Lecture 11: Machine-Dependent Optimization

CS 105

February 27, 2019

From Monday…

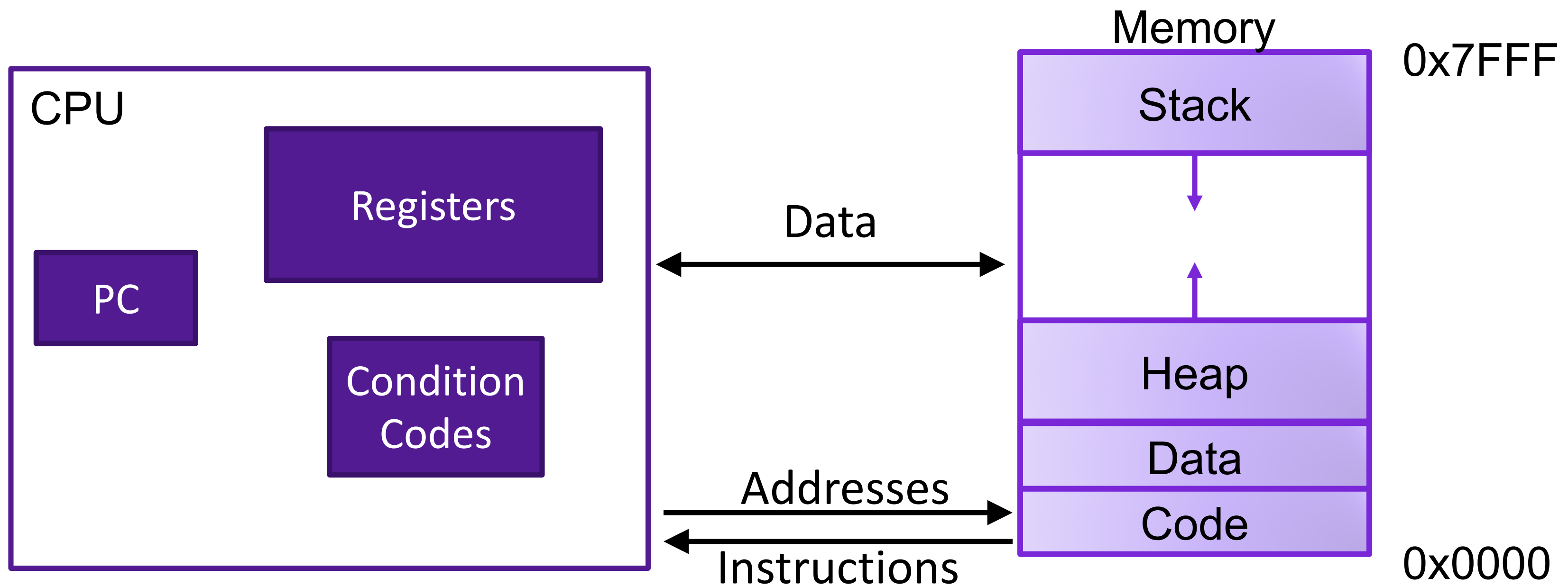
```
void combine1(vec_ptr v, data_t *dest)
{
  long i;
  *dest = IDENT;

  for (i = 0; i < vec_length(v); i++) {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OP val;
  }
}
```

```
void combine2(vec_ptr v, data_t *dest)
{
  long i;
  data_t t = IDENT;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  for (i = 0; i < length; i++){
