Freddy Chen 997363124
Anthony Alayo 997487401

ECE552 Lab 3 Superscalar Lab

Section 2.1 Explore the impact of Issue Width on performance

1. Fill in the following table with the measured CPI numbers. Also, provide the %CPI reduction over a 1-wide superscalar machine in parentheses for comparison.

Benchmarks	1-wide SS	2-wide SS	4-wide SS	8-wide SS
gcc.eio	1.9503	1.7364 (10.97%)	1.6702 (14.36%)	1.6702 (14.36%)
compress.eio	1.4870	1.3217 (11.11%)	1.2835 (13.69%)	1.2835 (13.69%)
go.eio	1.8770	1.6526 (11.96%)	1.5770 (15.98%)	1.5770 (15.98%)

2. Is there a cut-off point, where wider machines do not yield any performance benefits?
Yes, from 4-wide to 8-wide, we do not see any performance benefits.

3. No CPI reduction can signify that (a) there is no more available ILP to be extracted from the simulated program or that (b) our current hardware configuration is limiting the ILP we can extract. Here the second reason is the valid one: one sim-outorder configuration parameter did not scale properly with the machine-width increase.

(a) Identify which this configuration parameter cong-param1 is. Briefly explain your thought process and any tests you ran.
cong-param1 is **-fetch:ifqsize**, or the number of instructions that gets fetched. This did not scale properly with the machine-width increase because there won't be enough instructions in the dispatch queue to utilize the machine-width increase.

We ran the simulation from part 1 with the additional flag **-fetch:ifqsize 8** and we got a decrease in CPI from 4-wide SS (1.6702) to 8-wide SS (1.6034).

(b) Provide updated statistics for Table 1.
Here are updated statistics with a fetch width of 8 (increased from default of 4).

Benchmarks	1-wide SS	2-wide SS	4-wide SS	8-wide SS
gcc.eio	1.9503	1.7364 (10.97%)	1.6702 (14.36%)	1.6034 (17.79%)
compress.eio	1.4870	1.3217 (11.11%)	1.2835 (13.69%)	1.2830 (13.72%)
go.eio	1.8770	1.6526 (11.96%)	1.5770 (15.98%)	1.4914 (20.54%)

4. What would be a good ratio of config-param1 with respect to issue-width? Please explain.
A good ratio of fetch width to issue with would be 1:1, because the issue widths would not be utilized to its full extent it there aren't enough instructions in the dispatch queue.

5. Now take a step back and think of what would be the ideal CPI of an m-way superscalar processor. Ideal CPI is: ____. How does the measured CPI compare with the ideal CPI? Why the discrepancy, if any?
The ideal CPI of an m-way superscalar processor is 1/m.

The measured CPI is always going to be less because of hazards that arise from data dependencies, the amount of functional units available (structural hazards), and instructions that get flushed due to branch mispredictions.

6. Write a small microbenchmark (10-20 lines of code) that approximates this ideal CPI. Present the results and justify why you selected the specific sequence of instructions.
The results of a microbenchmark that consists of a for loop of independent instructions written in assembly:

Issue width	CPI of microbenchmark (with loop unrolling)
1	1.0002
2	0.5226
4	0.2838
8	0.2241

We selected the specific sequence of 8 independent instructions within the for loop so that the microbenchmark can take advantage of the issue-width of the processor as it changes from 1 to 8 through powers of 2. The loop in the microbenchmark is unrolled so that the processor can make better use of its wide fetch.
The independent instructions are all assembly instructions to move 0 into a different registers. The instructions are independent to minimize any data dependence hazards which can increase the CPI due to stalls. The for loop iterates a million times to ensure that the branch predictor can learn the pattern of the loop. The CPI's are above the ideal values because of the data dependency hazards that arise from evaluating the branch condition at the end of the loop.

7. Is this microbenchmark code typical of the programs you usually write? Explain why it is or why it is not typical; identify common program characteristics that are either present or lacking.
The microbenchmark code is not typical of programs we usually write because there are minimal data dependencies between instructions. Typically, programs take an input and apply operations on it and use the intermediate values to generate some sort of output.

Section 2.2 Explore the impact of Functional Units

1. Measure the impact of the number of functional units on CPI for different machine-widths (i.e., remember to use the same value for the decode, issue and commit widths). Report your findings in Tables 3 and 4 and plot the results for the integer functional units. At any given point, modify the count of a single functional unit type (e.g., Int-Cfg) while keeping the other three functional unit counts unchanged (set to the defaults).

