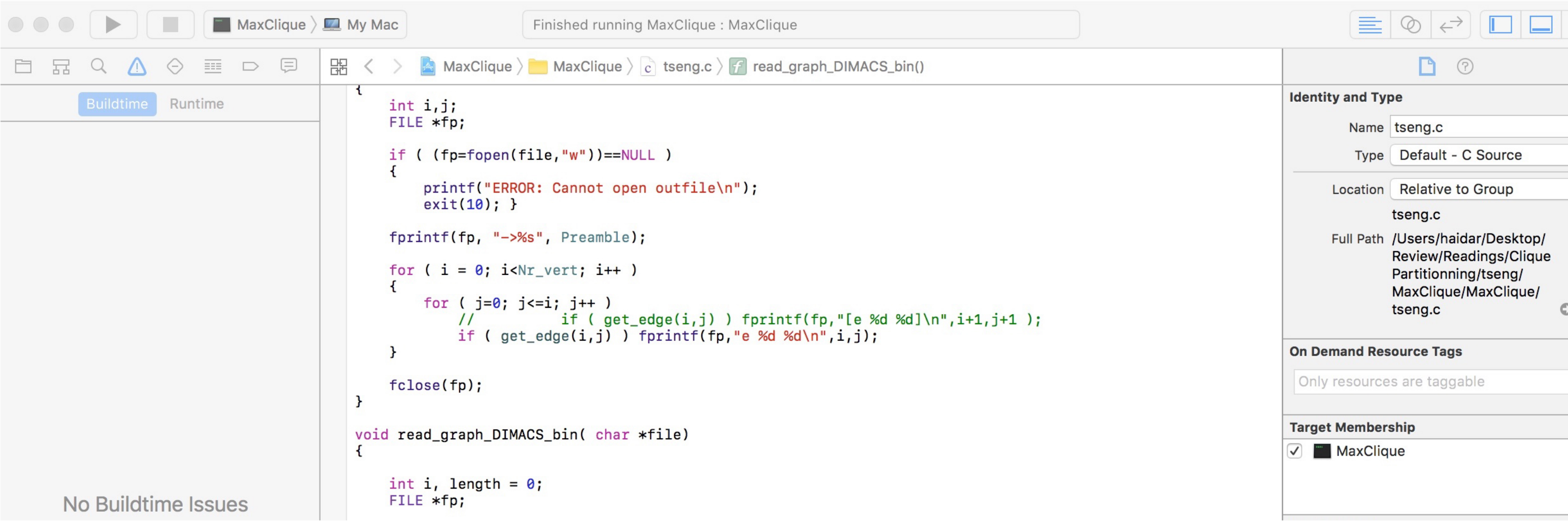
Hands-On Computer Organization Lab

With C and Verilog

Haidar M. Harmanani

First printing, March 2016


MaxClique
My Mac
Finished running MaxClique : MaxClique
MaxClique 〉 MaxClique 〉 tseng.c 〉 read_graph_DIMACS_bin()
Buildtime
Runtime
No Buildtime Issues

```
{
    int i,j;
    FILE *fp;

    if ( (fp=fopen(file,"w"))==NULL )
    {
        printf("ERROR: Cannot open outfile\n");
        exit(10); }

    fprintf(fp, "->%s", Preamble);

    for ( i = 0; i<Nr_vert; i++ )
    {
        for ( j=0; j<=i; j++ )
            //          if ( get_edge(i,j) ) fprintf(fp,"[e %d %d]\n",i+1,j+1 );
            if ( get_edge(i,j) ) fprintf(fp,"e %d %d\n",i,j);
    }

    fclose(fp);
}

void read_graph_DIMACS_bin( char *file)
{

    int i, length = 0;
    FILE *fp;
```

Identity and Type

Name tseng.c

Type Default - C Source

Location Relative to Group

tseng.c

Full Path /Users/haidar/Desktop/Review/Readings/Clique Partitionning/tseng/MaxClique/MaxClique/tseng.c

