CHIPMONC 4-Bit CPU: Final Report

1. Olab Operation and Customization

We began by wiring the chip to the stimulus and analyzer connections on the breadboard. We attempted to wire the signals to a physical location on the stimulus and analyzer connections that was similar to their location on the chip. This made it easier for us to complete the wiring and to find certain signals on the stimulus and analyzer connectors. In order to ensure that the signals that appeared on "top" of the chip (the far side) also appeared on top of the cable connectors, we added some "dum" signals to our command file.

Once the chip had been wired and the breadboard connected to the OmniLab Tester, the Omnilab software was relatively easy to operate. We began by loading the ELEC423 testbench and adjusting it to fit our needs. Once we became more familiar and comfortable with Olab, we made several permanent adjustments to the ELEC423 testbench and saved it as our own: CHIPMONC. The CHIPMONC setup was customized from the original in several ways. Most noticeably, we ran the LINELABE macro using our data, so the waveforms were appropriately labelled with the names of our signals. This made it much easier to compare the Olab outputs with the outputs of our IRSIM simulation. We also rearranged the order of the signals to match the order of the IRSIM output; as one might imagine, this greatly improved the ease of comparison, as well.

In addition to the major customizations mentioned above, we also made a few minor changes in the ELEC423 testbench. Within the sampling window, we set the default magnitude to 1 and the default clock rate to 680 KHz. In the triggering window, we changed a few parameters in the Analyzer setup and adjusted the trigger signal to be our restart signal. Once these changes had been made and the CHIPMONC testbench created, we were able to open Olab, load our testbench, and immediately load the test vector file of choice. We then made sure the stimulus was "on" (a star was visible in the center) and set the trigger to once.

Printing caused some minor problems initially. When "print screen" was chosen from the system menu, we were prompted for text to print with our picture. This was a convenient method of noting what test vector was being run as well as which chip was being tested. However, we soon discovered that the text printed on an otherwise blank sheet of paper, and the screen printed on a second sheet. Using our engineering ingenuity, we were able to devise a method of quickly placing the correctly-oriented "text paper" in the manual slot of the printer so that the Olab screen would print right below the text. This is how the Olab results attached to this report were printed.

2. Test Vectors

Seventeen test vectors were developed to test the chip as a whole. Each vector was designed to test the performance of a particular component or signal within the chip. Among the functions tested by the vectors are the ALU operations, the branching function and corresponding flag, the data and instruction memory segmentation fault flags, the "end of program" flag (used to indicate that the end of the 10-location instruction memory has been reached), the LDX and ADDX functions (used for array manipulation and access), and the carry-out flag. A complete list of what each vector tests and the maximum speed at which the test is successful can be found in the Speed Response section of this report. In addition, the irsim and Olab output of each vector is attached.

Several of the vectors test the same (or very similar) functions of the chip. Rather than pushing the limits of monotony, we have chosen a few example test vectors to present in this section and describe in detail. Each detailed description gives an overview of the test vector's purpose and the command file itself. Most also contain a the Chipmonc assembly code for the vector

EXAMPLE PROGRAM ONE

The first vector tests for the correct operation of the DMSEG flag, which should be active when the controller attempts to access an address in data memory that doesn't exist. This program should output a data memory seg fault on the second instruction.

CHIPMONC ASSEMBLY

```
addi 1111
st 1000
addi 0000
halt
```

CMD FILE

```
|dmseg on store

clock p_clka 0 1 0 0
clock p_clkb 0 0 0 1

vector in p_in{3:0}
vector out p_out{3:0}
vector state p_sb{0:3}
vector PC p_pc{3:0}
vector IR p_ir{3:0}
vector bus p_bus{3:1}

ana in p_enter out p_restart state PC IR p_clka p_clkb p_zeroflag p_carryout p_dmseg
p_imseg p_loading p_branch p_halted bus

V p_restart 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0


V in 0000 0000 0000 1010 1111 1111 1111 0101 1000 1111 1111 1010 0000 1111 1111 0000
0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000
1111 1111 0000 0000 1111 1111 0000 0000 1111 1111
```

V p_enter 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

R

EXAMPLE PROGRAM TWO
This example program tests the “End of Program” Instruction (RIP), which indicates to the user that the end of the given program has been reached and the instruction memory should be loaded with new instructions.

