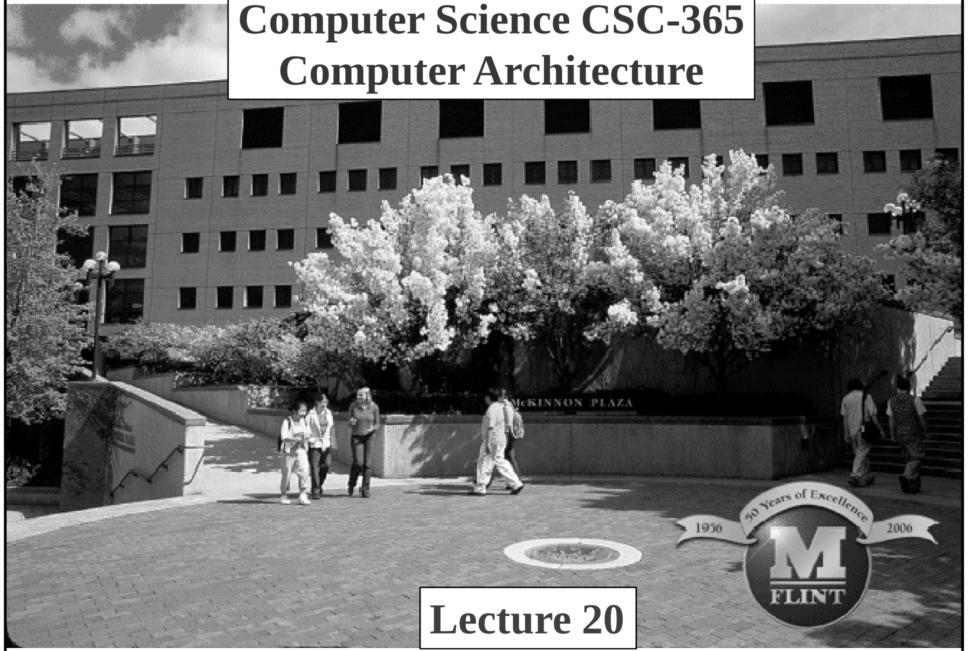
Computer Science CSC-365
Computer Architecture



Lecture 20

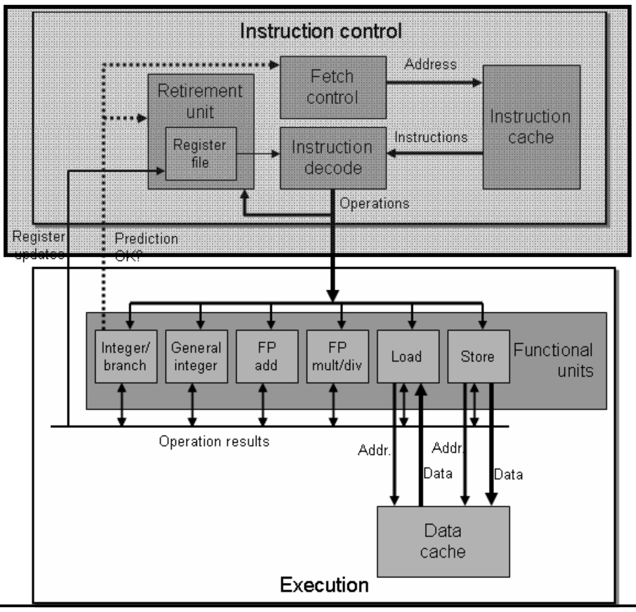
CSC-365
Computer Architecture

Chapter 5

Optimizing Program Performance

Section 5.7

Recall our Multiple Functional Units



Recall the Execution Unit (EU) Capabilities

- Multiple Instructions Can Execute in Parallel
  - 1 load
  - 1 store
  - 2 integer (one may be branch)
  - 1 FP Addition
  - 1 FP Multiplication or Division
- Benefit of multiple functional units
  - Since the Load/Store unit performs the MEM functions
    - It can now run in parallel to any EXE function
    - It also has its own Adder so we don’t need the other functional units for it to do its job

Recall the Execution Unit (EU) Latencies

- Possible Instructions:

Instruction	Latency	Issue Time
Compare	1	1
Integer Add	1	1
Jump	1	1
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Double/Single FP Multiply	5	2
Double/Single FP Add	3	1
Double/Single FP Divide	38	38

Example of the Processing of the ICU (A loop to multiply all array elements)

- Code to multiply all the elements of an array together:
  - Integer data, multiply operation

```
.L24:                          # Loop:
  imull (%eax,%edx,4),%ecx     # t *= data[i]
  incl %edx                    # i++
  cmpl %esi,%edx               # i:length
  jl .L24                      # if < goto Loop
```

