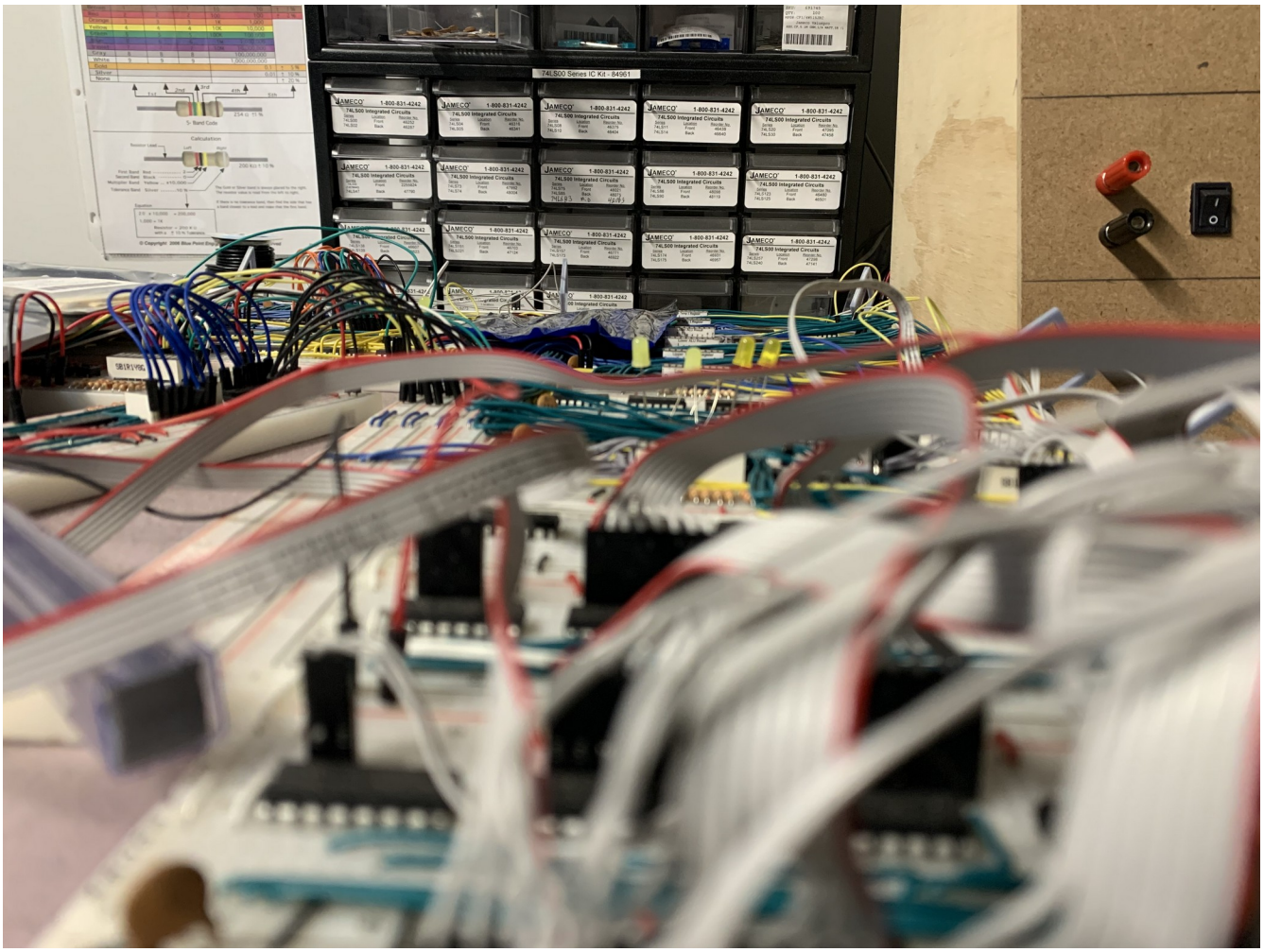

ERYJUS

16-Bit Computer From Scratch

Architecture Reference

To my loving wife, who has absolutely no idea what I am doing or why I am doing it but unrelentingly supports my efforts every day.

