in making the DPRF and providing it to you.

Some of the instructions require both inputs into to the ALU to be values pulled from the DPRF. However, other instructions contain a value within the instruction, such as an immval20, offset20, or PCAddr24 field. You may either pass all of these possible values to the next stage (requires bigger buffer registers), or condense them into just the values needed to execute the instruction in the following cycles (requires more logic, but buffer size can be optimized).

3. EX Stage

The EX stage is responsible for:

- Performing all necessary arithmetic and logic calculations

In the Execute (EX) stage, you will perform any arithmetic computations required by the instruction. This stage should host a complete ALU to perform the actual adding or NANDing as required by the instruction. For memory access instructions, this stage will perform the Base + Offset computation required to determine the memory address to access.

4. MEM Stage

The MEM stage is responsible for:

- Reading from or writing a result to memory

All you need to do is to use the value calculated in the EX stage as the address for the RAM. **Note that you must use the maximum address length for the RAM block - this is 24 bits.** To accomplish this, simply take the lower 24 bits of the calculated address. Depending on the instruction, this stage will need to pass either the value read from memory or the value computed in EX to the WB stage.

5. WB Stage

The WB stage is responsible for:

- Writing results back to the DPRF (dual-ported register file)

Depending on the instruction, you may need to write a value back to a register. To do this, your WB stage will attach to the data in and write enable inputs of the DPRF in ID/RR. Remember that the DPRF can write **and** read different registers **in the same clock cycle**, which is why WB and ID/RR can share the same register file. For instructions that do not write a register, your WB stage may not do anything at all.

4 General Advice

Subcircuits

For this project, we highly encourage using modular design and creating subcircuits when necessary. We **strongly** recommend using subcircuits when building your pipeline buffers as well as your forwarding unit.

Pipeline Buffers

For deciding what to pass through buffers, remember that we need to support the requirements of every possible instruction. Think of what each instruction needs to fulfill its duty, and pass a union of all those requirements. (By union we mean the mathematical union, for example say I1 needs PC and Rx, while I2 needs Rx and Ry, then you should pass PC, Rx and Ry through the buffer). You can also feel free to implement your hardware such that you re-use space in the buffer for different purposes depending on the instruction, but this is not required.

Control Signals

In the Project 1 datapath, recall that we had one main ROM that was the single source of all the control signals on the datapath. Now that we are spreading out our work across different stages of the pipeline, you have a choice of how to implement your signals!

There are two options:

1. You can either have a single large main ROM in ID/RR which calculates all the control signals for every stage.

   OR

2. you can have a small(er) ROM in each stage which takes in the opcode and assert the proper signals for that operation.

Note that if you choose the first method, you will need to pass all the signals needed for later stages through the earlier stages, and in the second method, you will need to pass the instruction opcode though all the stages so that you know which signals to assert during that stage.

Stalling the Pipeline

One must stall the pipeline when an instruction cannot proceed to the next stage because a value is not yet available to an instruction. This usually happens because of a data hazard. For example, consider two instructions in the following program:

1. `LW $t0, 5($t1)`

2. `ADDI $t0, $t0, 1`

Without stalling the ADDI instruction in the ID/RR stage, it will get an out of date value for $t0 from the regfile, as the correct value for $t0 isn't known the LW reaches the MEM stage! Therefore, we must stall. Consult the textbook (or your notes) for more information on data hazards. It is also important to note that through data forwarding, stalls can be lessened in penalty or in some cases avoided entirely. Data forwarding is discussed in the next section

To stall the pipeline, the stages preceding the stalled stage should disable writes into their buffers, i.e. they should continue to output the previous value into the next stage. The stalled stage itself will output NOOP (example, ADD $zero, $zero, $zero) instructions down the pipeline until the cause of the stall finishes.

Data Forwarding

If you really liked the busy-bit/read-pending signal forwarding described in lecture and in your book, feel free to use that. We present an alternate way to do forwarding in this section.