    t = t OP d[i];
  }
  *dest = t;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O0	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01

Assembly/Machine Code View



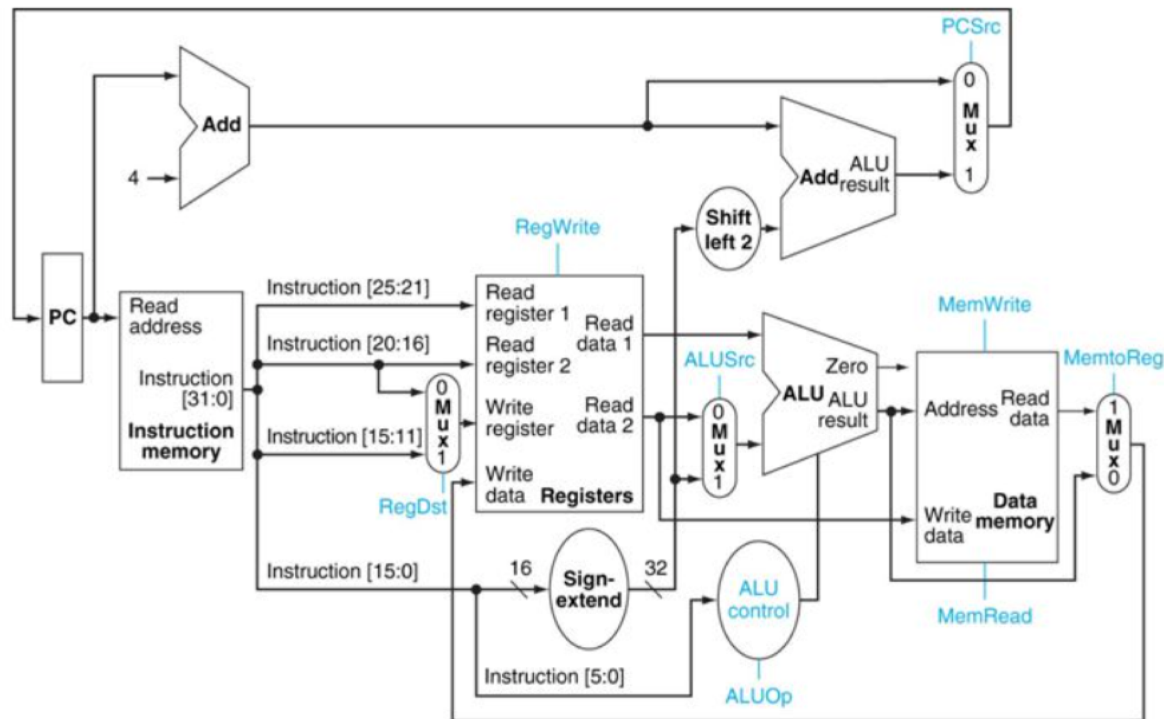
Programmer-Visible State

- PC: Program counter
- 16 Registers
- Condition codes

Memory

- Byte addressable array
- Code and user data
- Stack to support procedures

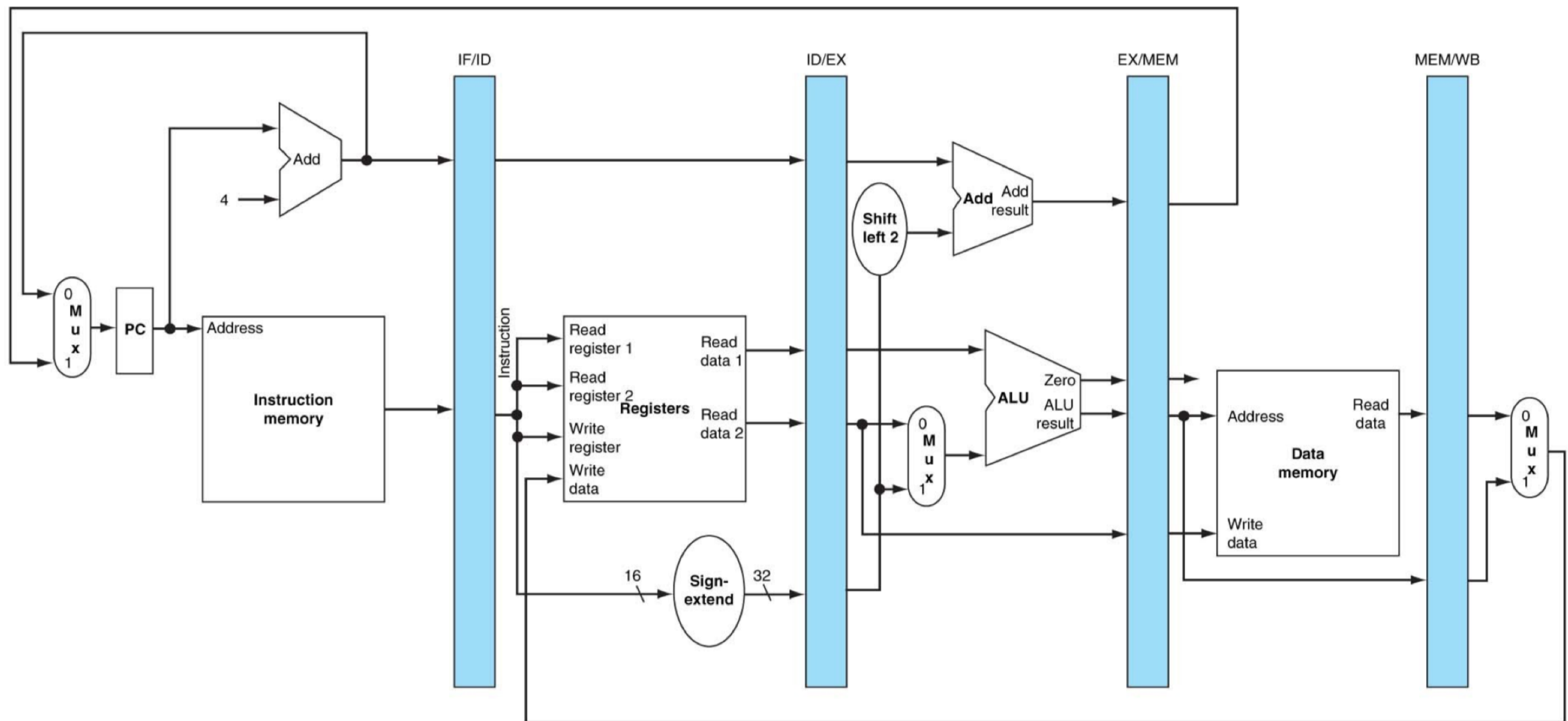