- Translation of First Iteration

```
.L24:
  imull (%eax,%edx,4),%ecx

  incl %edx
  cmpl %esi,%edx
  jl .L24
```

```

load (%eax,%edx.0,4) ➔ t.1
imull t.1, %ecx.0    ➔ %ecx.1
incl %edx.0          ➔ %edx.1
cmpl %esi, %edx.1    ➔ cc.1
jl-taken cc.1
```

Recall the Use of Renaming for all Registers

- Instruction was originally:

  imull (%eax,%edx,4),%ecx

- Which performed:

  %ecx = (%eax,%edx,4) * %ecx

- We now avoid WAW and WAR hazards by replacing this with:

  load (%eax,%edx.0,4) ➔ t.1

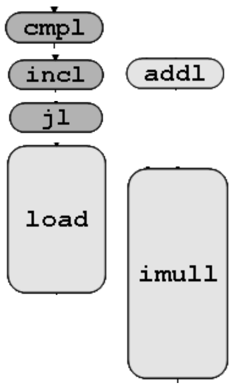
  imull t.1, %ecx.0 ➔ %ecx.1

Note there are multiple copies of registers possible

Recall the Computational Graph Elements

- Possible Instructions:

– Instruction	Latency	Issue Time	Notation
– Compare	1	1	
– Integer Add/Increment	1	1	
– Jump	1	1	
– Load / Store	3	1	
– Integer Multiply	4	1	
– Integer Divide	36	36	
– Double/Single FP Multiply	5	2	
– Double/Single FP Add	3	1	
– Double/Single FP Divide	38	38	



Visualizing Operations in the EXE Stage for a The Following Instructions

- Recall the code to perform multiplications of all the elements of the array together
- It consists of two sets of operations
  1. Actual multiplications of the array elements
  2. Incrementing the index for the array

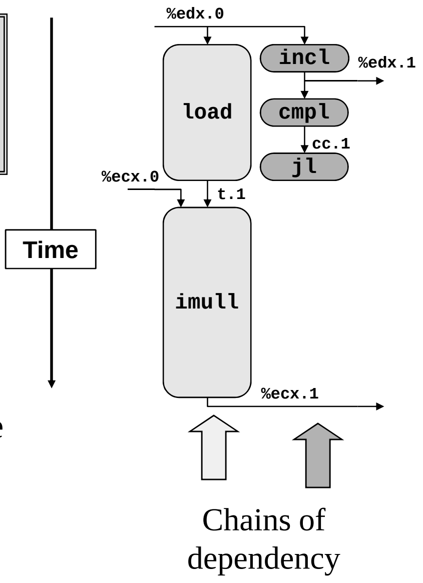
```
load (%eax,%edx.0,4) ➔ t.1
imull t.1, %ecx.0    ➔ %ecx.1
incl %edx.0          ➔ %edx.1
cmpl %esi, %edx.1    ➔ cc.1
jl-taken cc.1
```

First we can look at the multiplications in the instruction stream

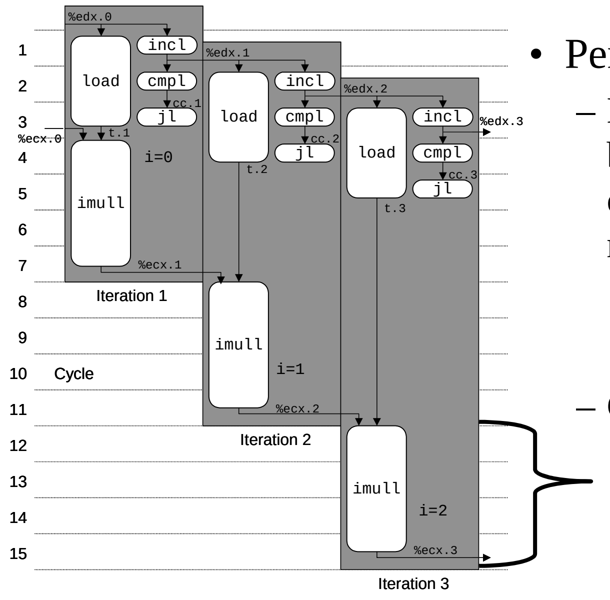
Recall Computational Graph in the EXE Stage for the Complete Array Multiplication

```
load (%eax,%edx,4)  ➔ t.1
imull t.1, %ecx.0   ➔ %ecx.1
incl %edx.0          ➔ %edx.1
cmpl %esi, %edx.1   ➔ cc.1
jl-taken cc.1
```