Mostest!

Table of Contents

Introduction

First of all, thank you so much for your time and interest in this project. This project was born of a desire to learn more about digital electronics, specifically CPUs. I am no expert. Not even close. If you are reading this, it means that I have enjoyed some margin of success with this project.

My 16-Bit Computer From Scratch project (to which I refer in this reference as ***16bcfs***) is my third computer build project. The first was Ben Eater's (https://eater.net/) 8-bit Breadboard Computer. It was my first experience plugging integrated circuits (ICs) into a breadboard. I was hooked.

Once I completed Ben's kit build, I went on to try to expand that to 16-bits. This one failed. I started to get some really bad capacitance problems which I was never really able to work out. Near the end of this project, I discovered James Sharman's (https://www.youtube.com/@weirdboyjim) build. He also had an 8-bit build which was working quite well when I found it.

It was inspiration from James's work that I took on this project and decided to document it in YouTube video form (https://www.youtube.com/channel/UCxK3e2ptj5LElVEQTdC3oeQ).

Please, let me be absolutely clear about something: I absolutely respect both Ben's and James's works and my intention is not to try to replace them. Indeed, I would not be here today without what I learned from both of them. Instead, my objective is to present something different, extending what I have learned, hopefully in a different direction. A 16-bit computer from discrete logic ICs is ambitious.

Project Goals

It is important to have a set of goals for a project. Without goals, how will you ever know you have arrived at your destination. For the ***16bcfs***, my original goals did not persist and have evolved as I have realized my capabilities and limitations. The original goals were:

16-Bit Computer From Scratch Architecture Reference

- Create a 16-bit CPU using fundamental Integrated Circuits (ICs)
- Make a pipelined CPU, as near scalar (one instruction per clock cycle) as possible
- Make the CPU clock as fast as possible (minimum 1MHz; prefer 10+ MHz)
- Support hardware interrupts
- Memory will be dual-channel reads and single channel writes
- Contain up to 128KB addressable memory (2^16 words; 2^17 bytes)
- Integrate a counter/timer, UART, Video, keyboard, and storage into a computer (all fundamental ICs)
- Build this computer onto Printed Circuit Boards (PCBs)
- Write an emulator for this computer
- Write a cross-assembler for this computer
- Write a BIOS for this computer (with BIOS function call support)
- Include an Integer-BASIC on the BIOS
- Write a basic OS for this computer, loadable from storage
- Rewrite the assembler to be self-hosted on the computer
- Write a compiler for the computer (PASCAL?)
- Make the OS self-hosted on the computer

I will let you judge for yourself how I did with these goals. My own self-analysis against my goals is not kind.

CPU Architecture

This computer, as with any computer, has it own architecture. There is nothing groundbreaking in my build other than the simple fact it's the first 16-bit computer I have completed.

This section will outline the basics of the underlying architecture of the CPU. Nothing is intended to be hidden and the intent is to hopefully present this information in such as way as you can follow along. Think back to the early days of the home computers and the documentation which accompanied those systems. It is that standard of documentation for which I am aiming with this reference.

The ***16bcfs*** CPU is sub-divided into several sections:

- Organization
- Registers
- Buses
- Arithmetic Logic Unit (ALU)
- Instruction Execution & Control Logic
- Memory
- IO Ports
- CPU Mid-Planes and Backplane

Admittedly, memory is not typically part of the CPU unless we are talking about memory cache. However, in this case, the memory addressing architecture is closely tied to the CPU architecture and deserves to be included in the CPU overview. This CPU does not contain any memory cache.

Similarly, IO Ports are not part fo the CPU. However, the CPU does need to understand how to communicate with the IO Ports (and there are CPU instructions to do this). Therefore, they must also be considered as part of the CPU build.

Organization

Before we get into the physical architecture, we first need to understand how the data is organized on the computer. This detail is significant as it will dictate how all data is read, written, and moved throughout the system.

The ***16bcfs*** will be organized into 16-bit words. All memory access will be in 16-bits and will be bound on the 16-bit boundary. All data movements will be in full 16-bit words.