The Datapath for a CPU



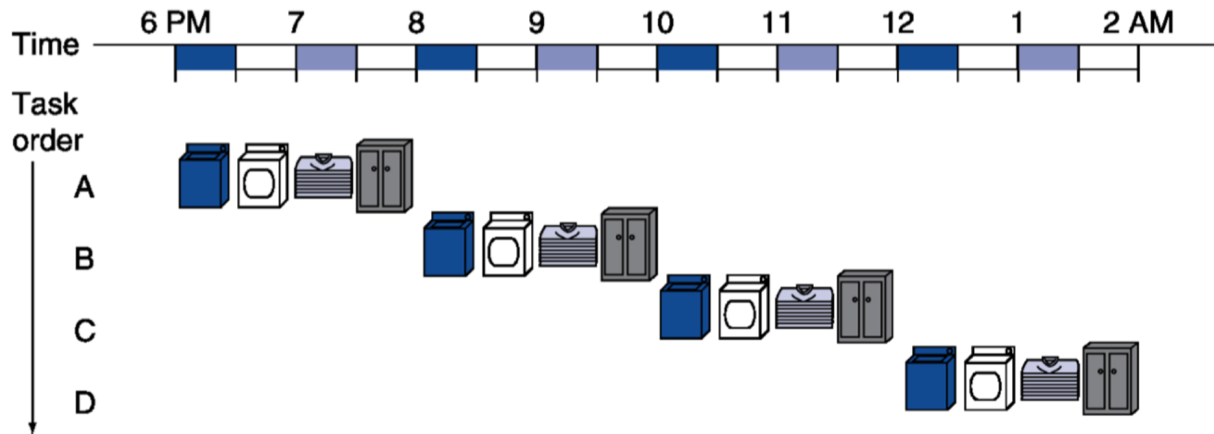
Pipelined Processors

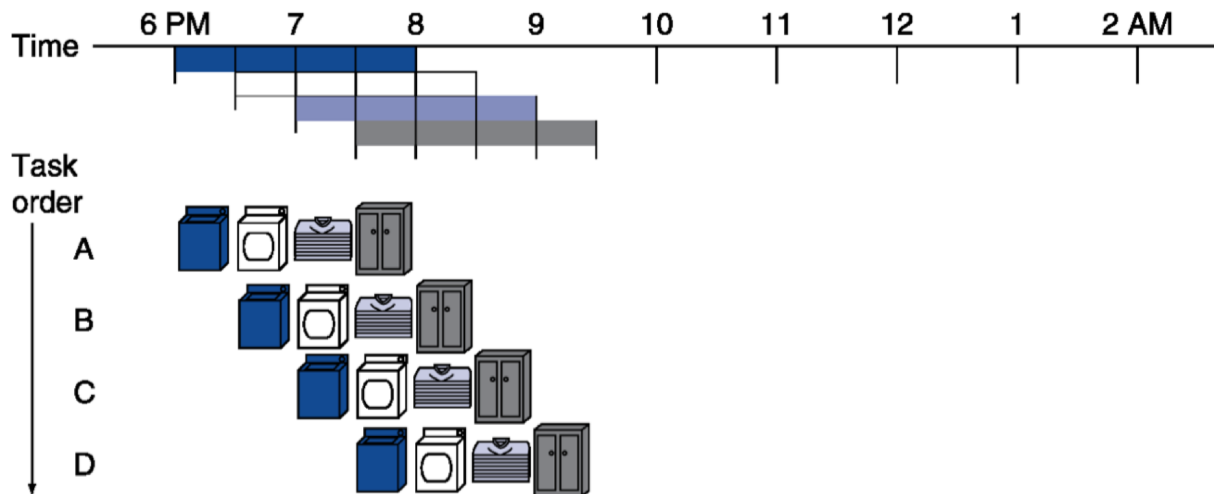
1. Instruction Fetch (IF)
2. Instruction Decode and register file read (ID)
3. Execute or address calculation(EX)
4. Memory Access (MEM)
5. Write back (WB)

Pipelined Processors

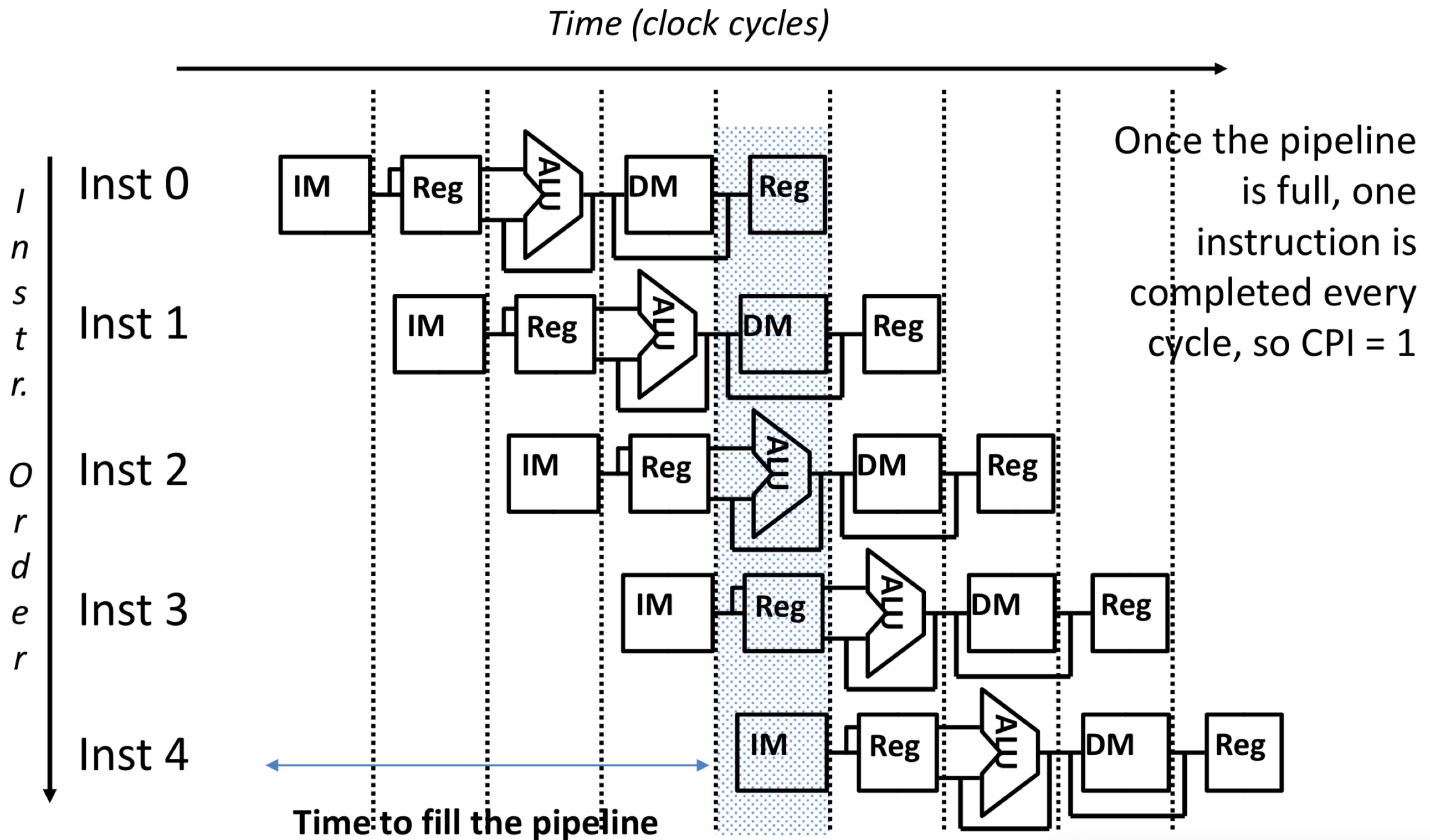


Pipelined Processors





Pipelined Processors



Superscalar Processors