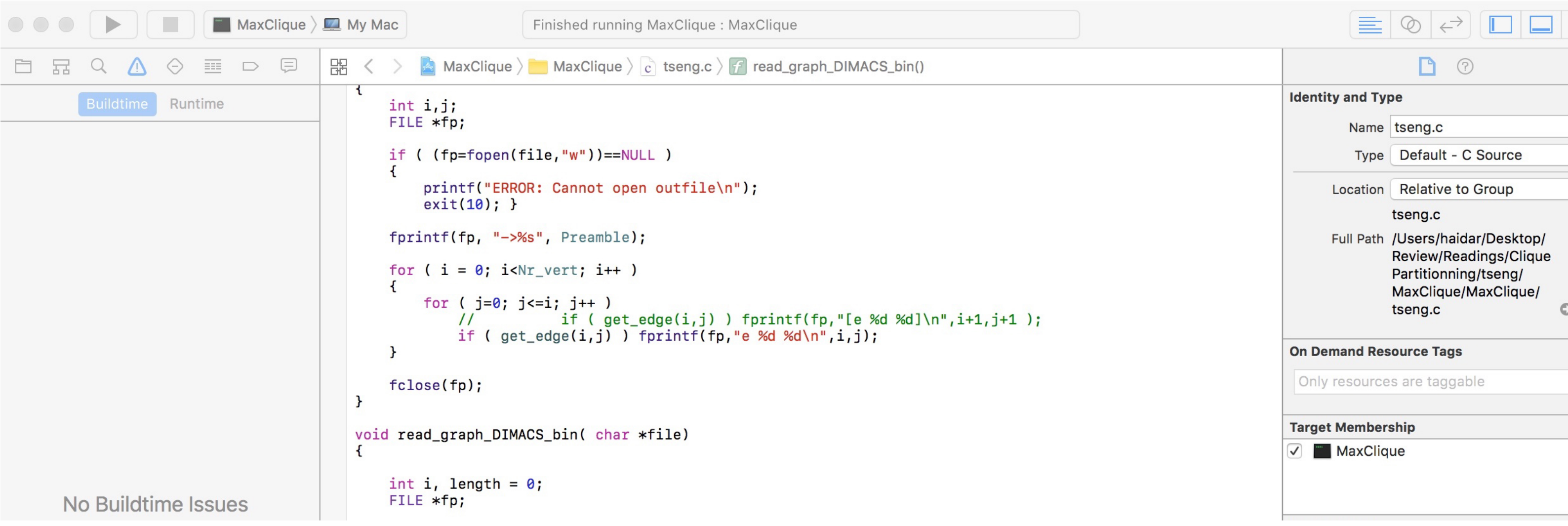
On Demand Resource Tags

Only resources are taggable

Target Membership

✓ MaxClique

Contents



```
{
    int i,j;
    FILE *fp;

    if ( (fp=fopen(file,"w"))==NULL )
    {
        printf("ERROR: Cannot open outfile\n");
        exit(10); }

    fprintf(fp, "->%s", Preamble);

    for ( i = 0; i<Nr_vert; i++ )
    {
        for ( j=0; j<=i; j++ )
            //           if ( get_edge(i,j) ) fprintf(fp,"[e %d %d]\n",i+1,j+1 );
            if ( get_edge(i,j) ) fprintf(fp,"e %d %d\n",i,j);
    }

    fclose(fp);
}

void read_graph_DIMACS_bin( char *file)
{

    int i, length = 0;
    FILE *fp;
```

Identity and Type

Name: tseng.c

Type: Default - C Source

Location: Relative to Group

tseng.c

Full Path: /Users/haidar/Desktop/Review/Readings/Clique Partitionning/tseng/MaxClique/MaxClique/tseng.c

On Demand Resource Tags

Only resources are taggable

Target Membership

✓ MaxClique

1. Milestone 1: The Pinky Processor 2.0

"RISC architecture is gonna change everything"

—Hackers (1995)

Objectives

1. Model, implement, and simulate a simple RISC ISA using C
2. Understand the inner operation of a modern computer.
3. Translate C code into a fictitious RISC-based programming language.
4. Write assembly language code using a fictitious RISC-based programming language.

1.1 The Pinky Machine

The design team has decided to update the *pinky processor* in order to include a more realistic instruction set that includes control flow and functions. The result is a general purpose 32-bit architecture that has a small memory for program and data storage, 32 registers for computations, and a relatively small instruction set. The design of the *pinky processor* is kept small to permit reasonably good test coverage and to allow a complete implementation of the machine. All registers are 16 bits wide, and when loaded with an address can specify any location in the memory. All instructions are 32 bits in length.