Now, not all devices and not all interactions with the outside world understand 16-bit data as the most granular data size. Most other systems recognize an 8-bit byte as the lowest element.

The ***16bcfs*** will need to be able to interact with these outside requirements. Some devices (Serial UART for example) will only respond to expose byte-sized data. To facilitate this, the ability to swap bytes of a 16-bit word will be necessary. In this manner, the CPU can send a 16-bit word to the UART, but the UART will only accept one of the bytes. Swapping bytes in place and re-sending the data makes will allow the CPU to interact with this byte-size requirement.

See the Byte-Swapped Pseudo-Bus for more information.

Buses

When we speak of ***the bus*** in a computer, you will undoubtedly understand that it is the most contended resource. Everything wants to use ***the bus***. Nearly always at the same time.

The ***16bcfs*** will also have a bus, well a few of them. I chose to include several buses in my CPU Architecture. This section is dedicated to outlining each of them and how they are used.

A bus is a collection of 1 or more signals which will have multiple components which have an interest in this collection of signals. A bus is intended to allow several components to assert a value onto the

bus and several components to listen to what is on that bus. However, the key here is that only one thing can assert signals onto any given bus at a time; multiple signals asserted at the same time will only muddy the signals.

The following buses in the ***16bcfs*** all have 16 signals, representing each of the 16 bits in the data word.

Main Bus

Generally speaking, the Main Bus conducts a word of data around the ***16bcfs***. This word can be asserted by or read by nearly anything on the computer. This would include registers, memory, devices, I/O Control Ports and more. If nothing is asserting onto this bus, then the value is pulled low to all zeros for each bit. See the RZ register for more information.

ALU Bus A

ALU Bus A is used to deliver a data word to the left-hand side of a binary ALU operation (such as ADDing 2 numbers together), or the only data word to a unary ALU operation (such as a bitwise NOT). The only device on the computer which will read from ALU Bus A is the ALU. Anything else merely presents data to be consumed by the ALU. If nothing is asserting onto this bus, then the value is pulled low to all zeros for each bit. See the RZ register for more information.

ALU Bus B

ALU Bus B is used to deliver a data word to the right-hand side of a binary ALU operation (such as ADDing 2 numbers together). The only device on the computer which will read from ALU Bus B is the ALU. Anything else merely presents data to be consumed by the ALU. If nothing is asserting onto this bus, then the value is pulled low to all zeros for each bit. See the RZ register for more information.

Address Bus 1

Address Bus 1 will contain the memory address for the next instruction to be fetched from memory. The only device which will ever have interest in asserting to this bus is the *current* Program Counter register (see the Register section). Something will always be asserting onto this bus when the computer is operating.

Address Bus 2

Address Bus 2 will contain the memory address for a 16-bit wide data cell for loading from or storing to a memory location. This will be used for only some instructions. For memory operations using this bus, data will be read from or asserted to the Main Bus depending on the operation being conducted. If nothing is asserting onto this bus, then the value is pulled low to all zeros for each bit.

Byte Swap Pseudo-Bus

The Swap Bus is another entry point to assert onto the Main Bus. The difference with this bus and the Main Bus is that this bus will swap the 2 bytes such that the Most Significant Byte is presented on the lower 8 bits of the bus and the Least Significant Bye is presented on the upper 8 bits. Bit order within the bytes is preserved. Nothing can read from this pseudo bus directly, it is all read from the Main Bus.

Fetch Bus

The Instruction Fetch Bus connects the memory value retrieved as a result of looking up what is on Address Bus 1 and presenting that value which was stored in memory to the Fetch Register as a word in the instruction stream. During this read and presentation to the Fetch Register, there is no discernment as to whether the word is an instruction or an immediate value. During reset, assertion to this bus is suppressed and the value is pulled low to all zeros for each bit. This has the effect of sending a NOP-AL instruction to the Fetch Register (see the NOP instruction below).

Instruction Execution Bus

The Instruction Execution Bus is used to move an instruction from the Fetch Register to the Instruction Register. Some instructions will need to suppress the contents of the Fetch Register as they are considered to be an immediate value and will be directed to other locations. When an instruction is suppressed from asserting from the Fetch Register, the value is pulled low to all zeros for each bit. This has the effect of sending a NOP-AL instruction to the Fetch Register (see the NOP instruction below).