- Definition: A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

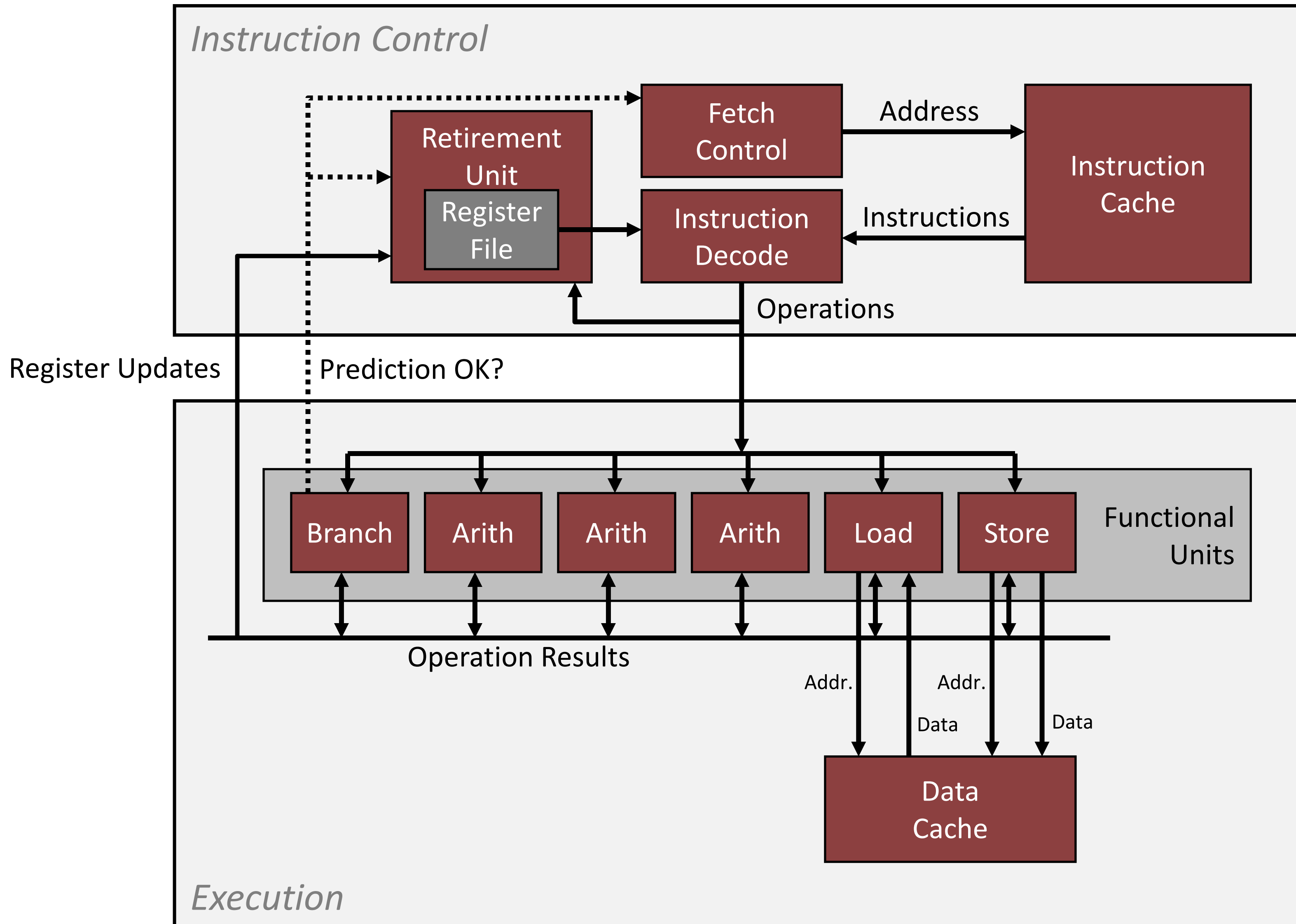
- Benefit: without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have

- Most modern CPUs are superscalar.
- Intel: since Pentium (1993)

What might go wrong?

- instructions might need same hardware (Structural Hazard)
  - duplicate functional units

Modern CPU Design



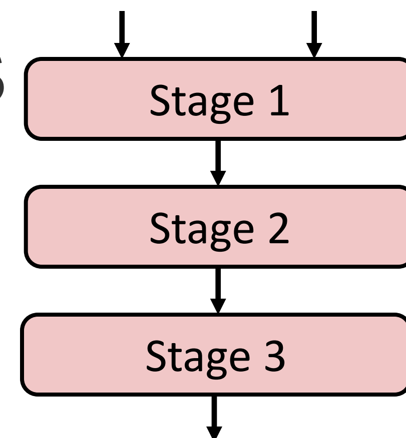
Example: Haswell CPU (introduced 2013)

- 4-way superscalar, 14 pipeline stages
- 8 Total Functional Units
- Multiple instructions can execute in parallel
  - 2 load, with address computation
  - 1 store, with address computation
  - 4 integer
  - 2 FP multiply
  - 1 FP add
  - 1 FP divide
- Some instructions take > 1 cycle, but can be pipelined

Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	3-30	3-30
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	3-15	3-15

Pipelined Functional Units