Forwarding is one way to increase the performance of the pipeline. This allows us to get values computed in stages beyond ID/RR back to ID/RR so that we do not have to stall the instruction. I would strongly recommend against using the busy bit/read pending bit strategy suggested in the book - this has some very nasty edge cases and requires much more logic than necessary.

I would recommend that you make a forwarding unit that implements various stock rules. The forwarding unit should take in the two register values you are reading, the output value from the EX stage, the output value from the MEM stage, and the output value from the WB stage. To forward a value from a future stage back to ID/RR, you must check to see if the destination register number from a particular stage is equal to your source register numbers in the ID/RR stage. If so, you must forward the value from that stage to your ID/RR stage.

You shouldn't update the value of the register when you forward the value back - writes to the register file should only occur in the WB stage. Of course, forwarding cannot save you from one situation: when the destination register of a LW instruction is the source register of an instruction immediately after it. In this case, you must stall the instruction in the ID/RR stage. I will leave it to you to flesh out all of the stall rules.

Keep in mind: the zero register can never change, therefore it should not be considered for forwarding and stalling situations.

Flushing the Pipeline

For the CALL/RET/SKP/GOTO instructions, we calculate the target in the ID/RR stage of the pipeline. However, the next instruction the IF stage fetches while ID/RR is computing the target may not be the next instruction we want to execute. When this happens, we must have a hardware mechanism to "cancel" or "flush" the incorrectly-fetched instructions after we realize they are incorrect.

In implementing your flushing mechanism, we **highly recommend** avoiding the asynchronous clear feature of registers in Brandonsim, as this may cause timing issues. Instead, we suggest using a multiplexer to selectively send a NOOP into the buffer input.

Skip Prediction

When you encounter a SKP instruction, you should predict that the SKP is not taken. This means there should be no stalling, Fetch should simply go on and retrieve the next instruction at PC + 1.

Upon resolving the branch, the pipeline should continue normally in the case of a correct prediction, or flush the instruction following the SKP in the case of an incorrect prediction.

5 Testing

When you have constructed your pipeline, you should test it instruction by instruction to see if you have all the necessary components to ensure proper execution.

Be careful to only use the instructions listed in the appendix - there are some subtle points in having a separate instruction and data memory. Load the assembled program into both the instruction memory and the data memory and let your processor execute it. Any writes to memory will only affect the data memory.

6 Deliverables

Please submit all of the following files in a **.tar.gz** archive. You must turn in:

- Brandonsim Datapath File (Conte-200-pipeline.circ)

If you are running on a Linux or Unix-based machine, run `make submit` to automatically package your project for submission.

Always re-download your assignment from T-Square after submitting to ensure that all necessary files were properly uploaded. If what we download does not work, you will get a 0 regardless of what is on your machine.

This project will be demoed. In order to receive full credit, you must sign up for a demo slot and complete the demo. We will announce when demo times are released.

7   Appendix A: Conte-200 Instruction Set Architecture

The Conte-200 is a simple, yet capable computer architecture.

The Conte-200 is a **word-addressable**, **32-bit** computer. **All addresses refer to words**, i.e. the first word (four bytes) in memory occupies address 0x0, the second word, 0x1, etc.

All memory addresses are truncated to 24 bits on access, discarding the 8 most significant bits if the address was stored in a 32-bit register. This provides roughly 67 MB of addressable memory.

7.1   Registers

The Conte-200 has 16 general-purpose registers. While there are no hardware-enforced restraints on the uses of these registers, your code is expected to follow the conventions outlined below.

