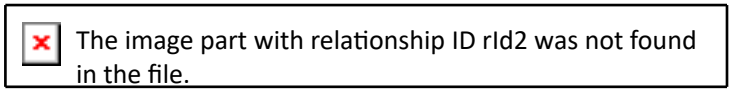
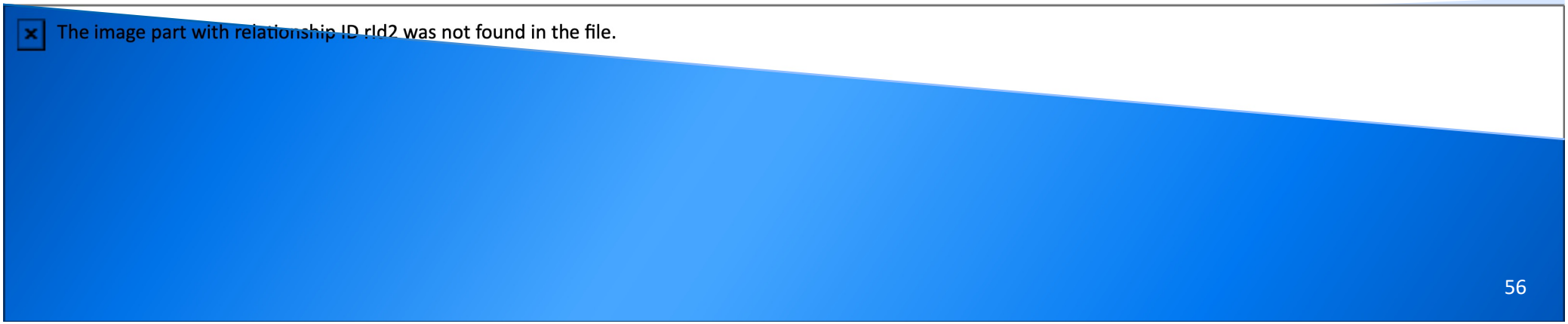


Department of Computer Science

Lecture #16: Chap. 5: Program Performance Optimization (2)

Prof. Soraya Abad-Mota, PhD



From the index cards

- Procedures may be optimized "inside" by the compiler's optimizer, it is the calls to procedures what will not be optimized or weak optimizations.
- Issue time: how many cycles need to pass before the processor can start a new operation of that kind that is independent of the current data

- CPU vs cores
- Are switch statements subject to prediction? (today)

Previously: presented a benchmark

- combine function operates on all the elements of the vector to compute either the summation or the product of all the elements.
- Notice that this function has a for loop in which a function is called in the condition of the loop and in the body of the loop. (This is why it is relevant to do problem 5.3)
- Showed results of performance by making common optimizations described in previous lecture

The effect of all Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

- Eliminates sources of overhead in loop

The bounds (1)

- *Latency bound*: minimum value for the CPE for any function that must perform the combining op. in a strict sequence. (How many cycles must pass until can start new op.)

  e.g. 1 FU for int * with issue of 1 ➔ 1 * per cycle

x86-64 Compilation of Combine4

- Inner Loop (Case: Integer Multiply)

```
.L519:                                  # Loop:
  imull  (%rax,%rdx,4), %ecx            # t = t * d[i]
  addq   $1, %rdx                       # i++
  cmpq   %rdx, %rbp                     # Compare length:i
  jg     .L519                          # If >, goto Loop
```

Sec 5.7.3

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

latency/issue time

The bounds (2)

- *Throughput bound:* minimum bound for the CPE based on the maximum rate at which the FU can produce results (C/issue time)

