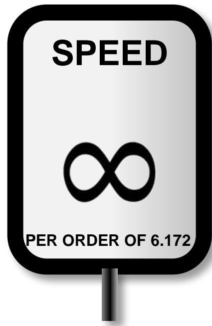
6.172 PERFORMANCE ENGINEERING OF SOFTWARE SYSTEMS



Computer Architecture and Performance Engineering

Saman Amarasinghe

Fall 2010

Outline

- Overview of Computer Architecture
- Profiling a Program
- Set of Example Programs

© Saman Amarasinghe 2008

Computer Architecture Overview

Instructions

Memory System

Processor Bus and IO Subsystem

Disk System

GPU and Graphics System

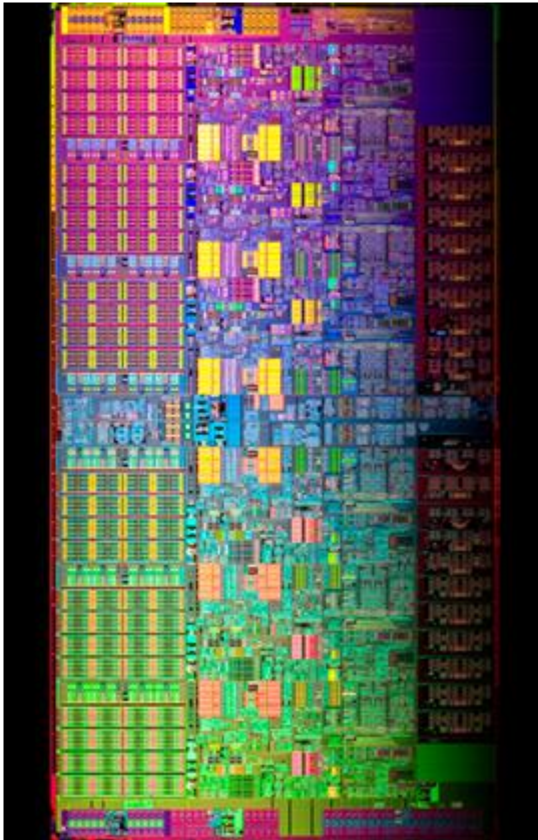
Network

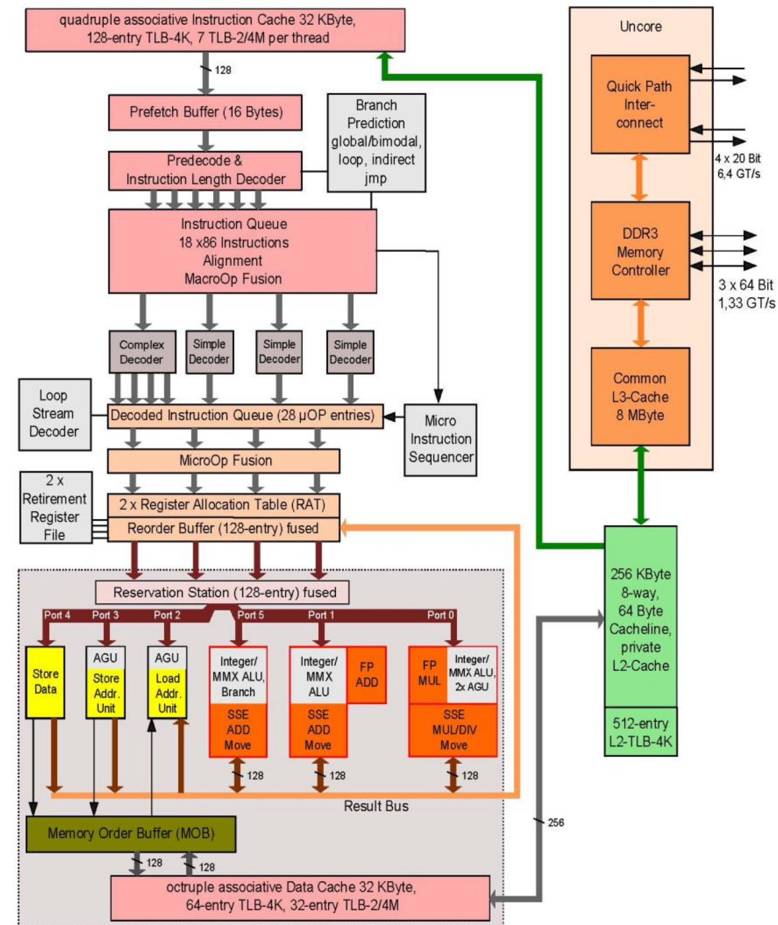
© Saman Amarasinghe 2008

Intel® Nehalem™ Microarchitecture – Computer Architecture Overview

Instructions

Memory System

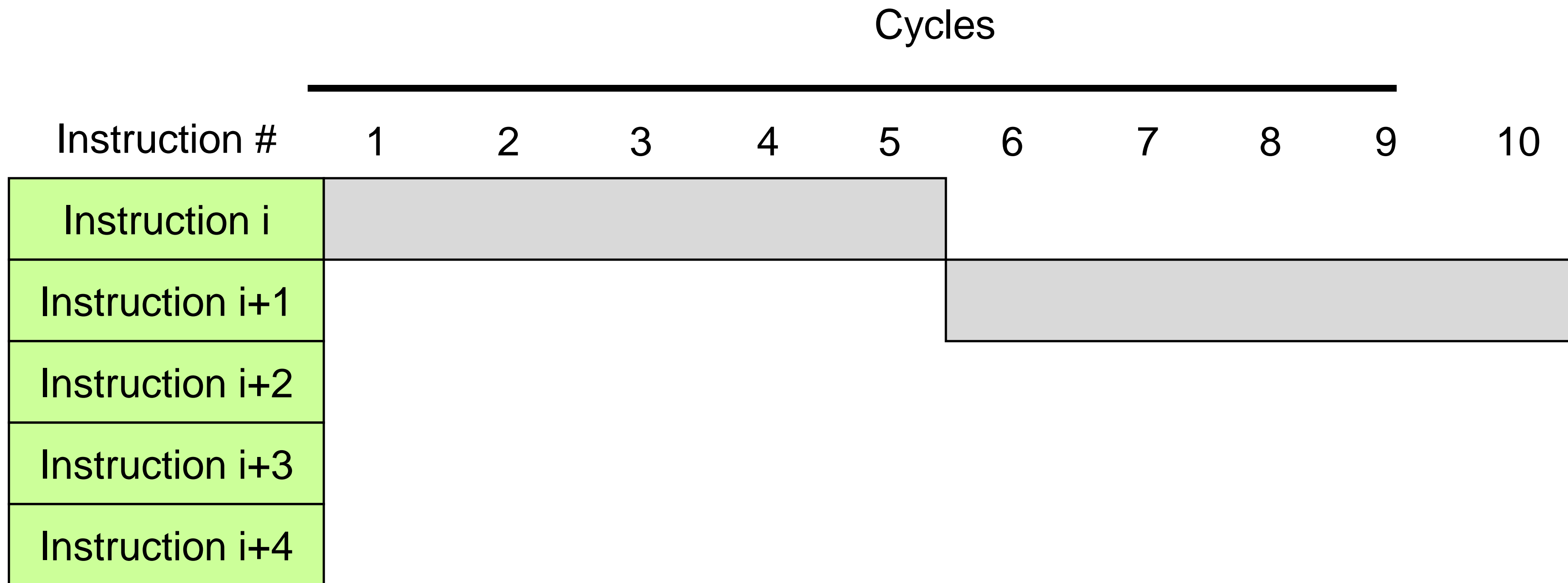




Reprinted with permission of Intel Corporation.