CHIPMONC ASSEMBLY

First Block:
addi 0001
rip

Second Block:
addi 1111
ld 0010
not
add 0010
sr
wo
ld 0010
wo
halt

CMD FILE

| rip and load prog1

clock p_clka 0 1 0 0
clock p_clkb 0 0 0 1

vector in p_in{3:0}
vector out p_out{3:0}
vector state p_sb{0:3}
vector PC p_pc{3:0}
vector IR p_ir{3:0}
vector bus p_bus{3:1}

ana in p_enter out p_restart state PC IR p_clka p_clkb p_zeroflag p_carryout p_dmseg p_imseg p_loading p_branch p_halted bus

V p_restart 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

V in 0000 0000 0000 1010 0001 1111 1111 0001 0000 1111 1111 1010 1111 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111

V p_enter 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1

R

|run the first block
V
V p_restart 0
V p_enter 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0
R

|load and run next block
V p_restart 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

V in 0000 0000 0000 1010 0111 1111 1111 0101 0010 1111 1111 1101 0000 1111 1111 1110 0010 1111 1111 1011 0000 1111 1111 0010 0000 1111 1111 0100 0010 1111 1111 0010 0000 1111 1111 0000 0000 1111 1111

V p_enter 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1

R

EXAMPLE PROGRAM THREE

The third example vector tests the functioning of the LDX and ADDX instructions. These commands are used to access specific elements in arrays. LDX loads the indexing register with the value in the TP and is used to index arrays. ADDX is analogous to ADD except that it fetches the value pointed to by the IX register rather than by the TP register. Since the IX register is not overwritten when a new instruction is fetched, it can be used to keep track of the next index within an array.

Command File

| test ldx and addx

| write out 1111

clock p_clka 0 1 0 0
clock p_clkb 0 0 0 1
vector in p_in{3:0}
vector out p_out{3:0}
vector state p_sb{0:3}
vector PC p_pc{3:0}
vector IR p_ir{3:0}
vector bus p_bus{3:1}

ana in p_enter out p_restart state PC IR p_clka p_clkb p_zeroflag p_carryout p_dmseg p_imseg p_loading p_restart p_enter p_branch p_halted bus

V p_RESTART 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

V in 0000 0000 0000 1010 0101 1111 1111 0101 0000 1111 1111 0111 0000 1111 1111 1010 1010 1111 1111 0101 0101 1111 1111 1001 0000 1111 1111 1000 0000 1111 1111 0010 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111

V p_enter 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1

R

EXAMPLE PROGRAM FOUR

This program tests the functionality of the instruction memory segmentation fault flag, which is asserted when the controller attempts to access a location that does not exist in instruction memory. The program branches to address 12, which does not exist, so IMSEG should be asserted.

CHIPMONC ASSEMBLY

addi 0101
bnez 1100

CMD FILE

| branch to 13 for IMSEG

clock p_clka 0 1 0 0
clock p_clkb 0 0 0 1

vector in p_in{3:0}
vector out p_out{3:0}
vector state p_sb{0:3}

vector PC p_pc{3:0}
vector IR p_ir{3:0}
vector bus p_bus{3:1}

ana in p_enter out p_restart state PC IR p_clka p_clkb p_zeroflag p_carryout p_dmseg p_imseg p_loading p_branch p_halted bus

V p_RESTART 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

V in 0000 0000 0000 1010 0101 1111 1111 0011 1100 1111 1111 0000 0000 1111 1111 0011 0001 1111 1111 1111 0111 1111 1111 0011 0010 1111 1111 0000 0000 1111 1111 0010 0000 1111 1111 1100 1010 1111 1111 0010 0000 1111 1111

V p_enter 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1

R

EXAMPLE PROGRAM FIVE
Vector five tests the chip by creating an infinite loop in which the controller should branch continusly.

