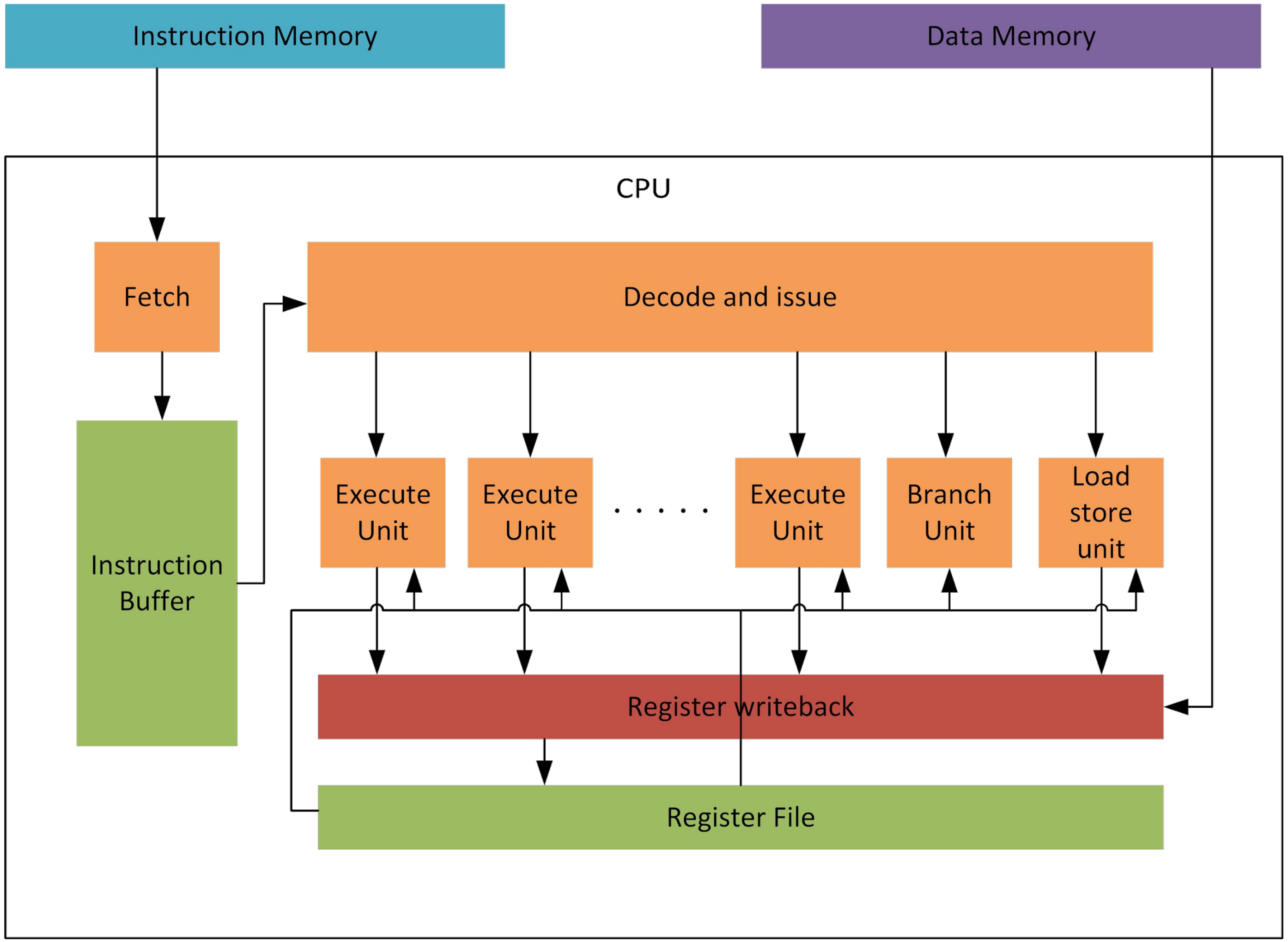
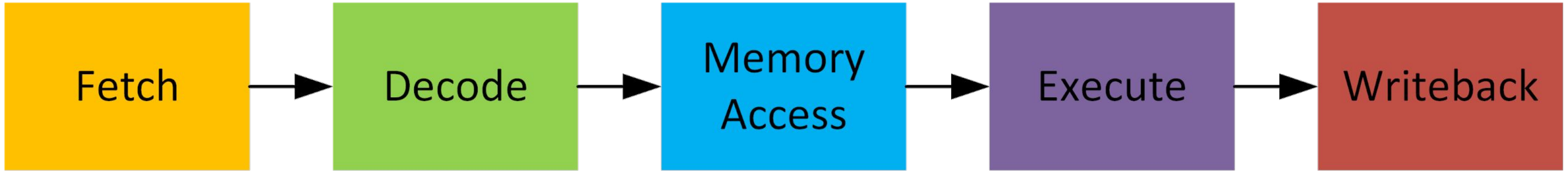
Advanced Computer Architecture

Alex Harman


Instruction Memory
Data Memory
CPU
Fetch
Decode and issue
Instruction Buffer
Execute Unit
Execute Unit
. . . . .
Execute Unit
Branch Unit
Load store unit
Register writeback
Register File

My Architecture

- N execute units
- Out of order
- 1 load/store unit
- 1 branch unit
- 5-deep pipeline



Simulator options

- `-a` Assemble program
- `-e` Number of execution units to use
- `-b` Size of the instruction buffer
- `-f` File to use
- `-d` Debug mode, writing the state every iteration
- `-i` Interactive mode, steps through cycle by cycle and displays the state
- `-h` Help, including a full list of options

There are several other options used to tweak the debug output.