- We can parallelize these instructions
  - Creates two unique paths of dependencies now
- Multiple lateral entries denote simultaneous instruction issues



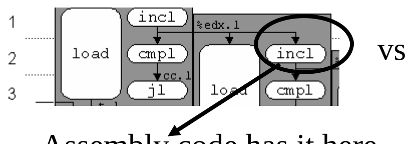
3 Iterations of Combining Product



- Performance
  - Limiting factor becomes ***latency*** of integer multiplier
  - Gives CPE of 4.0
    - Every 4 clock cycles a new result is generated

Benefits of Using the Computational Graph

- We can easily see the parallelism that is going on in our machine
- We can also see we are using Out-of-Order Execution

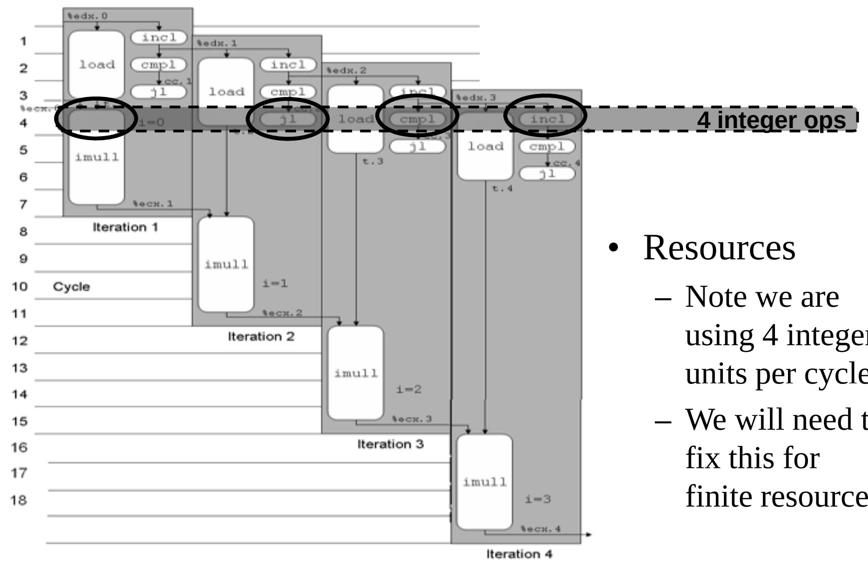


```
load (%eax,%edx,4) ➔ t.1
imull t.1, %ecx.0  ➔ %ecx.1
incl %edx.0         ➔ %edx.1
cmpl %esi, %edx.1  ➔ cc.1
jl-taken cc.1
```

Assembly code has it here

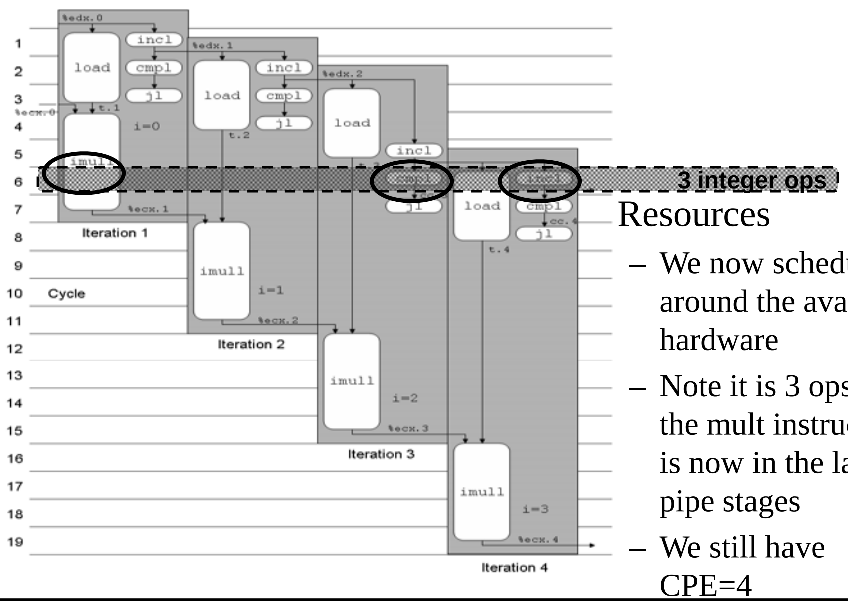
- Instructions are being executed in the order in time that the resources are available
  - This order is different than when the assembly code dictated
  - Of course there must be no data dependencies between instructions whose order is changed

