

## 1 Requirements

- Download the proper version of Brandonsim. A copy of Brandonsim is available under Resources on T-Square. In order to run Brandonsim, Java must be installed. If you are a Mac user, you may need to right-click on the JAR file and select “Open” in the menu to bypass Gatekeeper restrictions.
- Brandonsim is not perfect and does have small bugs. In certain scenarios, files have been corrupted and students have had to re-do the entire project. Please back up your work using some form of version control, such as a local git repository. **Do not use public git repositories, it is against the Georgia Tech Honor Code.**
- The Conte-200 assembler is written in Python. If you do not have Python 2.6 or newer installed on your system, you will need to install it before you continue.

## 2 Project Overview and Description

Project 1 is designed to give you a good feel for exactly how a processor works. In Phase 1, you will design a datapath in Brandonsim to implement a supplied instruction set architecture. You will use the datapath as a tool to determine the control signals needed to execute each instruction. In Phases 2 and 3 you are required to build a simple finite state machine (the “control unit”) to control your computer and actually run programs on it.

**Note:** You will need to have a working knowledge of Brandonsim. Make sure that you know how to make basic circuits as well as subcircuits before proceeding. The TAs are always here if you need help.

### 3 Phase 1 - Implement the Datapath

#### Conte-200 Datapath

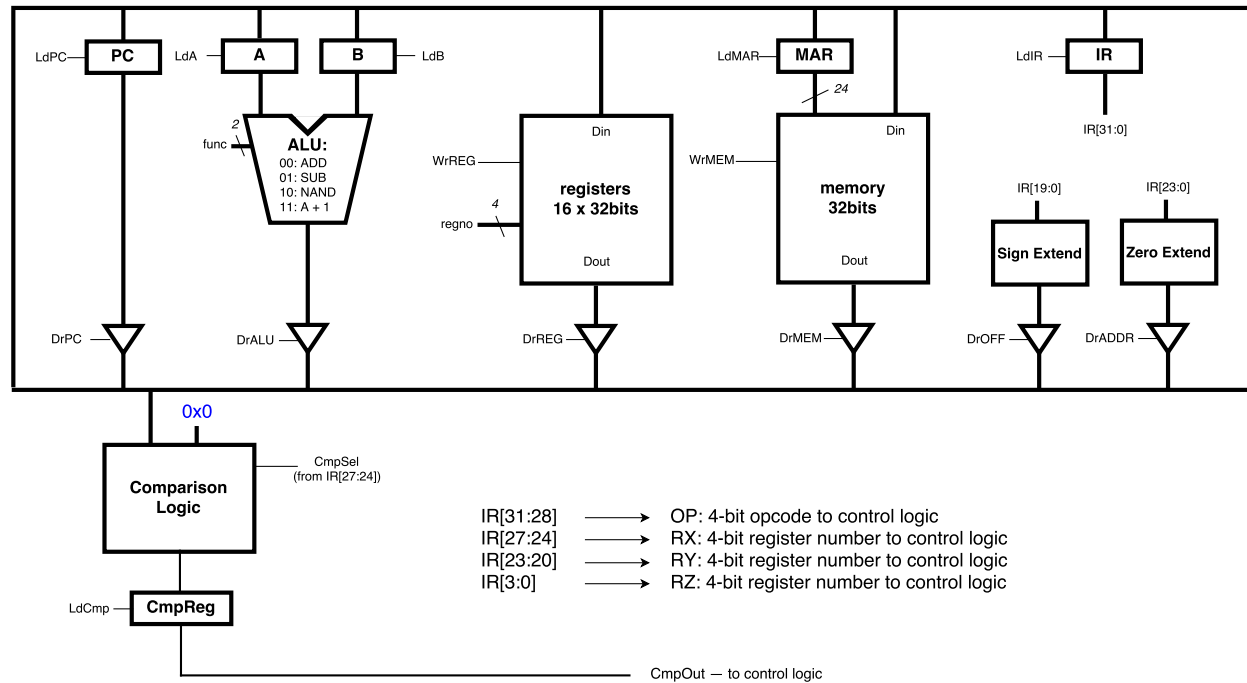


Figure 1: Datapath for the Conte-200 Processor

In this phase of the project, you must learn the Instruction Set Architecture (ISA) for the processor we will be implementing. Afterwards, we will implement a complete Conte-200 datapath in Brandonsim using what you have just learned.

#### You must do the following:

1. Learn and understand the Conte-200 ISA. The ISA is fully specified and defined in Appendix A: Conte-200 Instruction Set Architecture. **Do not move on until you have fully read and understood the ISA specification.** *Every single detail* will be relevant to implementing your datapath in the next step.
2. Using Brandonsim, implement the Conte-200 datapath. To do this, you will need to use the details of the Conte-200 datapath defined in Appendix A: Conte-200 Instruction Set Architecture. You should model your datapath on Figure 1.