CHIPMONC ASSEMBLY
addi 1000
bnez 0000 ;pc <-pc+0

CMD FILE

| infinite loop, branch to same address

clock p_clka 0 1 0 0
clock p_clkb 0 0 0 1

vector in p_in{3:0}
vector out p_out{3:0}
vector state p_sb{0:3}
vector PC p_pc{3:0}
vector IR p_ir{3:0}
vector bus p_bus{3:1}

ana in p_enter out p_restart state PC IR p_clka p_clkb p_zeroflag p_carryout p_dmseg p_imseg p_loading p_branch p_halted bus

V p_RESTART 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0

V in 1111 1111 1111 1010 0100 1111 1111 0011 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111

V p_enter 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1

R

EXAMPLE PROGRAM SIX
The final vector tests all the possible operations executed within the ALU.

Command File

| test all aluops

|out =0011 co goes high near end and beginning (sub op)

clock p_clka 0 1 0 0
clock p_clkb 0 0 0 1
vector in p_in{3:0}
vector out p_out{3:0}
vector state p_sb{0:3}
vector PC p_pc{3:0}
vector IR p_ir{3:0}
vector bus p_bus{3:1}

|enter first block

V p_RESTART 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0

V in 1111 1111 1111 1010 1111 1111 1111 0101 0000 1111 1111 1001 0000 1111 1111 1010 1100 1111 1111 0101 0011 1111 1111 0001 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111

V p_enter 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1

R

|run first block

V p_enter 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

v p_restart 0

R

|enter next block

V p_RESTART 0

V

V in 1111 1111 1111 1100 1111 1111 1111 1111 0000 1111 1111 1101 0000 1111 1111 1011 1100 1111 1111 1100 0011 1111 1111 1110 0011 1111 1111 0010 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111

V p_enter 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1

R

|run next block

V

V p_enter 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

v p_restart 0

R

3. Testing Methodology and Results

We ran all the vectors described above on our first test chip. All outputs matched those of irsim runs, except for an occasional problem with bus3 (S-A-1). When we jiggled the interconnection wires a bit, the problem was resolved. Therefore, we attribute this to a wiring flaw on our part and deemed the chip acceptable.

Because of the large number of test vectors we have, we decided to not test the next chips so thor oughly, and only ran about 1 / 3 of our total test set. All the chips were deemed acceptable (by this less-rigorous test set) except for one, giving us a yield of 80%. Chip 3 failed in one output bit. IMSEG is a flag of an attempt to access an Instruction Memory location that does not exist. However, IMSEG remained low even on test vectors that accessed illegal addresses. All other bits worked as expected.

Since only one bit was faulty and all other chips were correct in this bit, we theorize that the problem is not in the mask but in the bonding or wafer. A mask problem would likely have been detected in other chips as well, whereas a bonding mistake or wafer defect could affect only one. Additionally, since a wafer defect would probably affect more than a single bit, it seems likeliest that the flaw is in the bonding of the output pin.

4. Verification of Results

We ran a total of 17 programs to verify that the OLAB results matched with the irsim outputs. Please refer to the attached irsim and OLAB outputs for details. All OLAB results matched with the irsim ones. We tested for the different functions of CHIPMONC. First, we verified that all ALU operations in the manufactured chip work as expected. The programs that we used to test the different ALU operations are programs 1, 4, 5, 6, 12, and 14. In all of the testing, the output waveforms of OLAB were correct. Other functions that we tested include branching (programs 2, 10, 13), DMSEG on different instructions (programs 3, 7, 8, 15, 16), the “rip” instruction (program 4), loading and adding register X (program 9), infinite looping and branching (program 11), and IMSEG (program 17). Four out of the five CHIPMONC chips passed all the above testing.

5. Speed Response of CHIPMONC

Each of the 17 test vectors was run on the CHIPMONC CPU to determine the maximum operating speed of the chip. The results were not unanimous--not all programs were successful up to the same maximum frequency. The maximum frequency for each of the test programs is shown in the table below.