Machine-Width	Benchmarks	Integer ALUs		
		Int-Cfg1 (2 ialus)	Int-Cfg2 (4 ialus)	Int-Cfg3 (8 ialus)
2	gcc.eio	1.7364	1.7364	1.7364
	compress.eio	1.3217	1.3217	1.3217
	go.eio	1.6526	1.6526	1.6526
4	gcc.eio	1.692	1.6702	1.6702
	compress.eio	1.3037	1.2835	1.2835
	go.eio	1.5872	1.577	1.577
8	gcc.eio	1.6271	1.6034	1.6016
	compress.eio	1.3035	1.283	1.2826
	go.eio	1.5028	1.4914	1.4904

Freddy Chen 997363124
Anthony Alayo 997487401

2. Justify the collected results from the previous question. You can modify sim-outorder to get a breakdown of instructions per functional unit requirement class according to Table 2.
As the amount of integer ALU units increase to match the issue width of the pipeline, CPI improves because more instructions can utilize these functional units at the same time.

3. After analyzing your findings, can you identify a meaningful ratio of number of functional units to machine width? What are the determining factors?
It seems like a 1:1 ratio of number of functional units to machine width for integer ALU units work well. We compared the CPI across different issue widths and integer ALU units as the determining factor for deciding what a good ratio is. If there are less functional units, CPI increases, likely due to stalls used to alleviate structural hazards. If there are too many functional units, we don't see a performance gain because they are underutilized. Floating point and load/store operations probably don't need to have a 1:1 ratio with the machine width because there are statistically less of these instructions in programs.

4. Identify two negative side-effects of having more functional units. Explain your answer.
Having more functional units results in more complex stall logic and bypass logic (O(N^2) for N-wide issue). The bypass network is very expensive because it takes more area to accommodate the wires required to forward values to previous stages, and each wire is 32 or 64 bit wide. If these functional units are not being utilized properly, then the processor becomes underutilized and the cost to implement the extra functional units go to waste.

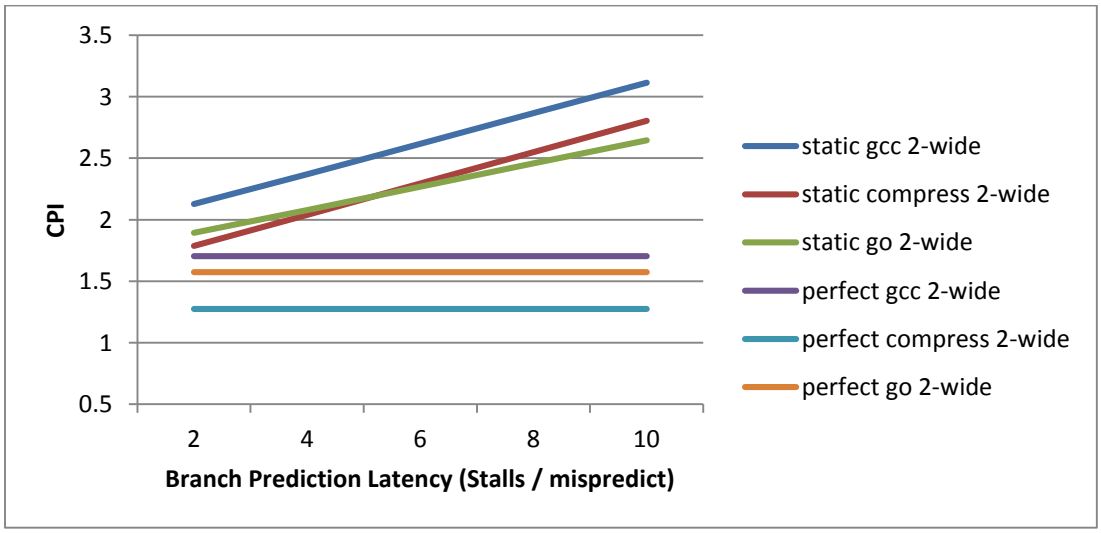
Section 2.3 Branch Prediction in superscalar processes

Static Always Not Taken CPIs

Machine-Width	Benchmarks	Latency				
		2	4	6	8	10
2	gcc.eio	2.1288	2.3703	2.6179	2.8657	3.1134
	compress.eio	1.7858	2.0403	2.295	2.55	2.8053
	go.eio	1.8927	2.078	2.2675	2.4571	2.6467