Table 1: Registers and their Uses

Register Number	Name	Use	Callee Save?
0	$zero	Always Zero	NA
1	$at	Reserved for the Assembler	NA
2	$v0	Return Value	No
3	$a0	Argument 1	No
4	$a1	Argument 2	No
5	$a2	Argument 3	No
6	$t0	Temporary Variable	No
7	$t1	Temporary Variable	No
8	$t2	Temporary Variable	No
9	$s0	Saved Register	Yes
10	$s1	Saved Register	Yes
11	$s2	Saved Register	Yes
12	$k0	Reserved for OS and Traps	NA
13	$sp	Stack Pointer	No
14	$fp	Frame Pointer	Yes
15	$ra	Return Address	No

1. **Register 0** is always read as zero. Any values written to it are discarded. **Note:** for the purposes of this project, you must implement the zero register. Regardless of what is written to this register, it should always output zero.

2. **Register 1** is a general purpose register. You should not use it because the assembler will use it in processing pseudo-instructions.

3. **Register 2** is where you should store any returned value from a subroutine call.

4. **Registers 3 - 5** are used to store function/subroutine arguments. **Note:** registers 2 through 8 should be placed on the stack if the caller wants to retain those values. These registers are fair game for the callee (subroutine) to trash.

5. **Registers 6 - 8** are designated for temporary variables. The caller must save these registers if they want these values to be retained.

6. **Registers 9 - 11** are saved registers. The caller may assume that these registers are never tampered with by the subroutine. If the subroutine needs these registers, then it should place them on the stack and restore them before they jump back to the caller.

7. **Register 12** is reserved for handling interrupts. While it should be implemented, it otherwise will not have any special use on this assignment.

8. **Register 13** is your anchor on the stack. It keeps track of the top of the activation record for a subroutine.

9. **Register 14** is used to point to the first address on the activation record for the currently executing process. Don't worry about using this register.

10. **Register 15** is used to store the address a subroutine should return to when it is finished executing. It is automatically used for this purpose by the CALL and RET instructions.

7.2 Instruction Overview

The Conte-200 supports a variety of instruction forms, only a few of which we will use for this project. The instructions we will implement in this project are summarized below.

Table 2: Conte-200 Instruction Set

	31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4	3 2 1 0
ADD	0000	DR	SR1	unused	SR2
ADDI	0001	DR	SR1	immval20	
NAND	0010	DR	SR1	unused	SR2
SKP	0011	mode	SR1	unused	SR2
GOTO	0100	0000	PCaddr24		
LEA	0101	DR	PCaddr24		
LW	1000	DR	BaseR	offset20	
SW	1001	SR	BaseR	offset20	
CALL	1100	TR	unused		
RET	1101	unused			
HALT	1111	unused			

7.2.1 Conditional Branching

Conditional branching in the Conte-200 ISA is provided via two instructions: the SKP ("skip") instruction and the GOTO ("unconditional branch") instruction.

The SKP instruction compares two registers and skips the immediately following instruction if the comparison evaluates to true. If the action to be conditionally executed is only a single instruction, it can be placed immediately following the SKP instruction. Otherwise a GOTO can be placed following the SKP instruction to branch over to a longer sequence of instructions to be conditionally executed.

7.3 Detailed Instruction Reference

7.3.1 ADD

Assembler Syntax

```
ADD    DR, SR1, SR2
```

Encoding

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4	3 2 1 0
0000	DR	SR1	unused	SR2

Operation

```
DR = SR1 + SR2;
```

Description

The ADD instruction obtains the first source operand from the SR1 register. The second source operand is obtained from the SR2 register. The second operand is added to the first source operand, and the result is stored in DR.

7.3.2 ADDI

Assembler Syntax

```
ADDI   DR, SR1, immval20
```

Encoding

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0001	DR	SR1	immval20

Operation

```
DR = SR1 + SEXT(immval20);
```

Description

The ADDI instruction obtains the first source operand from the SR1 register. The second source operand is obtained by sign-extending the immval20 field to 32 bits. The resulting operand is added to the first source operand, and the result is stored in DR.