Mid-Planes

There are several mid-planes which are involved in the 16bcfs build. These mid-planes provide for connections for common functionality and then plug into the one common backplane for the entire build.

Control Logic Mid-Plane

The Control Logic Mid-Plane provides a common location for handling all the Control Logic lookup and the output signals from the Control Logic. While not truly backplane, this mid-plan does also contain some logic for de-multiplexing the different signals to provide mutual exclusivity.

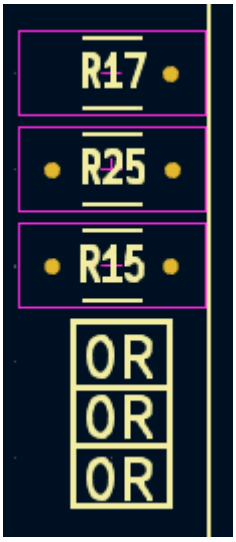

Figure 2: R15, R17, & R25 0R Ressitors

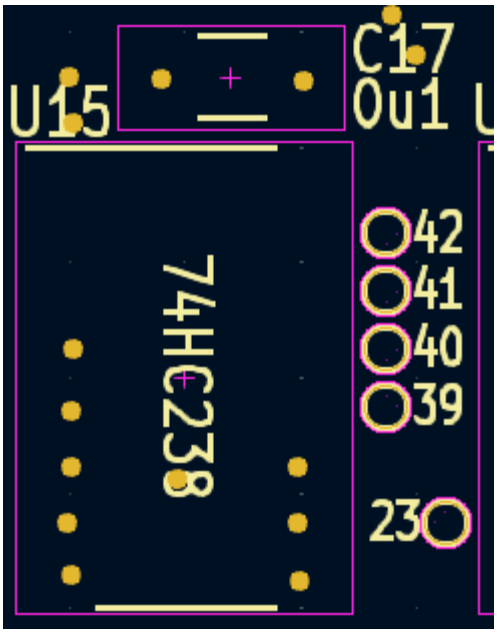

Figure 1: U15 with its test points (39-42)`

Not all ICs are populated. Not all connections are used. When an IC is populated but a signal is unused, a test point was added with the solder mask removed to allow to bodge a new signal in the future if required (see Figure 1). Similarly, when a connector off-board is used, a 1206 0R resistor is to be placed on that connector signal to ground (see Figure 2). This will ground the signal but that grounded signal can be replaced as required in the future. A simple bodge-wire from the test point to the signal-side of the 0R resistor (removing the resistor) is all that is needed to add new functionality.

16-Bit Computer From Scratch Architecture Reference

It is assumed that all signals will be used from the Control ROM Control Module. Every signal either makes its way through a de-multiplexer and then to an off-board connector or is directly routed to an off-board connector. Anything that is unused in the control logic is routed off board in case it is to be used in the future.

The following table shows the list of available signals on the off-board connectors.

Jumper	Pin	Currently Driven By
J22	9	Control ROM 6, Pin 8
J22	10	Control ROM 6, Pin 7
J22	11	Control ROM 5, Pin 8
J22	12	Control ROM 5, Pin 7
J22	13	Control ROM 5, Pin 6
J22	14	Control ROM 5, Pin 5
J22	15	Control ROM 5, Pin 4
J22	16	Control ROM 5, Pin 3
J25	16	0 Ohm Resistor to Ground
J38	11	0 Ohm Resistor to Ground
J38	12	0 Ohm Resistor to Ground
J38	13	0 Ohm Resistor to Ground
J38	14	0 Ohm Resistor to Ground
J38	15	0 Ohm Resistor to Ground
J38	16	0 Ohm Resistor to Ground
J39	16	0 Ohm Resistor to Ground
J40	14	0 Ohm Resistor to Ground
J46	16	0 Ohm Resistor to Ground
J73	11	0 Ohm Resistor to Ground
J73	12	0 Ohm Resistor to Ground
J73	13	0 Ohm Resistor to Ground
J73	14	0 Ohm Resistor to Ground
J73	15	0 Ohm Resistor to Ground

Jumper	Pin	Currently Driven By
J73	16	0 Ohm Resistor to Ground
J74	15	0 Ohm Resistor to Ground

Registers

The ***16bcfs*** CPU has several general-purpose registers and a small number of special-purpose registers. This section details out these registers in turn.