	Function Tested	Maximum Speed
1.	ALU Operations	7 MHz
2.	Branching Function and Flag	34 MHz
3.	Data Memory Seg Fault on STORE Instruction	34 MHz
4.	ALU Operations and “End of Program” Flag	17 MHz

5.	ALU Operations	17 MHz
6.	ALU Operations: Shift Right and Subtract	34 MHz
7.	Data Memory Seg Fault on STORE Instruction	34 MHz
8.	Data Memory Seg Fault on LOAD Instruction	34 MHz
9.	LOADX and ADDX Functions (used for array manipulation)	17 MHz
10.	Branching Function and Instruction Memory Seg Fault	34 MHz
11.	Infinite Branching Loop	34 Mhz
12.	ALU Operations that Activate Carry-Out Flag	34 Mhz
13.	Branch After Change in Accumulator Value	34 MHz
14.	All ALU Operations	17 MHz
15.	Data Memory Seg Fault on a LOADX Instruction	34 MHz
16.	Data Memory Seg Fault on an ADDX Instruction	34 MHz
17.	Instruction Memory Seg Fault when “End of Program” Instruction is not Included	34 MHz

As one can see by viewing the above table, all test programs were successful through at least 17 MHz, and most were successful through 34 MHz. (All frequency values in this section refer to testing with a 4 bit clock.) Due to the limitations of Olab, we were unable to test at frequencies between 17 MHz and 34 MHz, and we were unable to test at frequencies greater than 34 MHz. Thus, it is likely that the programs which show maximum frequencies of 17 MHz actually have a somewhat higher maximum frequency, and the programs which show maximum frequencies of 34 MHz may have a much higher true maximum frequency.

We are happy to say that every test performed above the expected maximum clock frequency calculated last fall. Based on the fact that spice gave a minimum clock period of 55 ns, we calculated an estimated maximum clock frequency of 1/55 ns = 18.2 MHz. However, engineering design requires a margin of error in which the performance of the design can fluctuate, so we reduced this expected value by half to 9 MHz. As seen above, the chip performed all of the tests successfully at a speed of 17 MHz, which very clearly exceeds the 9 MHz estimated maximum. In fact, the 17 MHz value comes very close to the original (spice-determined) maximum-clock frequency estimation of 18.2 MHz. Recalling that we were unable to test at frequencies in the 20 MHz range, it is highly possible that the circuit actually meets this 18.2 MHz cutoff. In any case, we can say with confidence that CHIPMONC will perform successfully at a maximum clock frequency of 17 MHz.

Our spice modeling predicted that the longest path would be through the ALU, and our test vector results support this hypothesis. In viewing the table above, it is easy to notice that the vectors that performed at a maximum speed of 17 MHz instead of 34 MHz were those that involved testing of the ALU operations. There were two test vectors (numbers 6 and 12) that included ALU operations and still performed at the 34 MHz speed, but this seems to be less common. We can infer that these tests did not "make use" of the longest paths through the ALU and thus were successful at the faster rate.

6. "Burn" Test

We ran an additional test to see how our chip would respond to the rigors of running for an extended period of time. We left a functional chip running an infinite loop program overnight for a total of 14 hours and about 60,000 processor cycles. We then ran some test vectors on the chip to see if it had been damaged. All vectors matched the expected output. We therefore conclude that continuous, extended use does not harm the chip. We are also pleased to report that, as far as we can tell, the extended use didn't damage the test equipment, either.

7. Suggestions

The thing that gave us the most trouble was getting OLAB to display signals, due to the fact that we had neglected to load the correct benchtop. We assumed that default.bch or last.bch, whichever was loaded when we ran OLAB, would work and passed many frustrating hours until we realized we needed to explicitly load elec423.bch. Our suggestion is to either make this the default or state more explicit in the instructions that this step is necessary. Also, the lab assistant or professor should keep in mind that the lack of the appropriate benchtop causes no signals to display and include that as a possible solution when approached with the problem.