The new processor has one single 1024K memory. The program counter (PC) is 8 bits. While all instructions reference register operands, only the load and store instructions can reference the memory. In order to keep the design simple, the *Pinky Machine* has two separate memories: a *data memory* and an *instruction memory* that always start at address 0. The `pinky processor` has now 32 16-bit registers (Table 1.1) that are organized into a register file and labeled `W0-W31`. Registers `W0-W7` can be used to pass arguments to subroutines, or to return values from subroutines while registers `W8-W9` are used for temporary computations. Registers `W19-W29` are used for general purpose computations. The main difference between both sets of registers is that the values of temporary registers are not preserved by the callee on a subroutine or function call. Finally,

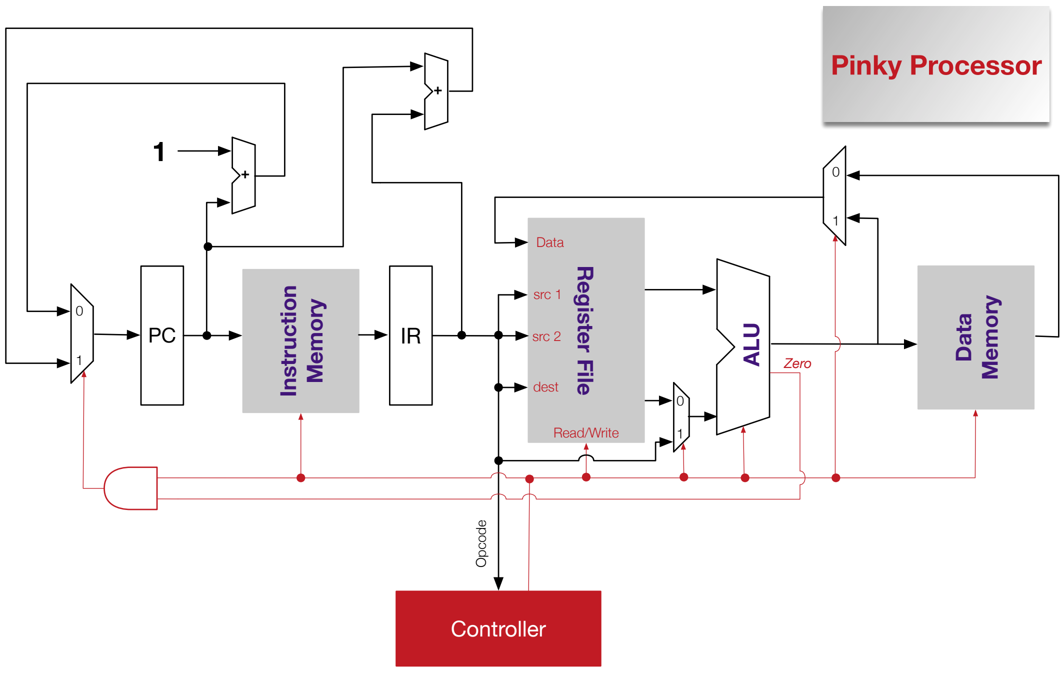


Figure 1.1: Data path for the Pinky Machine

Register	Purpose	Effect or Usage
`W0 - W7`	Subroutines arguments/results	Passing arguments to methods
`W8 - W18`	Temporary registers	Values will not be saved.
`W19 - W29`	Saved	
`W30`	Link register	`W30` ← PC
`W31`	The constant value 0	

Table 1.1: Pinky Processor Registers

registers `W30` and `W31` are special purpose registers: the first is used to save the return address after a subroutine call while the second corresponds to the constant 0. The corresponding datapath for the `Pinky` processor is shown in Figure 1.1

1.2 Instruction Set

The `Pinky Processor` instructions are shown in Figure 1.2 and further detailed in Table 1.2. We will represent these using C macros including: 1) an encoding that represents the instruction in memory; 2) arguments that model the inputs to the instruction, and 3) pseudocodes that represent what the instructions do. The `Pinky Processor` has the following instruction types:

1. The `D-Type` which refers to `load` and `store` instructions.
2. The `R-Type` which refers to register-to-register instructions. These include: `ADD`, `SUB`, `AND`, `OR`, `XOR`, `ANDS`, `ADDS`, `SUBS`, `SL`, and `SR`.
3. The `I-Type` which refers to immediate instructions including: `ADDI` and `SUBI`.
4. The `CB-Type` which refers to conditional branch instructions including: `CBZ`, `CBNZ`, `BZ`, and `BN`.
5. The `B-Type` which refers to branch instructions including `BR` and `BL`.
6. The `Halt-Type` which refers to one instruction, `Halt`, which is used to stop the program