RZ – Zero Register

Register Index 0x00

The Zero Register is a read-only pseudo register which is used to assert the numeric value 0 onto the Main Bus, ALU Bus A, or ALU Bus B.

The Zero Register's hardware implementation is via pull-down resistors and therefore stores no value. Rather it's purpose is to provide a natural default value for the CMOS logic gates when there is no other value being asserted.

R1 – General Purpose Register 1

Register Index 0x01

R1 through R12 are General Purpose Registers intended to be used by software for any general purpose. There is no hardware use defined for these Registers.

The General Purpose Registers are initialized to 0 during reset.

R2 – General Purpose Register 2

Register Index 0x02

R1 through R12 are General Purpose Registers intended to be used by software for any general purpose. There is no hardware use defined for these Registers.

The General Purpose Registers are initialized to 0 during reset.

R3 – General Purpose Register 3

Register Index 0x03

R1 through R12 are General Purpose Registers intended to be used by software for any general purpose. There is no hardware use defined for these Registers.

The General Purpose Registers are initialized to 0 during reset.

R4 – General Purpose Register 4

Register Index 0x04

R1 through R12 are General Purpose Registers intended to be used by software for any general purpose. There is no hardware use defined for these Registers.

The General Purpose Registers are initialized to 0 during reset.

R5 – General Purpose Register 5

Register Index 0x05

R1 through R12 are General Purpose Registers intended to be used by software for any general purpose. There is no hardware use defined for these Registers.

The General Purpose Registers are initialized to 0 during reset.

R6 – General Purpose Register 6

Register Index 0x06

R1 through R12 are General Purpose Registers intended to be used by software for any general purpose. There is no hardware use defined for these Registers.

The General Purpose Registers are initialized to 0 during reset.

R7 – General Purpose Register 7

Register Index 0x07

R1 through R12 are General Purpose Registers intended to be used by software for any general purpose. There is no hardware use defined for these Registers.

The General Purpose Registers are initialized to 0 during reset.

R8 – General Purpose Register 8

Register Index 0x08

R1 through R12 are General Purpose Registers intended to be used by software for any general purpose. There is no hardware use defined for these Registers.

The General Purpose Registers are initialized to 0 during reset.

R9 – General Purpose Register 9

Register Index 0x09

R1 through R12 are General Purpose Registers intended to be used by software for any general purpose. There is no hardware use defined for these Registers.

The General Purpose Registers are initialized to 0 during reset.

R10 – General Purpose Register 10

Register Index 0x0A

R1 through R12 are General Purpose Registers intended to be used by software for any general purpose. There is no hardware use defined for these Registers.

The General Purpose Registers are initialized to 0 during reset.

R11 – General Purpose Register 11

Register Index 0x0B

R1 through R12 are General Purpose Registers intended to be used by software for any general purpose. There is no hardware use defined for these Registers.

The General Purpose Registers are initialized to 0 during reset.

R12 – General Purpose Register 12

Register Index 0x0C

R1 through R12 are General Purpose Registers intended to be used by software for any general purpose. There is no hardware use defined for these Registers.

The General Purpose Registers are initialized to 0 during reset.

SP – User Stack Pointer Register

Register Index 0x0D

The User Stack Pointer Register is used to keep track of the location of the stack during program execution. Values can be pushed onto the User Stack and popped from the User Stack. The User Stack Pointer Register always points to the last value pushed onto the stack, making it simpler to refer to that location as a scratch memory location.

The User Stack Pointer Register is initialized to 0 during reset.

RA – User Return Address Register

Register Index 0x0E

The User Return Address Register is used to keep track of the address to which a subroutine will return for User Programs.