Perfect Prediction CPIs

Machine-Width	Benchmarks	Latency				
		2	4	6	8	10
2	gcc.eio	1.7041	1.7041	1.7041	1.7041	1.7041
	compress.eio	1.2758	1.2758	1.2758	1.2758	1.2758
	go.eio	1.5738	1.5738	1.5738	1.5738	1.5738



Freddy Chen 997363124
Anthony Alayo 997487401

1. Select one static branch prediction scheme and explore the impact of branch misprediction latency on CPI for a 2-wide machine. Plot the measured CPI statistics versus the branch misprediction latency. In the same graph also plot the CPI achieved with a perfect branch predictor. Discuss your findings.
For our static branch prediction scheme, we chose always not taken. According to our plot, as branch misprediction latency increases for a 2-wide machine, the CPI also increases. The CPIs for perfect predictions stay the same across latency changes because the CPU isn't paying any stall penalties for mispredictions.

CPIs of Dynamic 2-level prediction scheme with varying first level size

Machine-Width	Benchmarks	Size of L1		
		1	32	128
2	gcc.eio	1.8733	1.8565	1.8076
	compress.eio	1.3504	1.2903	1.2838
	go.eio	1.7186	1.7236	1.6997
4	gcc.eio	1.8097	1.7924	1.7431
	compress.eio	1.3189	1.2514	1.2449
	go.eio	1.647	1.6512	1.6265

2. Select one dynamic branch prediction scheme and explore the impact of its configuration parameters on CPI for a 2-wide machine. For example, if you select the 2-level branch predictor, you can choose to change the number of entries in the first-level table. Your exploration should include at least three distinct values for the chosen configuration parameter, and you can use the default misprediction latency. Report your findings in a table.
For our dynamic branch prediction scheme, we chose the 2-level predictor and explored the effects of altering the size of the first-level table. From our findings, we can see that as the size of the L1 table increases, our CPI decreases, most likely due to less stall penalties being paid for mispredictions.

3. Repeat the previous experiment for a 4-wide machine. Report your findings and justify the differences.
Repeating the previous experiment for a 4-wide machine produced a similar result pattern, where we see a decrease in CPI as the size of the L1 table increases. We also see that, as expected, the CPIs for the 4-wide machine are all lower than the CPIs for the 2-wide machine.

Section 2.4 Impact of Load/Store Queue Size

1. Modify the load/store queue size for different machine-widths and report your findings (i.e., CPI results).

CPI when varying load/store queue size

Machine-Width	Benchmarks	Size of LSQ			
		2	4	8	16
2	gcc.eio	1.8205	1.8183	1.8181	1.8181
	compress.eio	1.3467	1.331	1.3308	1.3308
	go.eio	1.7508	1.7495	1.7494	1.7494
4	gcc.eio	1.6726	1.6705	1.6702	1.6702
	compress.eio	1.302	1.2837	1.2835	1.2835
	go.eio	1.5784	1.577	1.577	1.577
8	gcc.eio	1.6055	1.6036	1.6034	1.6034
	compress.eio	1.3015	1.2832	1.283	1.283
	go.eio	1.4928	1.4915	1.4914	1.4914

Looking at our results, we can see that the CPI doesn't seem to be affected much when we vary the size of the load/store queue. When we examined the lsq_full parameter for each of our configurations, we saw that the fraction of the time the LSQ was full was less than 1%. This matches our CPI findings as we don't see much change with this parameter.

2. When will the load/store queue size become the bottleneck? Reason about its relationship with machine-width and the program's instruction mix.
The load/store queue size will become the bottleneck when the instruction mix has many loads and stores, and the machine-width is larger than the LSQ size. With a setup like this, the queue would be filled faster than it can be emptied and thus becoming the bottleneck.

Section 2.5 Compiler Optimizations

1. Provide assembly snippets from the two objdump files corresponding to the inner for-loop. Explicitly identify the unrolled loop sections, and comment on the compiler modifications. You can find a similar example in Figure 3.12 of your textbook [1].