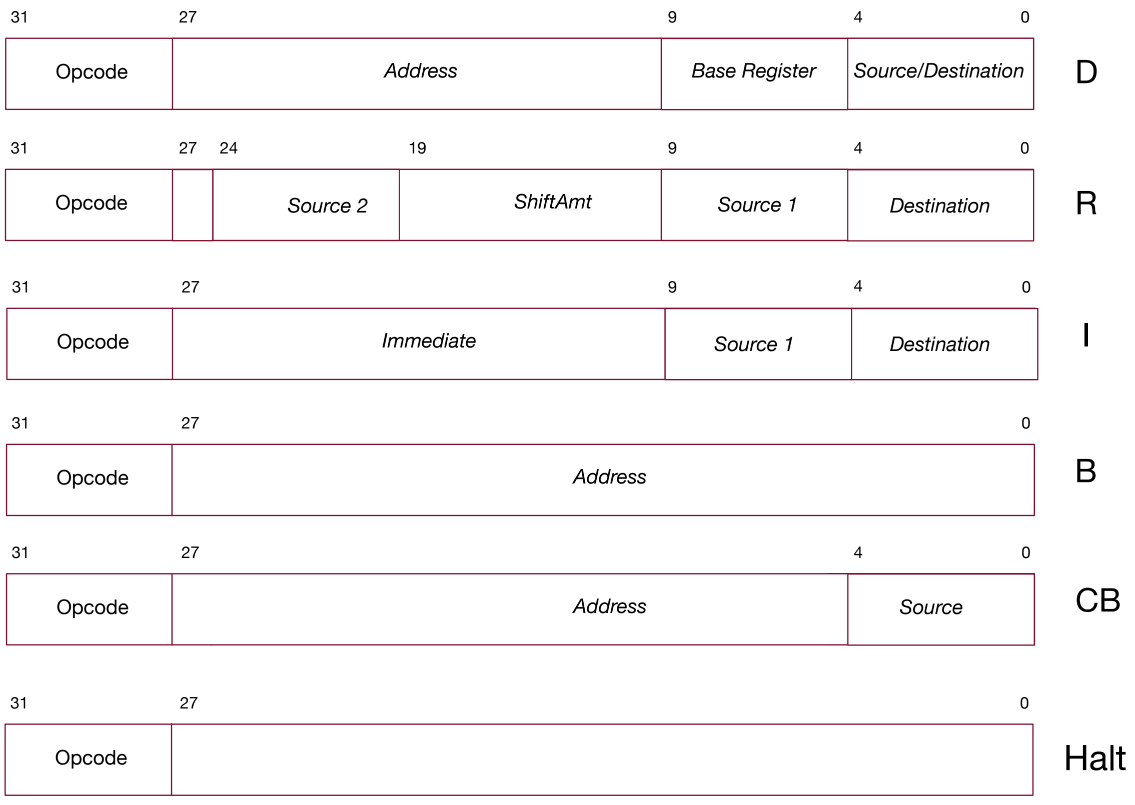


Figure 1.2: Pinky Machine ISA

execution.

In order to reduce the *opcode size*, we will use four bits and assign the same *opcode* to all R-type instructions. However, once an R-type in instruction is identified, the ALU decoder will further decode the instruction for the proper execution. The ALU *opcodes* are shown in Table 1.3.

1.3 Conditions, Loops, and Decisions

There are two types of branch instructions: *unconditional* and *conditional*. Unconditional branch instructions jump to an address *unconditionally* such as the `BR` and `BL` instructions. The `BR` instruction performs a direct, PC-relative branch. The offset from the current PC to the destination is encoded within the instruction. The range is limited by the space available within the instruction to record the offset and is +/-1024K. The *Branch and Link* instruction, `BL` , is used in order to call a subroutine by performing an absolute branch to the address specified in the destination register. The return address is saved in register `W30` and is used to restore the value of the PC when a `RET` instruction is executed.

The *Pinky Machine* has also two conditional branch instructions that start with a `B` and are postfixed with an appropriate condition code, `N` for Negative or `Z` for Zero as shown in Table 1.4, and resulting with two instructions, `BZ`, and `BN`. The condition flags are set or reset using the `SUBS`, `ANDS`, and `ADDS` instructions. When executing branch instructions and for simplicity, the processor will use the address in the instruction as is. That is, there will be no shifting or address manipulation. Data processing instructions do not affect the condition code flags. For example, the below `ANDS` instruction causes the ALU flags to be either set or cleared depending of the outcome of the instruction:

OpCode	**Instruction**	**Mnemonic**	**Effect**
0000	HALT	HALT	Halts execution
0001	LDW	LDW(W_a, address, displacement)	PC++; $W_a \leftarrow$ M[address + displacement]
0010	STW	STW(W_a, address, displacement)	PC++; M[address + displacement] $\leftarrow W_a$
0011	ADD	ADD(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] $+$ R[W_c]
0011	SUB	SUB(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] $-$ R[W_c]
0011	AND	AND(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] & R[W_c]
0011	OR	OR(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] $\mid$ R[W_c]
0011	XOR	XOR(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] $\oplus$ R[W_c]
0011	ADDS	ADDS(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] $+$ R[W_c]; $Z \leftarrow (R[W_a] == 0)\ ?\ 1\ :\ 0$ $N \leftarrow (R[W_a] < 0)\ ?\ 1\ :\ 0$
0011	SUBS	SUBS(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] $-$ R[W_c] $Z \leftarrow (R[W_a] == 0)\ ?\ 1\ :\ 0$ $N \leftarrow (R[W_a] < 0)\ ?\ 1\ :\ 0$
0011	ANDS	ANDS(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] & R[W_c] $Z \leftarrow (R[W_a] == 0)\ ?\ 1\ :\ 0$ $N \leftarrow (R[W_a] < 0)\ ?\ 1\ :\ 0$
0100	SL	SL(W_a, W_b, $ShiftAmt$)	PC++; R[W_a] = R[W_b] $<<$ ShiftAmt
0101	SR	SR(W_a, W_b, $ShiftAmt$)	PC++; R[W_a] = R[W_b] $>>$ ShiftAmt
0110	ADDI	ADDI(W_a, W_b, Imm)	PC++; R[W_a] = R[W_b] $+$ Imm
0111	SUBI	SUBI(W_a, W_b, Imm)	PC++; R[W_a] = R[W_b] $-$ Imm
1000	CBZ	CBZ(W_a, $Address$)	if (W_a == 0) then PC $\leftarrow$ Address
1001	CBNZ	CBNZ(W_a, $Address$)	if (W_a != 0) then PC $\leftarrow$ Address
1010	BZ	BZ($Address$)	if (Z == 1) then PC $\leftarrow$ PC + Address;
1011	BN	BN($Address$)	if (N == 1) then PC $\leftarrow$ PC + Address;
1100	BR	BR($Address$)	PC $\leftarrow$ PC + Address
1101	BL	BL($Address$)	W30 $\leftarrow$ PC; PC $\leftarrow$ PC + Address
1110	RET	RET	PC $\leftarrow$ W30

Table 1.2: Pinky Processor Instruction Set Summary

OpCode	**ALU OpCode**	**Instruction**	**Mnemonic**
0011	000	ADD	ADD(W_a, W_b, W_c)
0011	001	SUB	SUB(W_a, W_b, W_c)
0011	010	AND	AND(W_a, W_b, W_c)
0011	011	OR	OR(W_a, W_b, W_c)
0011	100	XOR	XOR(W_a, W_b, W_c)
0011	101	ADDS	ADDS(W_a, W_b, W_c)
0011	110	SUBS	SUBS(W_a, W_b, W_c)
0011	111	ANDS	ANDS(W_a, W_b, W_c)

Table 1.3: ALU Opcodes

```
ADDS (W11, W11,W31)  // address 9
BZ(12)               // address 10
ADD(W12, W13, W14)   // address 11
```

Condition Code	Meaning
N	Negative condition code. Set to 1 if result is negative.
Z	Zero condition code. Set to 1 if the result of the instruction is 0.

Table 1.4: Condition Codes

The *Pinky Machine* has two additional conditional branch instructions, CBZ, CBNZ, that branch conditionally to an address subject to the value in the register. Let's take the following code as an example:

```
CBZ(W0, 15),                    // Address 12
SUBI(W0, W0, 1),                // Address 13
BR(12)                          // Address 14
```

In the above example, the CBZ instruction checks if W0 is zero. If it is then the control is transferred to the instruction at address 12. Otherwise, SUBI instruction decrements W0. We branch next to address 12 and the test is repeated. Another example, shown below, involves the Z flag. In this example, we check for equality, that is if the Z flag is set to 1 by the outcome of the *SUBS* instruction:

```
SUBS(W0, W7, W9)        // Address 5,  W0 = W7 - W9
BZ(15)                  // Branch to address 15 if W7 == W9
```

In the above example, we test for equality, W9 == W7, by subtracting W7 and W9. If the result is zero, then the Z flag would be set the o 1 and then we branch to o address 15.