There are 2 separate hardware registers which can act as the default User Program Counter Register, the other being the default User Return Address Register.

The User Return Address Register is initialized to 0 during reset.

NOTE: The S Flag controls which hardware register is the current User Program Counter with 0 being the default User Program Counter Register/User Return Address Register arrangement and 1 swapping the 2 hardware registers. The S Flag controls a hardware level change only, so the whichever register is acting as the User Program Counter is conceptually the User Program Counter Register. The S Flag simply controls which hardware register is accessed whenever the User Program Counter Register contents are needed.

PC – User Program Counter Register

Register Index 0x0F

The User Program Counter Register is used to keep track of the instruction in memory which is being executed for User Programs. Typically, with a few exceptions, the User Program Counter Register is incremented after each instruction is fetched into the Fetch Register.

There are 2 separate hardware registers which can act as the default User Program Counter Register, the other being the default User Return Address Register.

The User Program Counter Register is initialized to 0 during reset.

NOTE: The S Flag controls which hardware register is the current User Program Counter Register with 0 being the default User Program Counter Register/User Return Address Register arrangement and 1 swapping the 2 hardware registers. The S Flag controls a hardware level change only, so the whichever register is acting as the User Program Counter is conceptually the User Program Counter Register. The S Flag simply controls which hardware register is accessed whenever the User Program Counter Register contents are needed.

INTSP – Interrupt Stack Pointer Register

Register Index 0x10

The Interrupt Stack Pointer Register is used to keep track of the location of the stack during program execution. Values can be pushed onto the Interrupt Stack and popped from the Interrupt Stack. The Interrupt Stack Pointer Register always points to the last value pushed onto the stack, making it simpler to refer to that location as a scratch memory location.

The Interrupt Stack Pointer Register is initialized to 0 during reset.

INTRA – Interrupt Return Address Register

Register Index 0x11

The Interrupt Return Address Register is used to keep track of the address to which a subroutine will return for User Programs.

There are 2 separate hardware registers which can act as the default Interrupt Program Counter Register, the other being the default Interrupt Return Address Register.

The Interrupt Return Address Register is initialized to 0 during reset.

NOTE: The S Flag controls which hardware register is the current Interrupt Program Counter with 0 being the default Interrupt Program Counter Register/Interrupt Return Address Register arrangement and 1 swapping the 2 hardware registers. The S Flag controls a hardware level change only, so the whichever register is acting as the Interrupt Program Counter is conceptually the Interrupt Program Counter Register. The S Flag simply controls which hardware register is accessed whenever the Interrupt Program Counter Register contents are needed.

INTPC – Interrupt Program Counter Register

Register Index 0x12

The Interrupt Program Counter Register is used to keep track of the instruction in memory which is being executed for Interrupt Programs. Typically, with a few exceptions, the Interrupt Program Counter Register is incremented after each instruction is fetched into the Fetch Register.

There are 2 separate hardware registers which can act as the default Interrupt Program Counter Register, the other being the default Interrupt Return Address Register.

The Interrupt Program Counter Register is initialized to 0 during reset.

NOTE: The S Flag controls which hardware register is the current Interrupt Program Counter Register with 0 being the default Interrupt Program Counter Register/Interrupt Return Address Register arrangement and 1 swapping the 2 hardware registers. The S Flag controls a hardware level change only, so the whichever register is acting as the Interrupt Program Counter is conceptually the Interrupt Program Counter Register. The S Flag simply controls which hardware register is accessed whenever the Interrupt Program Counter Register contents are needed.

Assembly Language Reference

Conditional Execution

Every instruction in the ***16bcfs*** architecture is able to be conditionally executed. The conditional appears as a suffix in the instruction mnemonic and a prefix in the machine code.

When the condition is met, the instruction executes as normal. When the condition is not met, the instruction is suppressed from the Fetch Register into the Instruction Register and is replaced with a NOP instruction. See the NOP instruction for more information.

There are 16 conditions which can be evaluated and these are articulated in the table below.