First Code Snippet - Unoptimized

```
400238:   28 00 00 00   lw $2,0($3)          // this is a[i]
40023c:   00 00 02 03
400240:   42 00 00 00   addu $2,$2,$4        // this is a[i] = a[i]+i
400244:   00 02 04 02
400248:   34 00 00 00   sw $2,0($3)          // this saves a[i]
40024c:   00 00 02 03
400250:   43 00 00 00   addiu $3,$3,4        // this calcuates the location of a[i+1]
400254:   04 00 03 03
400258:   43 00 00 00   addiu $4,$4,1        // this is i = i+1
40025c:   01 00 04 04
400260:   5c 00 00 00   slti $2,$4,5         //this is used for branch condition
400264:   05 00 02 04
400268:   06 00 00 00   bne $2,$0,400238 <main+0x48>      // this is a loop exit condition
```

Second Code Snippet - Optimized

```
400228:   28 00 00 00   lw $7,0($3)          // this loads a[0]
40022c:   00 00 07 03                        // loop unrolling beginning here
400230:   28 00 00 00   lw $2,4($3)          // this loads a[1]
…….                                          // this load a[2]..a[4]
……..
40024c:   10 00 06 03
400250:   43 00 00 00   addiu $2,$2,1        // this calculates a[1] + 1
400254:   01 00 02 02
400258:   43 00 00 00   addiu $4,$4,2        // this calculates a[2] + 2
……..                                        // this does a[3] + 3, a [4] + 4
……..
40026c:   04 00 06 06
400270:   34 00 00 00   sw $7,0($3)          // this does  a[0] = a[0] + 0
400274:   00 00 07 03
400278:   34 00 00 00   sw $2,4($3)          // this does a[1] = a[1] + 1
……..                                        // this does a[2..4] = a[2..4] + 2..4
……..
400294:   10 00 06 03                        // here is outer loop unrolling
400298:   28 00 00 00   lw $7,0($3)          // This is another iteration of the above
40029c:   00 00 07 03                        // this is unrolling outer loop since  j = 0,1
4002a0:   28 00 00 00   lw $2,4($3)          // repeated...
……..
……..
```

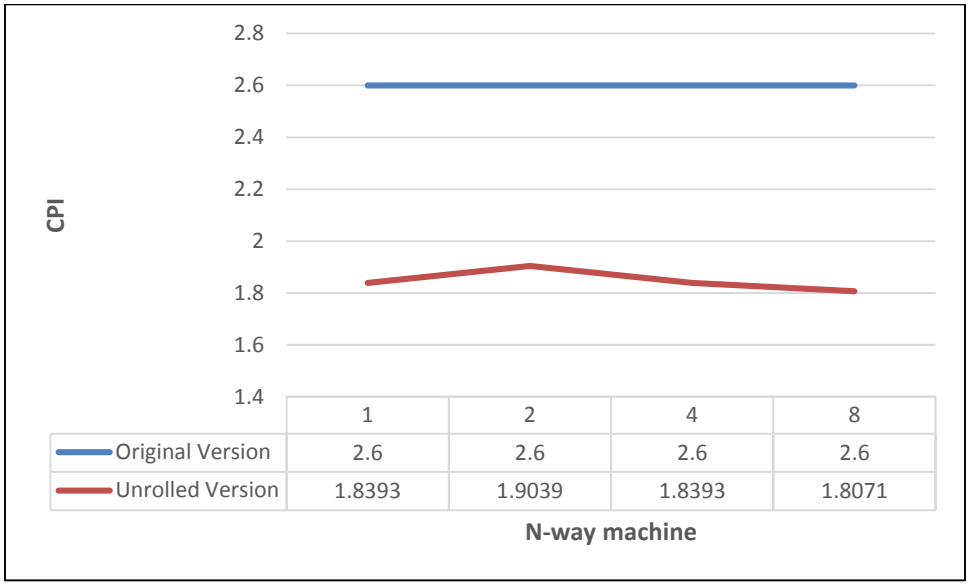
Freddy Chen 997363124
Anthony Alayo 997487401

As expected, when using the unroll loop flag, the compiler unrolls all iterations of the inner loop as seen in the assembly code. This increases the amount of ILP that can be extracted from our program because it increases the size of the basic block within the loop, which allows the CPU to do a wider fetch within the loop. This will directly affect our CPI, making it smaller.

2. Compare the performance (CPI) of the loops.c microbenchmark with and without loop-unrolling. Are the results as expected?
Original CPI: 2.6000
Unrolled CPI: 1.8393
The results are as expected. The unrolled version has less branches than the original due to the inner loop being unrolled, which implies a more effective fetching of instructions, which leads to better ILP, which gives us a lower CPI.

