Karl Kirch
CS 5113

## Performance Analysis of ISAs

The 4 Instruction Set Architectures that we were asked to compare are very similar in terms of instructions related to code flow (branching, goto, etc...). However, because of their different makeup when it comes to arithmetic operations, there are some serious differences in how the different ISAs perform both in memory access, ability to perform optimizations, and code size. The compilers that I wrote in a lot of cases demonstrate these innate properties related to their target ISA but due to lack of optimizations some ISAs (particularly the load and store architecture) do not match the expected results.

### CPI performance of each ISA

Lets first look at the raw performance of each ISA. For this performance metric I used the average CPI x the instruction count to get the total cost of each program.

The CPI list for the following metrics was :
ADD,10
SUB,10
MULT,10
DIV,10
PUSH,20
POP,20
STORE,40
LOAD,35
GOTO,5
BLT,8
BGT,8
BLE,9
BGE,9
BEQ,5
BNE,5
CLEAR,2

Comparing a memory intensive (medium) program, I observed the following data:

| ISA | Average CPI | Total Cost |
|---|---|---|
| Memory Memory | **37** | **3265** |
| Stack | 19 | 3575 |
| Load Store | 36 | **6565** |
| Accumulator | **17** | 4485 |

This shows that we the least cpu cost comes from the Memory Memory ISA and the Stack ISA. This is consistent with what we would expect as they would have the least amount of instructions. Interestingly, while the Load and Store ISA had a similar low instruction count to the Stack ISA, it's

average CPI eventually made it have the highest total cost.

When we compare a cpu intensive (medium) program, we get slightly different results:

| ISA | Average CPI | Total Cost |
|---|---|---|
| Memory Memory | **11** | **815** |
| Stack | 15 | 4999 |
| Load Store | **34** | **7215** |
| Accumulator | 15 | 4387 |

In this simulation, we see that the Memory Memory ISA is the clear winner which is expected as it would have a lot less instructions to perform as there are no load or store instructions to perform. We also see the simplicity of the Accumulator and Stack ISAs show as well with their relatively low total cost. Again we see the Load and Store ISA come in last. This is a bit misleading however because the compiler is not doing any code optimizations. Theoretically, the Load and Store ISAs cost should be much lower as many load and store instructions would be removed from the optimized assembly. This could be accomplished using dynamic register allocation, but as that wasn't necessarily part of the assignment, it didn't get included in the compiler. In fact the Load and Store compiler is completely un-optimized to the point that it only uses at most 3 registers and always reloads data fresh into each needed register.

**Memory Usage of each ISA**

Another metric we can look at is the memory usage and memory access of each ISA. For these numbers we're again using the same CPI and the memory intensive (medium) program.

| ISA | Memory Access Count | Memory Access (in bytes) |
|---|---|---|
| Memory Memory | **178** | **712** |
| Stack | **178** | **712** |
| Load Store | **178** | **712** |
| Accumulator | **178** | **712** |

Interestingly, despite the fact that the numbers look wrong, this is almost what we should expect to see. Memory Memory, Stack, and Accumulator ISAs all should have the same memory access usage. The one erroneous number is the Load and Store ISA memory usage. This is because again, the Load and Store compiler target is horribly un-optimized. With dynamic register allocation we could be looking at at a much smaller usage.

With the cpu intensive program, we see the same results as the memory intensive program (table omitted) .

Karl Kirch
CS 5113

**Code size of each ISA**

Another important aspect to look at is the generated code size. The calculated code sizes were consistent with what we'd expect to see theoretically.

For the memory intensive (medium) program, the following data was observed.

| ISA | Instructions | Code Size (in bytes) |
|---|---|---|
| Memory Memory | **88** | **444** |
| Stack | 188 | 544 |
| Load Store | 178 | **884** |
| Accumulator | **263** | 619 |

We expect that the Memory Memory ISA target will generate the smallest number of instructions and code because it doesn't have to load/store memory in cases of arithmetic instructions. The Load and Store ISA does remarkably well in the instruction count department considering that it is horribly un-optimized. However because each instruction almost always has 1-3 memory addresses associated with it, it's total code size balloons up to be the worst option.

With the cpu intensive program, we see much the same results.

| ISA | Instructions | Code Size (in bytes) |
|---|---|---|
| Memory Memory | **74** | **500** |
| Stack | **333** | 759 |
| Load Store | 210 | **1076** |
| Accumulator | 284 | 710 |

The difference here is that stack based arithmetic operations require more instructions to perform because of the extra push/pop commands that are required to do arithmetic operations.

**Memory Bandwidth of each ISA**

Lastly we can look at the total memory bandwidth that is used for each ISA. The total memory bandwidth is the combination of the code size and the memory access bandwith.

For the memory intensive program, the following was observed.

| ISA | Total Memory Bandwidth |
|---|---|
| Memory Memory | **1156** |
| Stack | 1256 |
| Load Store | **1596** |

| | |
|---|---|
| Accumulator | 1331 |

As can be seen, the Load and Store ISA compiler generates the most memory usage of the different ISAs. This is very erroneous as this compiler target is very unoptimized. Theoretically we'd see the Load and Store ISA having the best memory bandwidth (which is why it tends to be used in todays cpu hardware).

For the cpu intensive program we get much the same erroneous picture.

| ISA | Total Memory Bandwidth |
|---|---|
| Memory Memory | **1276** |
| Stack | 1535 |
| Load Store | **1852** |
| Accumulator | 1486 |

Again, this result is skewed by the fact that the Load and Store compiler doesn't have any assembly level optimizations.

**Summary**
Overall each ISA provides advantages and disadvantages in theoretical and real world use. The Memory Memory ISA produces dense code with low cpu cost. However it is hard to implement in hardware in a cost effective way. Stack and Accumulator ISAs provide a simple to use ISA with good memory usage and low CPI, but they can prove to be costly when it comes to instruction counts. Also they tend to be hard to optimize for because resources must be stored in the cpu and cannot be pipelined easily. Lastly the Load and Store ISA certainly seems to be a bad choice when it comes to ISAs, but both theoretically and in practice, the Load and Store ISA allows great optimization gains (when properly optimized and via pipelining) and allows simpler (ie cheaper) cpus to be manufactured.