Conditional	Mnemonic Suffix	Opcode Prefix (Binary)	Opcode Prefix (Hex)	Condition (Note: L = 1 when N != V
Always	-AL	0000	0	1
Never	-NV	0001	1	0
Equal	-EQ	0010	2	Z = 1
Not Equal	-NE	0011	3	Z = 0
Carry Set	-CS	0100	4	C = 1
Carry Clear	-CC	0101	5	C = 0
Minus	-MI	0110	6	N = 1
Positive or Zero	-PL	0111	7	N = 0
Overflow Set	-VS	1000	8	V = 1
Overflow Clear	-VC	1001	9	V = 0
Unsigned Higher	-HI	1010	A	C = 1 and Z = 0
Lower or Same	-LS	1011	B	C = 0 or Z = 1
Signed Greater or Equal	-GE	1100	C	L = 0
Signed Less Than	-LT	1101	D	L = 1
Signed Greater Than	-GT	1110	E	Z = 0 and L = 0
Signed Less Than or Equal	-LE	1111	F	Z = 1 or L = 1

JMP – Load PC with new Address

S	X	L	V	N	C	Z
-	-	-	-	-	-	-

Operation: R → PC

When this instruction is executed a new memory address is loaded into the PC Register. Control is transferred to this new location and the next instruction is loaded from this memory address.

When an immediate form of this instruction is executed, the contents of the Fetch Register are asserted onto the Main Bus as the Fetch Register will contain the value to be latched into the PC Register. As such, the value in the Fetch Register cannot be passed along to the Instruction Register and instead is replaced with a NOP instruction. This form of the instruction requires 2 words (the opcode and the immediate value) and therefore will take 2 CPU cycles to execute.

Addressing Mode	Assembly Language Form	Opcode	# Words	# Cycles
Immediate Value	JMP 1234	0FF3	2	2
Register	JMP R1	0060	1	1

MOV – Move a Value

S	X	Z	C	N	V	L
-	-	-	-	-	-	-

Operation: R → IO | IO → R | R → R | IO → IO

When this instruction is executed, data is moved from one location to another in the computer. This can take one of several forms:

A) Data can be moved from a Register to an IO Port. In this form data is asserted onto the Main Bus from a Register and latched by an IO Port. The source register may be the Fetch Register, Zero Register, or other register that is not directly maintainable. Flags are unchanged with this operation.

B) Data can be moved from an IO Port to a Register. In this form data is asserted onto the Main Bus from an IO Port and latched by a Register. The target registers that are not directly maintainable are invalid for this operation. Flags are unchanged with this operation.

C) Data can be moved from a Register to a Register. In this form data is asserted onto the Main Bus from a Register and latched by a Register. The source register may be the Fetch Register, Zero Register, or other register that is not directly maintainable. Flags are unchanged with this operation.

D) Data can be moved from an IO Port to another IO Port. In this form data is asserted onto the Main Bus from an IO Port and latched by an IO Port.

It is helpful to consider that for the ***16bcfs***, IO Ports are treated as registers.

When an immediate form of this instruction is executed, the contents of the Fetch Register are asserted onto the Main Bus as the Fetch Register will contain the value to be latched into the Register. As such, the value in the Fetch Register cannot be passed along to the Instruction Register and instead is replaced with a NOP instruction. This form of the instruction requires 2 words (the opcode and the immediate value) and therefore will take 2 CPU cycles to execute.

Addressing Mode	Assembly Language Form	Opcode	# Words	# Cycles
Immediate Value	MOV R1,<imm16>	001	2	2
Immediate Value	MOV R2,<imm16>	002	2	2
Immediate Value	MOV R3,<imm16>	003	2	2
Immediate Value	MOV R4,<imm16>	004	2	2
Immediate Value	MOV R5,<imm16>	005	2	2
Immediate Value	MOV R6,<imm16>	006	2	2
Immediate Value	MOV R7,<imm16>	007	2	2
Immediate Value	MOV R8,<imm16>	008	2	2
Immediate Value	MOV R9,<imm16>	009	2	2
Immediate Value	MOV R10,<imm16>	00A	2	2
Immediate Value	MOV R11,<imm16>	00B	2	2
Immediate Value	MOV R12,<imm16>	00C	2	2
Immediate Value	MOV SP,<imm16>	00D	2	2
Immediate Value	MOV RA,<imm16>	00E	2	2
Reg/Reg	MOV R1,RZ	011	1	1
Reg/Reg	MOV R2,RZ	012	1	1
Reg/Reg	MOV R3,RZ	013	1	1
Reg/Reg	MOV R4,RZ	014	1	1
Reg/Reg	MOV R5,RZ	015	1	1
Reg/Reg	MOV R6,RZ	016	1	1
Reg/Reg	MOV R7,RZ	017	1	1