### 3.1 Hints

#### 3.1.1 Subcircuits

Brandonsim enables you to break create reusable components in the form of subcircuits. **We highly recommend that you break your design up into subcircuits.** At a minimum, you will want to implement

your ALU in a subcircuit. The control unit you implement in Phase 2 is another prime candidate for a subcircuit.

### 3.1.2 Debugging

As you build the datapath, you should consider adding functionality that will allow you to operate the whole datapath by hand. This will make testing individual operations quite simple. We suggest your datapath include devices that will allow you to put arbitrary values on the bus and to view the current value of the bus. Feel free to add any additional hardware that will help you understand what is going on.

### 3.1.3 Memory Addresses

Because of Brandonsim limitations, the RAM module is limited to no more than 24 address bits. Therefore, per our ISA, any 32-bit values used as memory addresses will be truncated to 24 bits (with the 8 most significant bits disregarded). We recommend you implement this (i.e. truncate the MSB bits) before feeding the address value from the MAR (Memory Address Register) to the RAM.

### 3.1.4 Comparison Logic

The “comparison logic” box in Figure 1 is responsible for performing the comparison logic associated with the SKP instruction. The comparison logic should read the current value on the bus plus the mode bits from the IR. When executing SKP, you should compute  $A - B$  using the ALU. While this result of the ALU is being driven on the bus, the comparison logic should read the result  $A - B$  and output a single “true” or “false” bit for either the condition  $A \neq B$  or  $A \leq B$  depending on the mode input.

Your comparison logic should be purely combinational. Feel free to use any Brandonsim components you wish to aid in your implementation.

### 3.1.5 Zero Register

Your zero register must be implemented such that writes to it are ineffective, i.e., attempting to write a non-zero value to the zero register will do nothing. **Do not forget to do this or you will lose points!**

### 3.1.6 Register Select

From lecture and the textbook, you should be familiar with the “register select” (RegSel) multiplexer. The mux is responsible for feeding the register number from the correct field in the instruction into the register file. In order to implement the CALL/RET instructions, you will need to add another input to the mux that is always wired to the register number for the \$ra register. See Table 4 for a list of inputs your mux should have.

## 4 Phase 2 - Implement the Microcontrol Unit

In this phase of the project, you will use Brandonsim to implement the microcontrol unit for the Conte-200 processor. This component is referred to as the “Control Logic” in the images and schematics. The microcontroller will contain all of the signal lines to the various parts of the datapath.

**You must do the following:**

1. Read and understand the microcontroller logic:

- Please refer to Appendix B: Microcontrol Unit for details.
  - **Note:** You will be required to generate the control signals for each state of the processor in the next phase, so make sure you understand the connections between the datapath and the microcontrol unit before moving on.
2. Implement the Microcontrol Unit using Brandonsim. The appendix contains all of the necessary information. Take note that the input and output signals on the schematics directly match the signals marked in the Conte-200 datapath schematic (see Figure 1).

## 5 Phase 3 - Microcode and Testing

In this final stage of the project, you will write the microcode control program that will be loaded into the microcontrol unit you implemented in Phase 2. Then, you will hook up the control unit you built in Phase 2 of the project to the datapath you implemented in Phase 1. Finally, you will test your completed computer using a simple test program and ensure that it properly executes.

**You must do the following:**

1. Fill out the microcode.xlsx (Excel spreadsheet) file that we have provided. You will need to mark which control signal is high (that is 1) for each of the states. We have given you a starting template.
2. After you have completed all the states, convert the binary strings you just computed into hex and move them to the main ROM (the Excel sheet will do this for you; just copy-and-paste the hex column into the Brandonsim ROM). Do the same for the sequencer and condition ROMs.
3. Connect the completed control unit to the datapath you implemented in Phase 1. Using Figures 1 and 2, connect the control signals to their appropriate spots.
4. Finally, it is time to test your completed computer. Use the provided assembler (found in the “assembly” folder) to convert a test program from assembly to hex. For instructions on how to use the assembler and simulator, see README.txt in the “assembly” folder.

We recommend using test programs that contain a single instruction since you are bound to have a few bugs at this stage of the project. Once you have built confidence, test your processor with the provided **pow.s** program as a more comprehensive test case.

## 6 Deliverables

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

- Brandonsim Datapath File (Conte-200.circ)
- Microcode file (microcode.xlsx)

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 combines attributes of both ARM and the LC-2200 ISA defined in the Ramachandran & Leahy textbook for CS 2200.

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																						SR2
ADDI	0001					DR				SR1																						immval20
NAND	0010					DR				SR1																						SR2
SKP	0011					mode				SR1																						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

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
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

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
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

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
0	1	0	0	0	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

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
0	1	0	1	0	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

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
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.

## 8 Appendix B: Microcontrol Unit

You will make a microcontrol unit which will drive all of the control signals to various items on the datapath. This Finite State Machine (FSM) can be constructed in a variety of ways. You could implement it with combinational logic and flip flops, or you could hard-wire signals using a single ROM. The single ROM solution will waste a tremendous amount of space since most of the microstates do not depend on the opcode or the conditional test to determine which signals to assert. For example, since the condition line is an input for the address, every microstate would have to have an address for condition = 0 as well as condition = 1, even though this only matters for one particular microstate.