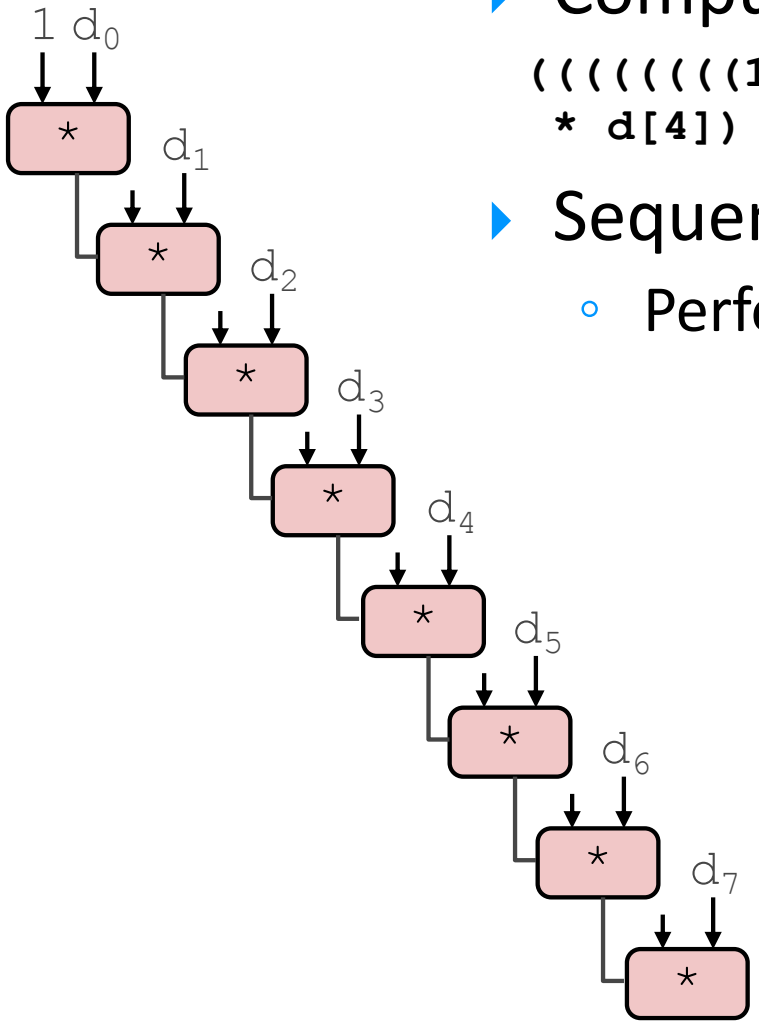
  e.g. integer addition has

  C = 4 → 4 ops/cycle but (4 FUs do integer arithmetic)

  2 FU for load → 2 reads per cycle

  ➔ throughput bound = 0.50

Combine4 = Serial Computation (OP = *)



- Computation (length=8)

```
((((((((1 * d[0]) * d[1]) * d[2]) * d[3])
 * d[4]) * d[5]) * d[6]) * d[7])
```

- Sequential dependence
  - Performance: determined by latency of OP

Data flow graphs: presented sec. 5.7.3 pp. 525-529, study on your own to be able to do some problems at end of chapter.
Tool for analyzing the performance.
Data dependencies.

Continue

Studying factors that constraint performance

+

how to improve further

Back to our running example: *combine*

1. The three measures of latency, issue time and capacity of arithmetic operations can affect the performance of our combining functions.
2. Effects measured by two bounds in CPE values
   - *Latency bound* and *Throughput bound*

Results so far (for combine4)

- Int + Int * fp + fp *
- 1.27 3.01 3.01 5.01 (which almost matches
- 1.00 3.00 3.00 5.00 *latency bound*)

Can we optimize further than this bound?

Can we achieve the *throughput bound*?

- 0.50 1.00 1.00 0.50

Loop Optimization: *loop unrolling*

Loop Unrolling (2x1)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration, by calculating the operations of 2 iterations in one.

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

```
x = (x OP d[i]) OP d[i+1];
```

- Helps integer add
  - Achieves latency bound (min value for CPE)
- Others don't improve. *Why?*
  - Still sequential dependency

Loop Unrolling with Reassociation (2x1a)

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

different order in the textbook

Compare to before

```
x = (x OP d[i]) OP d[i+1];
```

(sec. 5.9.2)

- Can this change the result of the computation?
- Yes, for FP. *Why? are FP + * associative?*

Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

- Nearly 2x speedup for Int *, FP +, FP *
  - Reason: Breaks sequential dependency

```
x = x OP (d[i] OP d[i+1]);
```