```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```



Time							
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start on new computation once values passed to i+1
- Complete 3 multiplications in 7 cycles, not 9 cycles

What might go wrong?

- instructions might need same hardware (Structural Hazard)
  - duplicate functional units
  - dynamic scheduling
- instructions might not be independent (Data Hazard)
  - dynamic scheduling
  - forwarding
  - stalling
- might not know next instruction (Control Hazard)
  - branch prediction
  - eliminate unnecessary jumps (e.g., loop unrolling)

combine, revisited

```
void combine2(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;
}
```

- Move vec_length out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

Measuring Performance

- **Latency:** How long to get one result, from beginning to end
  - Appropriate measure when each instruction must wait for another to complete
  - Upper bound on time
- **Throughput:** How long between completions of instructions
  - Usually (1/number of units) * (cycle time)
  - Appropriate measure when there are no dependencies between instructions
  - Lower bound on time

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
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O0	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

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

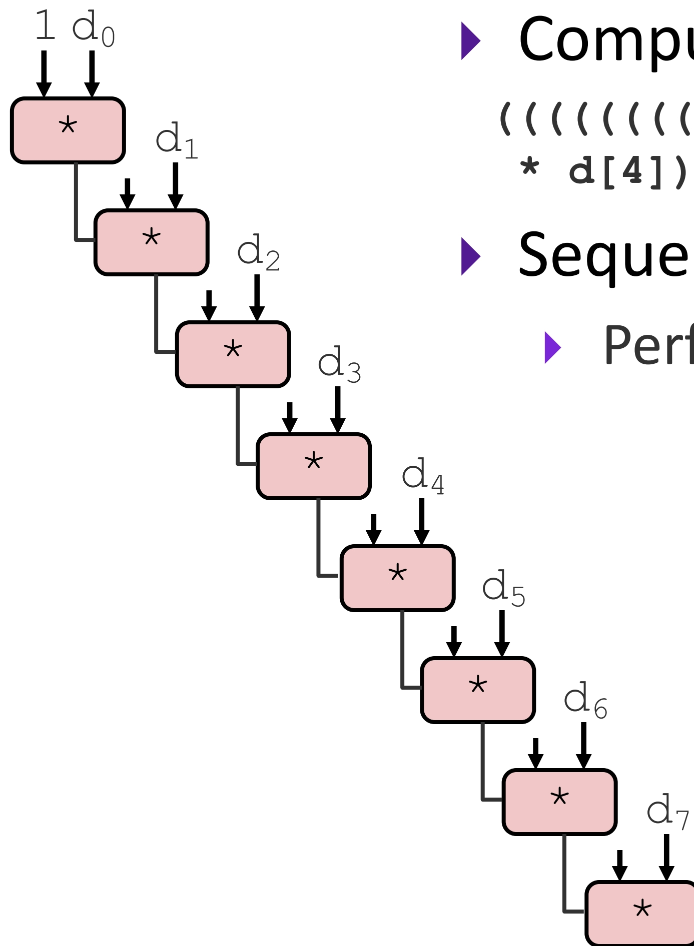
Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O0	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01
Unroll 2	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

- Helps integer add
  - Achieves latency bound
  - Is this the best we can do?