To solve this problem, we will use a three ROM microcontroller. In this arrangement, we will have three ROMs:

- the main ROM, which outputs the control signals,
- the sequencer ROM, which helps to determine which microstate to go at the end of the FETCH state,
- and the condition ROM, which helps determine whether or not to skip during the SKP instruction.

Examine the following:

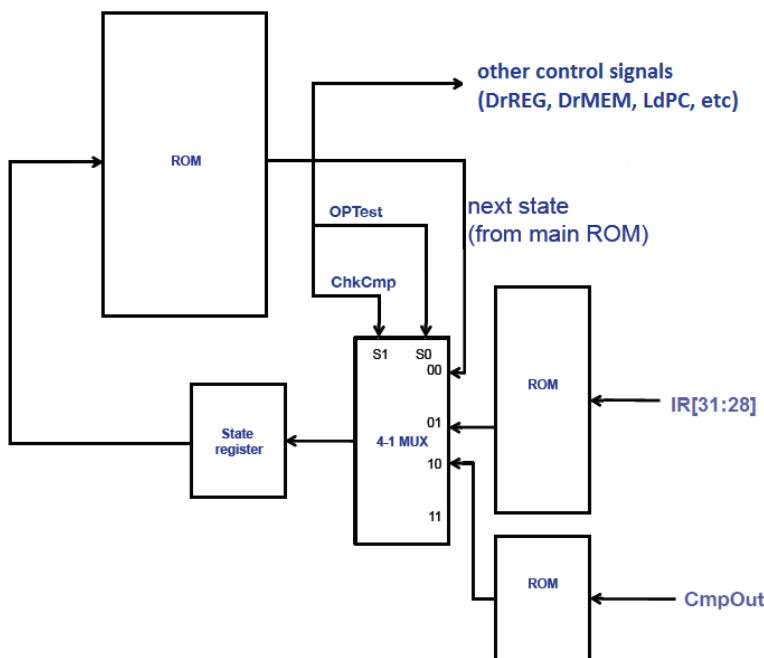


Figure 2: Three ROM Microcontrol Unit

As you can see, there are three different locations that the next state can come from: part of the output from the previous state (main ROM), the sequencer ROM, and the condition ROM. The mux controls which of these sources gets through to the state register. If the previous state's "next state" field determines where to go, neither the `OPTest` nor `ChkCond` signals will be asserted. If the opcode from the IR determines the next state (such as at the end of the FETCH state), the `OPTest` signal will be asserted. If the comparison circuitry determines the next state (such as in the SKP instruction), the `ChkCmp` signal will be asserted. Note that these two signals should never be asserted at the same time since nothing is input into the "11" pin on the MUX.

The sequencer ROM should have one address per instruction, and the condition ROM should have one address for condition true and one for condition false.

**Note:** Brandonsim has a minimum of two address bits for a ROM (i.e. four addresses), even though only one address bit (two addresses) is needed for the condition ROM. Just ignore the other two addresses. You should design it so that the high address bit for this ROM is permanently set to zero.

Before getting down to specifics you need to determine the control scheme for the datapath. To do this examine each instruction, one by one, and construct a finite state bubble diagram showing exactly what control signals will be set in each state. Also determine what are the conditions necessary to pass from one state to the next. You can experiment by manually controlling your control signals on the bus you've created in Phase 1 - Implement the Datapath to make sure that your logic is sound.

Once the finite state bubble diagram is produced, the next step is to encode the contents of the Control Unit ROM with the provided Excel sheet. Then you must design and build (in Brandonsim) the Control Unit circuit which will contain the three ROMs, a MUX, and a state register. Your design will be better if it allows you to single step and ensure that it is working properly. Finally, you will load the Control Unit's ROMs with the hexadecimal generated from the Excel sheet.

Note that the input address to the ROM uses bit 0 for the lowest bit of the current state and 5 for the highest bit for the current state.

Table 3: ROM Output Signals

Bit	Purpose	Bit	Purpose	Bit	Purpose	Bit	Purpose	Bit	Purpose
0	NextState[0]	6	DrREG	12	LdPC	18	WrREG	24	OPTest
1	NextState[1]	7	DrMEM	13	LdIR	19	WrMEM	25	ChkCmp
2	NextState[2]	8	DrALU	14	LdMAR	20	RegSelLo		
3	NextState[3]	9	DrPC	15	LdA	21	RegSelHi		
4	NextState[4]	10	DrOFF	16	LdB	22	ALULo		
5	NextState[5]	11	DrADDR	17	LdCmp	23	ALUHi		

Table 4: Register Selection Map

RegSelHi	RegSelLo	Register
0	0	RX (IR[27:24])
0	1	RY (IR[23:20])
1	0	RZ (IR[3:0])
1	1	\$ra

Table 5: ALU Function Map

ALUHi	ALULo	Function
0	0	ADD
0	1	SUB
1	0	NAND
1	1	A + 1