4 Iterations of Combining Product



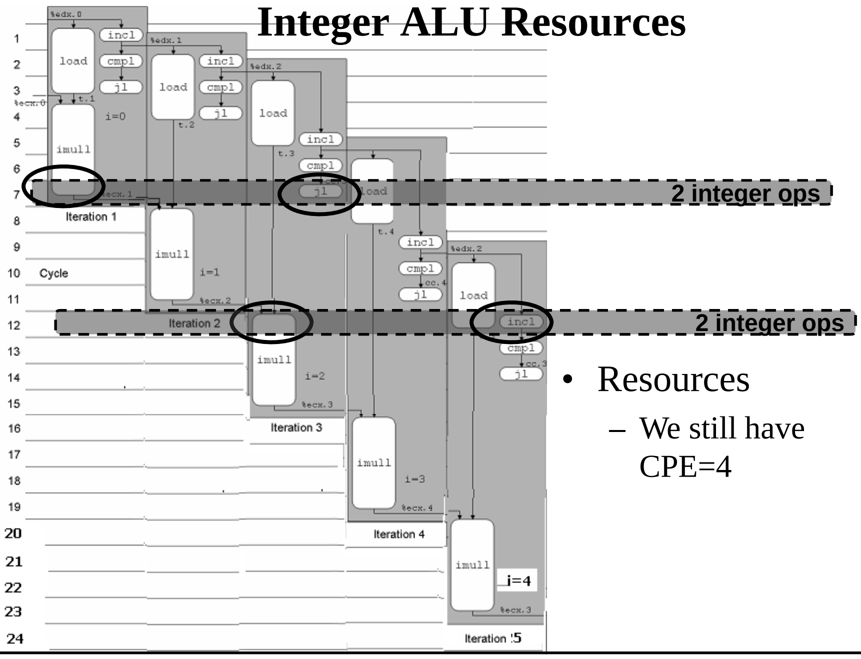
- Resources
  - Note we are using 4 integer units per cycle
  - We will need to fix this for finite resources

4 Iterations of Combining Product with 3 Integer ALU Resources



Resources
- We now schedule around the available hardware
- Note it is 3 ops, but the mult instruction is now in the later pipe stages
- We still have CPE=4

4 Iterations of Combining Product with 2 Integer ALU Resources



- Resources
  - We still have CPE=4

Operations in the EXE stage for Mult

- Notice that the CPE of the Multiplication example matches the latency of the Mult operation
  - We are limited by the amount of time the mult takes, even with an infinite amount of hardware
- Now lets look at an example where we replace the Mult operation with an Add
  - In other words we add all the elements of an array together
  - Recall the Add has a latency of only 1

Add Example for Hardware Scheduling

- Code to ADD all the elements of an array together:
  - Integer data, add operation

```
.L24:                          # Loop:
  iaddl (%eax,%edx,4),%ecx     # t += data[i]
  incl %edx                    # i++
  cmpl %esi,%edx               # i:length
  jl .L24                      # if < goto Loop
```

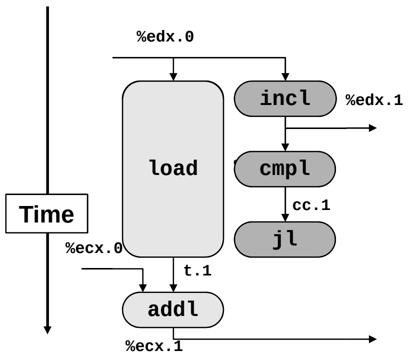
- Translation of First Iteration

```
.L24:
  iaddl (%eax,%edx,4),%ecx

  incl %edx
  cmpl %esi,%edx
  jl .L24
```

```
load (%eax,%edx.0,4) ➔ t.1
iaddl t.1, %ecx.0    ➔ %ecx.1
incl %edx.0          ➔ %edx.1
cmpl %esi, %edx.1    ➔ cc.1
jl-taken cc.1
```

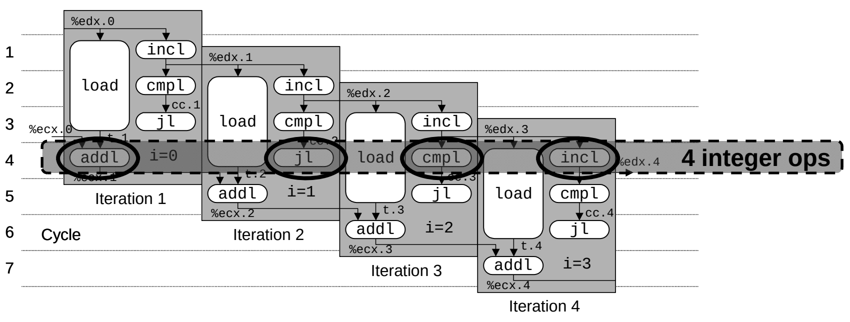
Visualizing Operations in the EXE stage for ADD



```
load (%eax,%edx,4) ➔ t.1
iaddl t.1, %ecx.0  ➔ %ecx.1
incl %edx.0         ➔ %edx.1
cmpl %esi, %edx.1  ➔ cc.1
jl-taken cc.1
```