Example Program:

Assembly:

```
MEM     0 0
MEM     1 0
MEM     2 0xDEAD
MEM     3 0xDEADBEEF

LDI     R0 2
MVI     R1 0xBEEF
MVI     R2 3
LD      R2 R2

LSI     R0 R0 1     "MyLabel"
ADD     R3 R0 R1
BRNI    R2 R3 >MyLabel

STI     R2 4
TERMINATE
```

Machine Code:

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000001101111010101101
11011110101011011011111011101111
---
00100011000000000000000000000010
00100111000100001011111011101111
00100111001000000000000000000011
00100010001000100000000000000000
00001101000000000000000000000001
00000010001100000001000000000000
01000111001000110000000000000100
00101001001000000000000000000100
11111111111111111111111111111111
```

Branching:

- Simulator has no branch prediction, so branching is expensive
- To demonstrate this, I use a simple fibonacci sequence calculator up to Fib(47).
- Best case: 236 cycles with looping, vs 98 unrolled.
- 1.0 instructions per cycle with looping vs 2.0 unrolled.

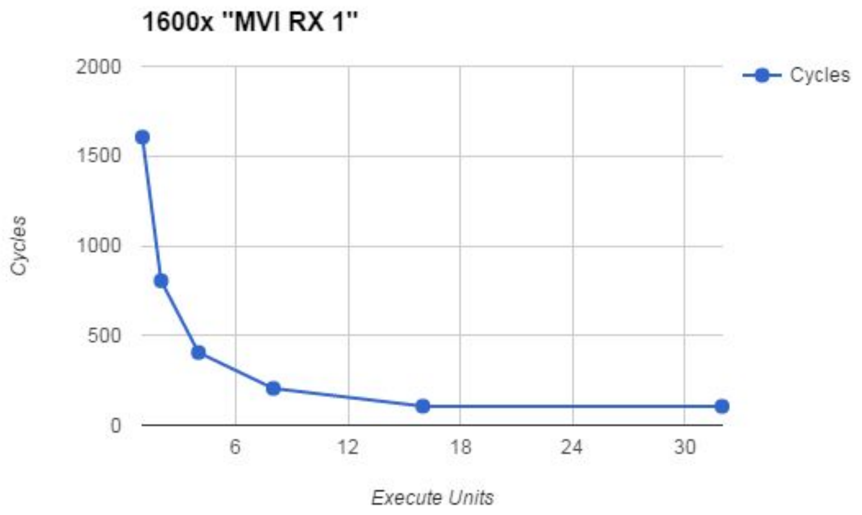
- The branching version benefitted from superscalar execution, whereas unrolled version did not.
    - With only 1 EU, it took 373 cycles compared to 236 with 3 or more.

Experiment 1 - Performance is capped at 16 EUs

- Because there are only 16 registers, the largest number of independent instructions is 16
- Increasing beyond this amount cannot improve performance, as there is no further parallelism to exploit
- Using, “`MVI RX 1`” for each register, dependencies are unavoidable after 16 instructions. After using all registers, repeat.
- 1600 instructions (+1 for terminate)

Experiment 1 - Result:

- There is a 1 cycle performance increase between 16 and 32 execute units.
  - This is because the 32 EU version can handle the terminate instruction at the same time as an MVI
- Beyond this, there is no benefit to using more than 16 execute units



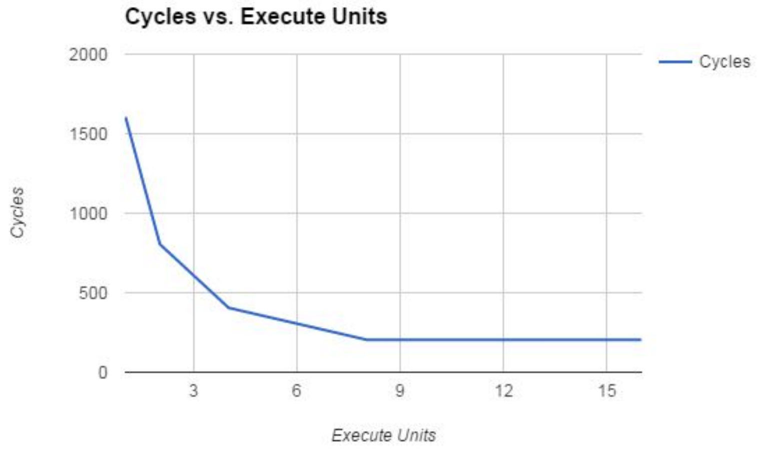
Experiment 2 - Lack of bypassing effective cap

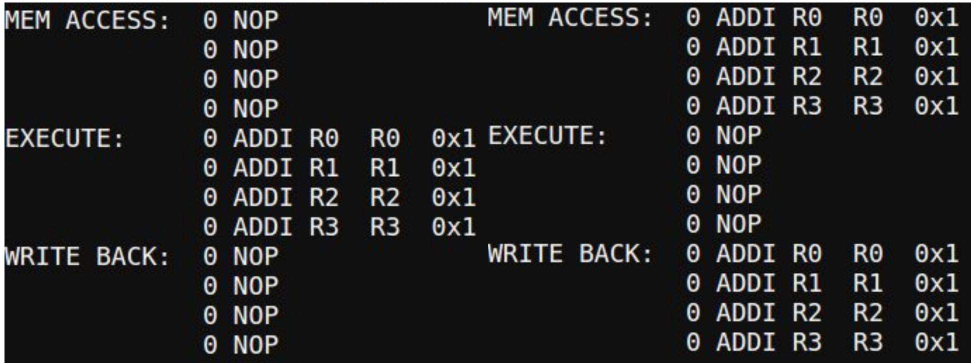
- Because of register writeback, ALU operations take an extra cycle past executing to complete.
- This can effectively cap the number of useful execute units at 8, due to read-after-write dependencies.
- Using only instructions such as “`ADDI RX RX 1`” we can force bubbles to appear in the pipeline to demonstrate this.

Experiment 2 - Result

As can be seen, the bubbles in the pipeline effectively half the throughput for these programs.

With a cap of 16 independent instructions, but half the pipeline being empty, we get the same performance using 16 EUs as we do with 8.

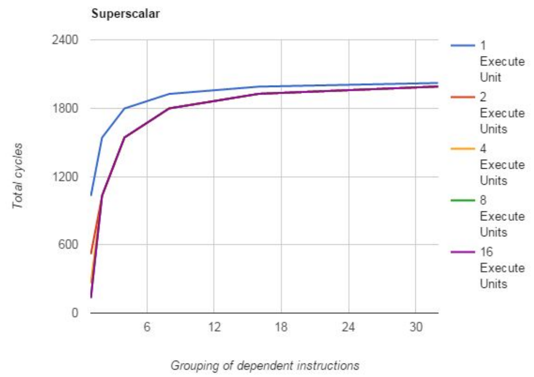


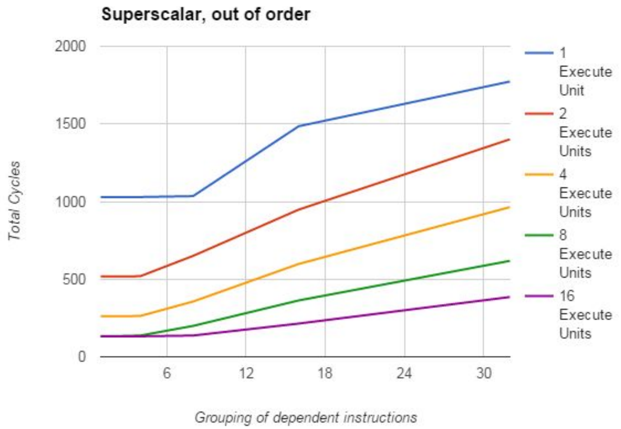