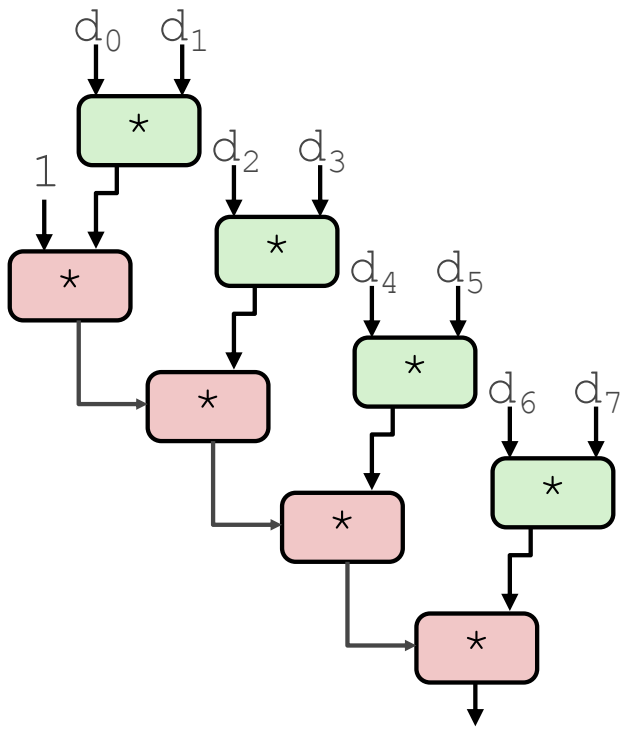
  - Why is that? (next slide)

2 func. units for FP *
2 func. units for load

4 func. units for int +
2 func. units for load

Reassociated Computation

```
x = x OP (d[i] OP d[i+1]);
```



The image part with relationship ID rId13 was not found in the file.

- What changed:
  - Ops in the next iteration can be started early (no dependency)
- Overall Performance
  - N elements, D cycles latency/op
  - (N/2+1)*D cycles:
    CPE = D/2

Loop Unrolling with Separate Accumulators (2x2)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
       x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- Different form of reassociation

Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

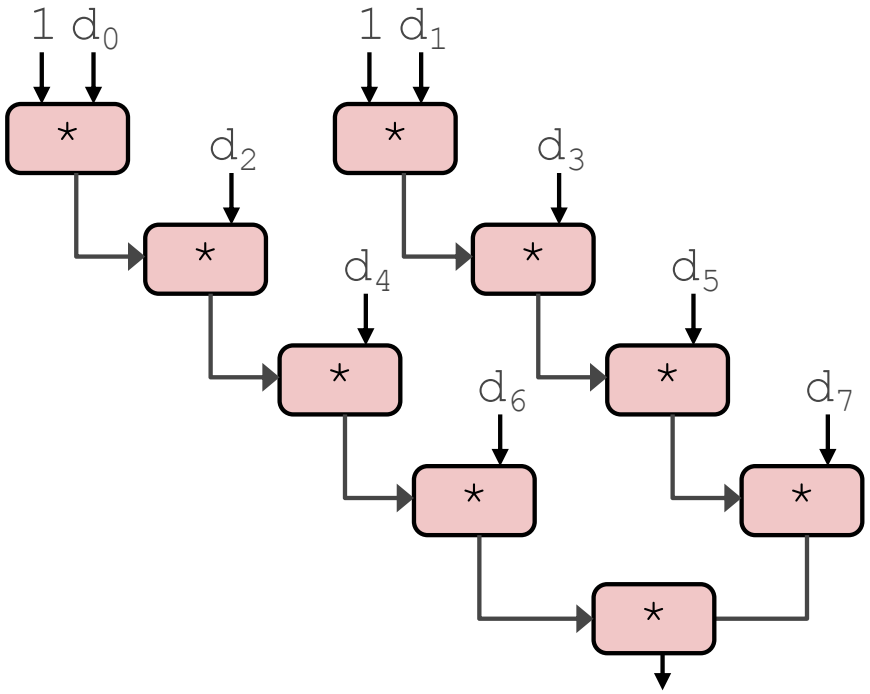
- Int + makes use of two load units

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

- 2x speedup (over unroll 2x1) for Int *, FP +, FP *

Separate Accumulators

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



- **What changed:**
  - Two independent "streams" of operations
- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles: **CPE = D/2**
  - CPE matches prediction!

What Now?