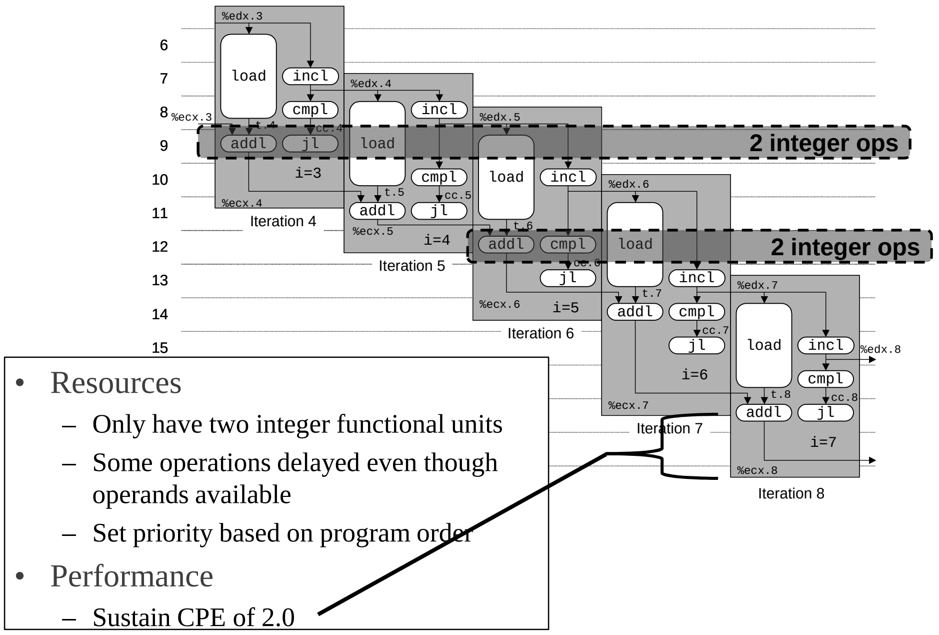
- Operations
  - Same as before, except that add has latency of 1
  - This simple change will have a dramatic impact on the scheduling

4 Iterations of Combining Sum



- Unlimited Resource Analysis
- Performance
  - Can begin a new iteration on each clock cycle
  - Should give CPE of 1.0
  - Would again require executing 4 integer operations in parallel !!!

Combining Sum: Resource Constraints



- Resources
  - Only have two integer functional units
  - Some operations delayed even though operands available
  - Set priority based on program order
- Performance
  - Sustain CPE of 2.0

Combining Sum: Resource Constraints II

- Note we now reschedule our instructions to match the availability of the hardware
- How do we actually schedule?
  - The details of this are extremely complicated
  - They are a major portion of CSC-565
- The general way is:
  - We track - what is the true order of execution of the instructions
  - Instruction priority is set based on its location in the machine code instruction streams
    - Earlier instructions are given higher priority than later instructions

Combining Sum: Resource Constraints III

- What do we see:
  - The performance of our system is now limited not by our usual data and control dependencies but now by hardware resources.
- There are now three limitations to performance:
  1. Data dependencies (Data Hazards)
  2. Control dependencies (Control Hazards)
  3. Hardware resource limitations
     - Called Structural Hazards

Combining Sum: Resource Constraints IV

- What to do?
  - Note that the first one is a limitation of our code
  - The second limitation we greatly reduced thru branch prediction
  - Is there anything we can do to minimize the impact of the third limitation?

Combining Sum: Resource Constraints V

- What limited our performance
  - There were not enough integer units to perform our add/mult operations or the incl, cmpl, or and jl instructions
  - Note the add and mult instructions were the only ones really involved with our calculation
  - The other three instructions were merely overhead for our loop that we used to compute the running product/sum

What to do??

- Therefore we need to find a way to reduce our loop overhead so we can free-up the functional units to perform the useful instructions
  - We do this with ***Loop Unrolling***

Next Class

- Next class:
  - Loop unrolling Section 5.8
  - Begin Chapter 6