```
MEM ACCESS:  0 NOP                MEM ACCESS:  0 ADDI R0  R0  0x1
             0 NOP                             0 ADDI R1  R1  0x1
             0 NOP                             0 ADDI R2  R2  0x1
             0 NOP                             0 ADDI R3  R3  0x1
EXECUTE:     0 ADDI R0  R0  0x1  EXECUTE:     0 NOP
             0 ADDI R1  R1  0x1               0 NOP
             0 ADDI R2  R2  0x1               0 NOP
             0 ADDI R3  R3  0x1               0 NOP
WRITE BACK:  0 NOP                WRITE BACK:  0 ADDI R0  R0  0x1
             0 NOP                             0 ADDI R1  R1  0x1
             0 NOP                             0 ADDI R2  R2  0x1
             0 NOP                             0 ADDI R3  R3  0x1
```

Experiment 3: Comparing in-order and out-of-order

- All of these programs do the same thing, they just order their instructions differently.
  - Along the x axis, we increase the proximity of dependent instructions.
- You can see how out-of-order execution allows the processor to scale much better than in-order.
- The in-order processor shows next to no scaling past 2 execute units.


Superscalar
Total cycles
Grouping of dependent instructions
1 Execute Unit
2 Execute Units
4 Execute Units
8 Execute Units
16 Execute Units


Superscalar, out of order
Total Cycles
Grouping of dependent instructions
1 Execute Unit
2 Execute Units
4 Execute Units
8 Execute Units
16 Execute Units