7.3.3 NAND

Assembler Syntax

```
NAND   DR, SR1, SR2
```

Encoding

<div>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</div>

0010	DR	SR1	unused	SR2

Operation

```
DR = ~(SR1 & SR2);
```

Description

The NAND instruction performs a logical NAND (AND NOT) on the source operands obtained from SR1 and SR2. The result is stored in DR.

> **HINT:** A logical NOT can be achieved by performing a NAND with both source operands the same. For instance,
>
> ```
> NAND DR, SR1, SR1
> ```
>
> ...achieves the following logical operation: $DR \leftarrow \overline{SR1}$.

7.3.4 SKP

Assembler Syntax

```
SKPNE   SR1, SR2
SKPLE   SR1, SR2
```

Encoding

<div>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</div>

0011	mode	SR1	unused	SR2

`mode` is defined to be `0x0` for SKPNE, and `0x1` for SKPLE.

Operation

```
if (MODE == 0x0) {
    if (SR1 != SR2) PC = PC + 1;
} else if (MODE == 0x1) {
    if (SR1 <= SR2) PC = PC + 1;
}
```

Description

The SKP instruction compares the source operands SR1 and SR2 according to the rule specified by the mode field. For mode `0x0`, the comparison succeeds if SR1 does NOT equal SR2. For mode `0x1`, the comparison succeeds if SR1 is less than or equal to SR2.

If the comparison succeeds, the incremented PC (address of instruction + 1) is incremented again, for a resulting PC of (address of instruction + 2). **This effectively "skips" the immediately following instruction.** If the comparison fails, the program continues execution as normal.

7.3.5   GOTO

Assembler Syntax

```
GOTO    LABEL
```

Encoding

<small>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</small>

0100	0000	PCaddr24

Operation

```
PC = ZEXT(PCaddr24);
```

Description

The program unconditionally branches to the location specified by the zero-extended bits [23:0]. **This instruction is not PC-relative. It goes exactly to the address specified in the PCaddr24 field.**

7.3.6   LEA

Assembler Syntax

```
LEA    DR, label
```

Encoding

<small>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</small>

0101	DR	PCaddr24

Operation

```
DR = ZEXT(PCaddr24);
```

Description

An address is computed by zero-extending bits [23:0] to 32 bits and storing this result in DR. This instruction effectively performs the same computation as the GOTO instruction, but rather than performing an unconditional branch, merely stores the computed address into register DR.

7.3.7   LW

Assembler Syntax

```
LW   DR, offset20(BaseR)
```

Encoding

<small>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</small>

1000	DR	BaseR	offset20

Operation

```
DR = MEM[BaseR + SEXT(offset20)];
```

Description

An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word at this address is loaded into DR.

7.3.8   SW

Assembler Syntax

```
SW    SR, offset20(BaseR)
```

Encoding

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1001	SR	BaseR	offset20

Operation

```
MEM[BaseR + SEXT(offset20)] = SR;
```

Description

An address is computed by sign-extending bits [19:0] to 32 bits and then adding this result to the contents of the register specified by bits [23:20]. The 32-bit word obtained from register SR is then stored at this address.

7.3.9   CALL

Assembler Syntax

```
CALL   TR
```

Encoding

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

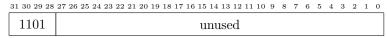| 1100	TR	unused

Operation

```
$ra = PC;
PC = TR;
```

Description   First, the incremented PC (address of the instruction + 1) is stored into the $ra register. Next, the PC is loaded with the value of register TR, and the computer resumes execution at the new PC.

7.3.10   RET

Assembler Syntax

```
RET
```

Encoding

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1101	unused

Operation

```
PC = $ra;
```
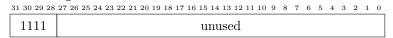
Description

The PC is loaded with the value of the $ra register, and the computer resumes execution at the new PC.

7.3.11   HALT

Assembler Syntax

HALT

Encoding

31 30 29 28	27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1111	unused

Description

The machine is brought to a halt and executes no further instructions.