1.4 Loops and decisions

Loops can be programmed in the *Pinky Processor* in a similar way using branches. For example, assume we have the following C code:

```
if (a == 5)
   b = 7;
```

This could be translated in the Pinky Assembly Code as follows:

```
SUBS(W0, W5, W9)  // Address 5.  W0 = W5 - W9.  Assume W5 = 5
CBNZ(W0, 8),      // Address 6.  Branch to address 15 if Z flag is Zero
LDW(W9,10),       // Address 7.  Assume M[10] == 7
```

Another example using a while loop:

```
while (a != 0)
{
b = b + c;
a = a - 1;
}
```

Can be transllted to the following code fragment:

```
LDW(W8, 20),      // Address 10: Assume a is at address 20
ADDS(W1,W8,W31), // Address 11: Add W8 to 0.
CBZ(W8, 16),      // Address 12: If result is 0 then branch to address 16
ADD(W9, W9, W10),// Address 13: It is not so Add 1 to b
SUB(W8, W8, 1),  // Address 14: Decrement A
BR(10)            // Repeat
```

1.5 C Elements for an Instruction Set

There are four elements that you would need to implement in order to simulate an instruction set. These are as follows:

1. Opcodes and mnemonics which will be implemented using C macros.
2. Registers which will be modeled using C variables.
3. Memories which will be implemented using an array.
4. Hardware elements such as multiplexers and ALUs which will be implemented using C functions.

Instructions will be packed in one word using C macros. For example, the ADD instruction is implemented as follows:

```
#define ADD(Dest, Reg2, Reg1) (OP_ALU << 28) | (OP_ALU_ADD << 25) | (Reg2 <<
20) | (Reg1 << 5) | Dest
```

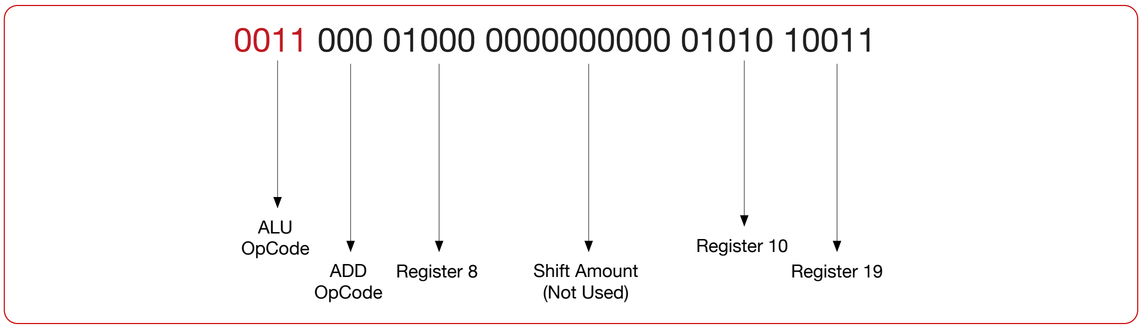
You will, of course, remove all references to individual ALU opcodes and use the following instead with the assumption of having one ALU with a dedicated decoder:

```
...
#define OP_ALU  0x3
...
#define OP_ALU_ADD  0x0
```

The above ADD would be executed as follows:

```
ADD (W19, W8,W10)
```

The resulting Pinky machine language language instruction is shown below:



1.6 Capstone Tasks

The lab this week involves completing the design of the `Pinky` processor. This involves updating your code from last week so that it includes the new instructions. The tasks shall include:

- Task 1: Complete the missing *mnemonics* and *opcodes* in the provided starter code.
- Task 2: Update the *fetch and decode unit* by adding the missing opcodes.
- Task 3: Implement the missing components using C functions.
- Task 4: Test your code by executing the following program (no need to worry about the printf statements):

```
int n1 = 5, n2 = 20, n3 = 30;

    if (n1 >= n2 && n1 >= n3)
        printf("%.2f is the largest number.", n1);

    // if n2 is greater than both n1 and n3, n2 is the largest
    if (n2 >= n1 && n2 >= n3)
        printf("%.2f is the largest number.", n2);

    // if n3 is greater than both n1 and n2, n3 is the largest
    if (n3 >= n1 && n3 >= n2)
        printf("%.2f is the largest number.", n3);
```

- Task 5: Test your code by executing the following program:

```
int i = 5; x = 10, z = 20;

y = x + 5;
w = z - 5;
if (w == y)
     output = w & y;
 else
    output = w | y;
sum(y+w)

int sum(x, y)
{
  return (x +y)
}
```