Unrolling & Accumulating

- Idea
  - Can unroll to any degree L
  - Can accumulate K results in parallel
  - L must be multiple of K
- Limitations
  - Diminishing returns
    - Cannot go beyond throughput limitations of execution units
  - Large overhead for short lengths
    - Finish off iterations sequentially
- Summary of unrolling and accumulating experiments (next three slides)

Unrolling & Accumulating: Double *

- Case
  - Intel Haswell
  - Double FP Multiplication
  - Latency bound: 5.00. Throughput bound: 0.50

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51		2.51		
3			1.67					
4				1.25		1.26		
6					0.84			0.88
8						0.63		
10							0.51	
12								0.52

Accumulators

Unrolling & Accumulating: Int +

- Case
  - Intel Haswell
  - Integer addition
  - Latency bound: 1.00. Throughput bound: 0.50

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	
2		0.81		0.69		0.54		
3			0.74					
4				0.69		1.24		
6					0.56			0.56
8						0.54		
10							0.54	
12								0.56

Accumulators

Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

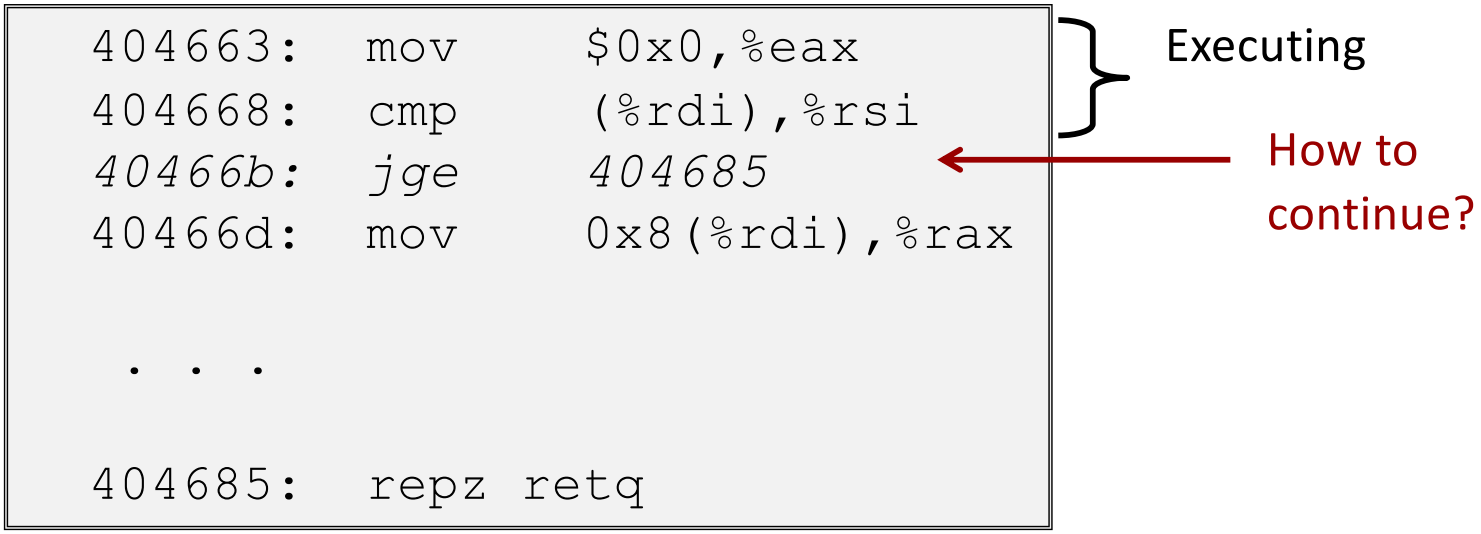
- Limited only by throughput of functional units
- Up to 42X improvement over original, unoptimized code

Last topics in Ch. 5

- End of chap. 5 (Performance)
  - Dealing with Conditionals
  - Conclusions of Performance Improvement