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

Combine = 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

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

Loop Unrolling with Reassociation (2x1a)

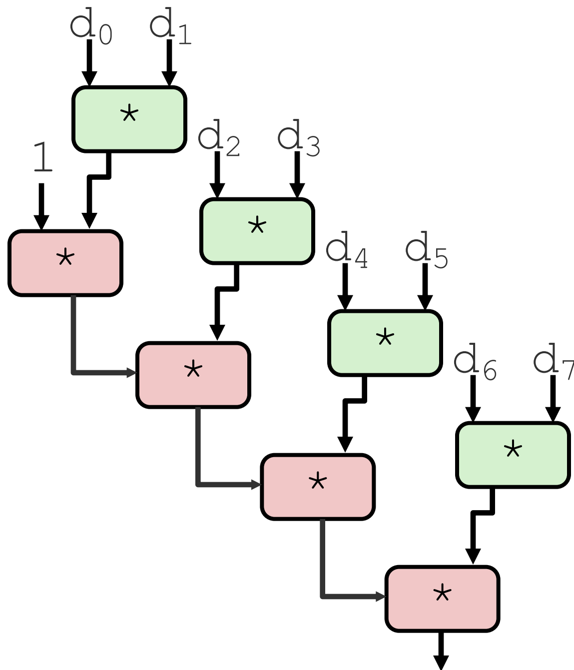
```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
   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

Compare to before

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

Reassociated Computation

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



- 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**

Effect of Reassociation

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Combine1 –O0	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01
Unroll 2	1.01	3.01	3.01	5.01
Unroll 2a	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]);
```

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

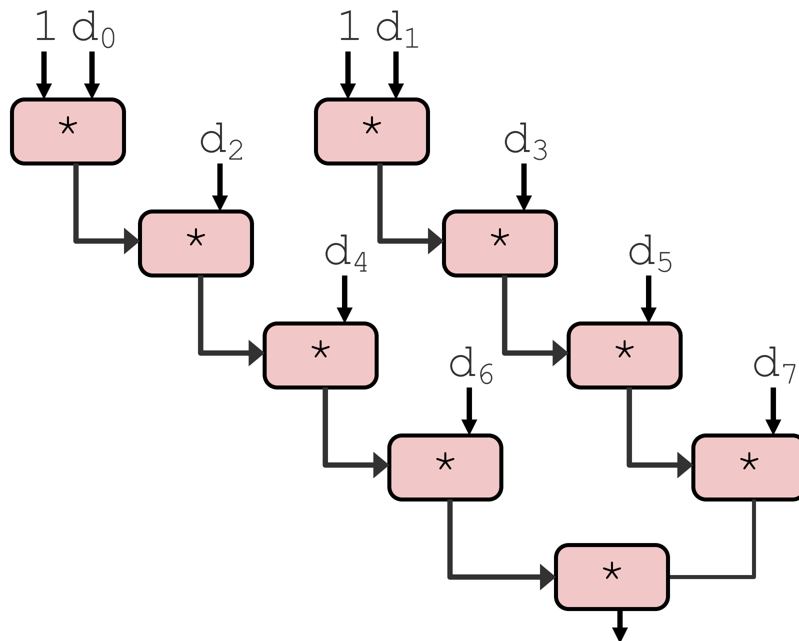
4 func. units for int +
2 func. units for load

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;
}
```

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?

Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	**Add**	**Mult**	**Add**	**Mult**
Combine1 –O0	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01
Unroll 2	1.01	3.01	3.01	5.01
Unroll 2a	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

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

- Int + makes use of two load units
- 2x speedup (over unroll2) for Int *, FP +, FP *

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

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: 1.00

Accumulators

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

Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O0	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine2	1.27	3.01	3.01	5.01
Unroll 2	1.01	3.01	3.01	5.01
Unroll 2a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Optimal Unrolling	0.54	1.01	1.01	0.52
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
  - Combines machine-independent and machine-dependent factors