Reg/Reg	MOV R8,RZ	018	1	1
Reg/Reg	MOV R9,RZ	019	1	1
Reg/Reg	MOV R10,RZ	01A	1	1
Reg/Reg	MOV R11,RZ	01B	1	1
Reg/Reg	MOV R12,RZ	01C	1	1
Reg/Reg	MOV SP,RZ	01D	1	1
Reg/Reg	MOV RA,RZ	01E	1	1
Reg/Reg	MOV R2,R1	022	1	1
Reg/Reg	MOV R3,R1	023	1	1
Reg/Reg	MOV R4,R1	024	1	1
Reg/Reg	MOV R5,R1	025	1	1
Reg/Reg	MOV R6,R1	026	1	1
Reg/Reg	MOV R7,R1	027	1	1
Reg/Reg	MOV R8,R1	028	1	1
Reg/Reg	MOV R9,R1	029	1	1
Reg/Reg	MOV R10,R1	02A	1	1
Reg/Reg	MOV R11,R1	02B	1	1
Reg/Reg	MOV R12,R1	02C	1	1
Reg/Reg	MOV SP,R1	02D	1	1
Reg/Reg	MOV RA,R1	02E	1	1
Reg/Reg	MOV R1,R2	031	1	1
Reg/Reg	MOV R3,R2	033	1	1
Reg/Reg	MOV R4,R2	034	1	1
Reg/Reg	MOV R5,R2	035	1	1
Reg/Reg	MOV R6,R2	036	1	1
Reg/Reg	MOV R7,R2	037	1	1
Reg/Reg	MOV R8,R2	038	1	1
Reg/Reg	MOV R9,R2	039	1	1
Reg/Reg	MOV R10,R2	03A	1	1
Reg/Reg	MOV R11,R2	03B	1	1
Reg/Reg	MOV R12,R2	03C	1	1
Reg/Reg	MOV SP,R2	03D	1	1
Reg/Reg	MOV RA,R2	03E	1	1
Reg/Reg	MOV R1,R3	041	1	1
Reg/Reg	MOV R2,R3	042	1	1
Reg/Reg	MOV R4,R3	044	1	1
Reg/Reg	MOV R5,R3	045	1	1
Reg/Reg	MOV R6,R3	046	1	1
Reg/Reg	MOV R7,R3	047	1	1
Reg/Reg	MOV R8,R3	048	1	1
Reg/Reg	MOV R9,R3	049	1	1
Reg/Reg	MOV R10,R3	04A	1	1
Reg/Reg	MOV R11,R3	04B	1	1
Reg/Reg	MOV R12,R3	04C	1	1
Reg/Reg	MOV SP,R3	04D	1	1
Reg/Reg	MOV RA,R3	04E	1	1

NOP – No Operation

S	X	Z	C	N	V	L
-	-	-	-	-	-	-

Operation: No operation

Perform no operation. The CPU execution state (flags, registers, memory) not updated with the exception of PC Register, which is updated to the next instruction.

This is the only instruction which is hard-coded into the CPU. It is used with an instruction fetch is suppressed (such as on reset) and the Fetch Register and Instruction Register need to be cleared out. Pull-down resistors on the Fetch Bus will pull the default value to 0x0000 and allow for a NOP instruction to be artificially generated and loaded into these registers.

NOP is the only instruction which does not support conditional execution. The instruction is in the form of always executing. NOP can be considered to be NOP-AL, even though the NOP-AL mnemonic does not exist.

Addressing Mode	Assembly Language Form	Opcode	# Words	# Cycles
None	NOP	000	1	1