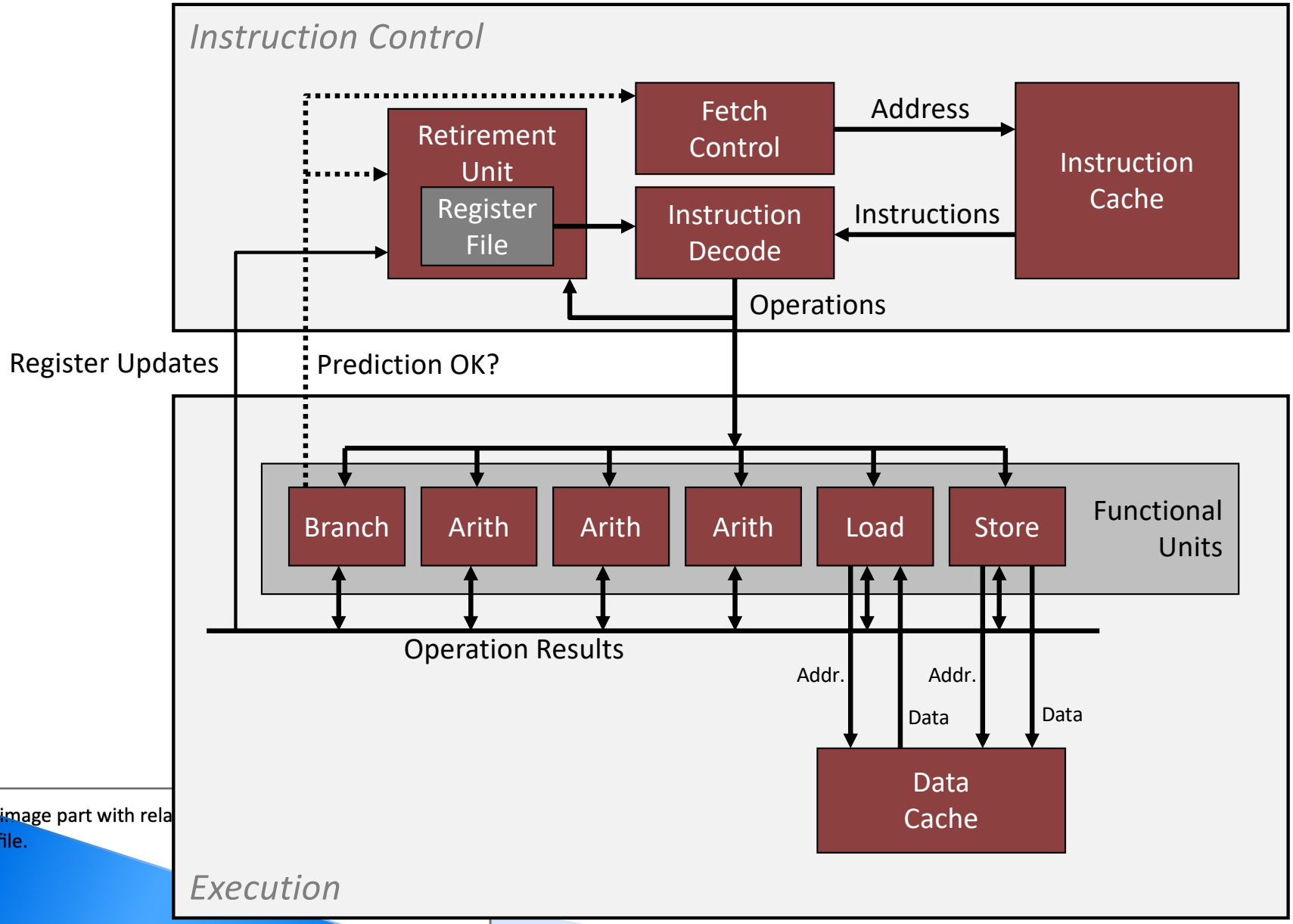
What About Branches?

- Challenge
  - Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy



  - When ICU encounters conditional branch, cannot reliably determine where to continue fetching

Modern CPU Design (recall)



The image part with rela
the file.