3. What is the effect of the -bpred nottaken configuration option? HINT: Look at the branch instructions in your two loops.
Number of branches in original version: 6,000,000
Number of branches in unrolled version: 500,000
Since we are using the not taken static branch predictor, the predictions are mostly incorrect. This would imply a large CPI penalty as we have many branches in our code. The original version will be hurt by this the most as it has 6,000,000~ branches. The unrolled version has an order of magnitude less, and so its CPI isn't hurt by ILP loss as much as the original version.

4. Repeat the previous experiment, but for different machine-widths. Plot your results.



Freddy Chen 997363124
Anthony Alayo 997487401

Section 2.6 Using McPat

1. Report area, dynamic power and leakage for the aforementioned components for different machine- widths. You can omit components not impacted by machine-width, but you need to justify your choice. Considering your previous performance, area and power measurements, what width machine would you build and why?

Instruction Fetch Unit				
Machine Width	1	2	4	6
Area (mm^2)	4.73534	5.89917	8.22682	10.5545
Peak Dynamic Power (W)	3.2594	4.58905	7.24836	9.90768
Subthreshold Leakage (W)	0.403005	0.425821	0.471454	0.51709
Gate Leakage (W)	0.056879	0.067233	0.087941	0.10865
Runtime Dynamic Power (W)	6.4075	8.19023	11.7557	15.3212

Execution Unit				
Machine Width	1	2	4	6
Area (mm^2)	11.8032	12.0532	12.7194	13.5253
Peak Dynamic Power (W)	8.19692	9.32527	11.6716	14.0704
Subthreshold Leakage (W)	1.207	1.23021	1.27913	1.3283
Gate Leakage (W)	0.224083	0.234504	0.255788	0.27712
Runtime Dynamic Power (W)	14.1713	15.9711	19.8283	23.6854

Integer RF				
Machine Width	1	2	4	6
Peak Dynamic Power (W)	0.202026	0.404053	0.808105	1.21216

Floating Point RF				
Machine Width	1	2	4	6
Area (mm^2)	0.081116	0.136319	0.408745	0.82095
Peak Dynamic Power (W)	0.018745	0.052701	0.158108	0.31597
Subthreshold Leakage (W)	0.000492	0.000847	0.001656	0.00271
Gate Leakage (W)	6.48E-05	0.000123	0.000253	0.00043
Runtime Dynamic Power (W)	0.029755	0.041826	0.062742	0.08359

In the above tables I only displayed the fields that changed with regards to each component. Here are the explanations for parameters that did not change:

- For the Integer RF, only the peak dynamic power changed because we are not adding more registers with the increase in machine width.
- The renaming unit did not change because no new registers were added.
- The load/store unit and memory management unit did not change as we increased the machine width because they are not dependent on the machine width, they are dependent on the amount of functional units.
- The L2 and L3 caches did not change because they are independent of machine width.

Freddy Chen 997363124
Anthony Alayo 997487401

Looking at the performance, area, and power measurements, we can see that the ALU components increase these parameters linearly. For instruction fetch and execution components, it increases at a rate a little less than linear. This first tells us that we are definitely paying a price for adding hardware and we should be utilizing it if we are adding it.

Now we can say that determining which machine width to build from these numbers alone is not a good decision. We need to be able to know how much ILP we can extract, giving us a better CPI, and thus making our increase in hardware worth the cost. I would benchmark the CPU with increasing machine widths until my CPI no longer decreases. I would then use that machine width.

2. Select a reasonable machine width and for that modify the number of ALUs (i.e., parameter ALU per core). Identify the metrics/components this modification affects and explain the trend. Do these results change your answer to Question 4 from Section 2.2? Why/why not?
For this question we chose to change the number of ALUs for a 4 wide machine. Changing the number of ALUs affected the area, peak dynamic power, subthreshold leakage, gate leakage and results broadcast bus. The trend is as expected; an increase in integer ALUs brings an increase in all the parameters affected.

It seems that the area, peak dynamic power, subthreshold leakage and gate leakage scaled linearly while the results broadcast bus scaled expontentially. The linear scaling implies that each ALU is independent of other components. The results broadcast bus approaches an O(n^2) scaling which makes sense as the results must be broadcasted to all other components.

These results don’t change our answer to question 4 from section 2.2 as we discussed an O(n^2) complexity increase for stall/bypass logic that increases area dramatically.