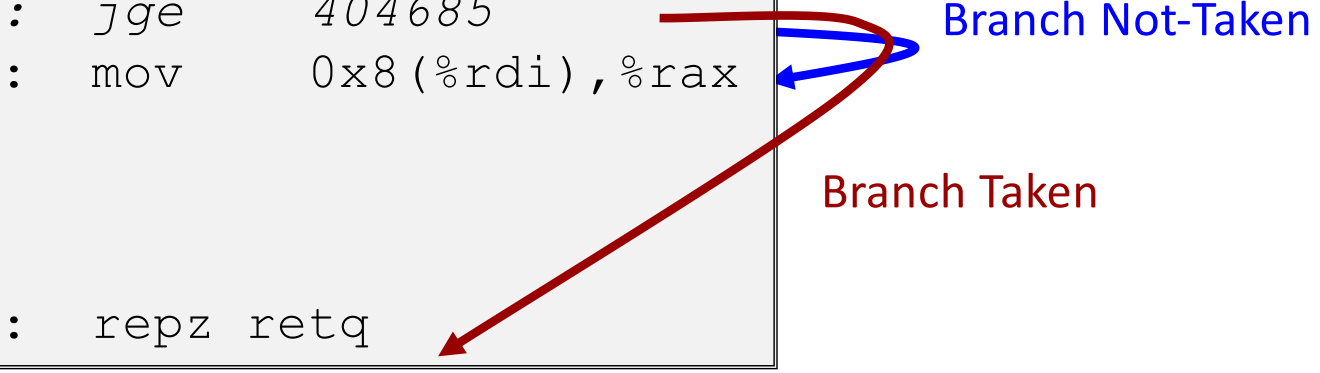
Branch Outcomes

- When conditional branch is encountered, cannot determine where to continue fetching
  - Branch Taken: Transfer control to branch target
  - Branch Not-Taken: Continue with next instruction in sequence
- Cannot resolve until outcome determined by branch/integer unit

```
404663:   mov     $0x0,%eax
404668:   cmp     (%rdi),%rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi),%rax

 . . .

404685:   repz retq
```



The image part with relationship ID rId13 was not found in the file.

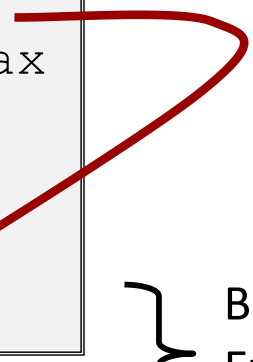
Branch Prediction

- Idea
  - Guess which way branch will go
  - Begin executing instructions at predicted position
    - But don't actually modify register or memory data

```
404663:   mov     $0x0,%eax
404668:   cmp     (%rdi),%rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi),%rax

 . . .

404685:   repz retq
```



The image part with relationship ID rId13 was not found in the file.

Branch Prediction Through Loop

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 98

Predict Taken (OK)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 99

Predict Taken (Oops)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

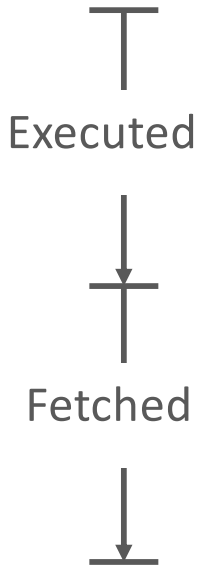
i = 100

Read invalid location

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 101

Assume vector length = 100



Branch Misprediction Invalidation

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 98

Assume
vector length = 100

Predict Taken (OK)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 99

Predict Taken (Oops)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 100

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 101

Invalidate

Branch Misprediction Recovery

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
401036:  jmp    401040
 . . .
401040:  vmovsd %xmm0,(%r12)
```

i = 99

Definitely not taken

Reload Pipeline

- Performance Cost
  - Multiple clock cycles on modern processor
  - Can be a major performance limiter
- Instruction *retirement* vs *flushed*

Performance Improvement Conclusions (section 5.13)

- Basic strategies for optimizing program performance:
  - High-level Design: appropriate algorithms and data structures for the problem + avoid algorithms or coding techniques with asymptotically poor performance. (e.g n^2 instead of n log n)
- Basic coding principles: avoid optimization blockers
  - eliminate excessive function calls: move computations out of loops + compromises: modularity vs greater efficiency
  - eliminate unnecessary mem refs: temporary variables to store intermediate results, store in array or global var only when final value is computed

Performance Improvement Conclusions (2)

- Low-level Optimizations: take advantage of HW capabilities
  - unroll loops to reduce overhead and further optimize
  - increase instruction-level parallelism: e.g. with multiple accumulators and reassociation
  - The authors have found that unrolling a loop and accumulating multiple values in parallel is a more reliable way to achieve improved program performance.
  - rewrite conditional operations to enable conditional data transfers