© Saman Amarasinghe 2008

Instruction Execution



Instruction Execution

IF: Instruction fetch
ID : Instruction decode
EX : Execution
MEM: Memory access
WB : Write back

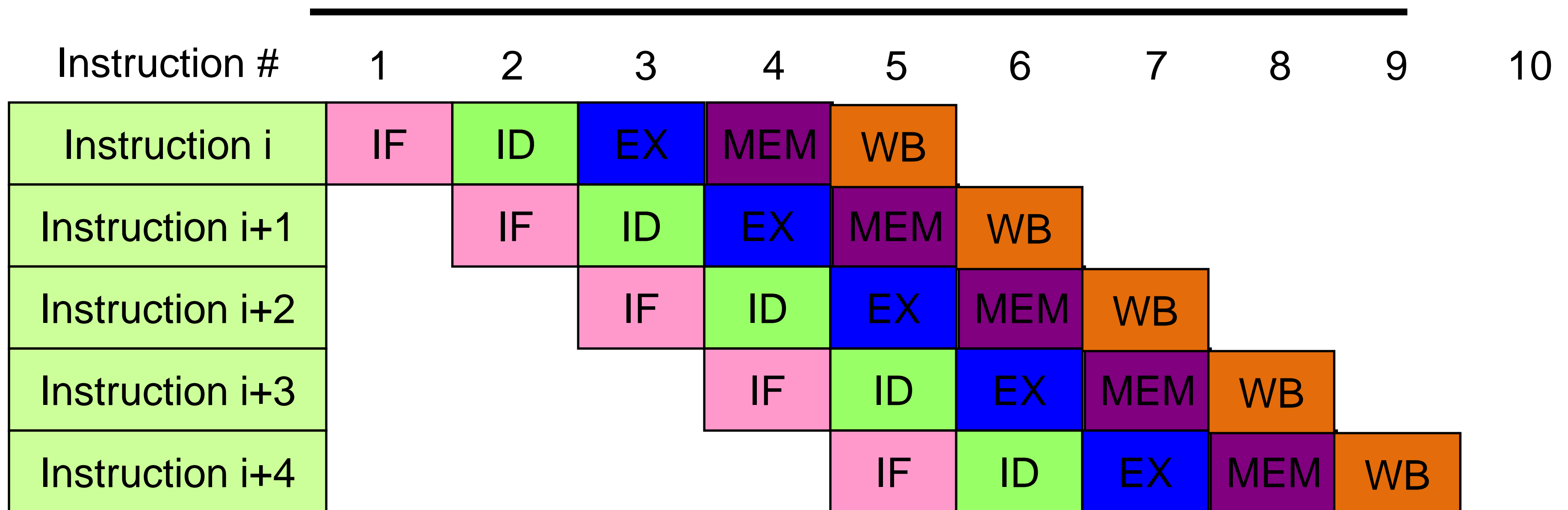
Instruction #	1	2	3	4	5	6	7	8	9	10
Instruction i	IF	ID	EX	MEM	WB					
Instruction i+1						IF	ID	EX	MEM	WB
Instruction i+2										
Instruction i+3										
Instruction i+4										

Cycles

Pipelining Execution

IF: Instruction fetch
ID : Instruction decode
EX : Execution
MEM: Memory access
WB : Write back

Cycles



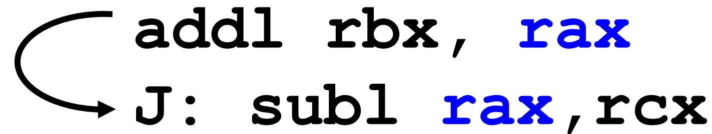
Limits to pipelining

Hazards prevent next instruction from executing during its designated clock cycle

- Structural hazards: attempt to use the same hardware to do two different things at once
- Data hazards: Instruction depends on result of prior instruction still in the pipeline
- Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

Data Hazards: True Dependence

Instr$_J$ is data dependent (aka true dependence) on Instr$_I$:

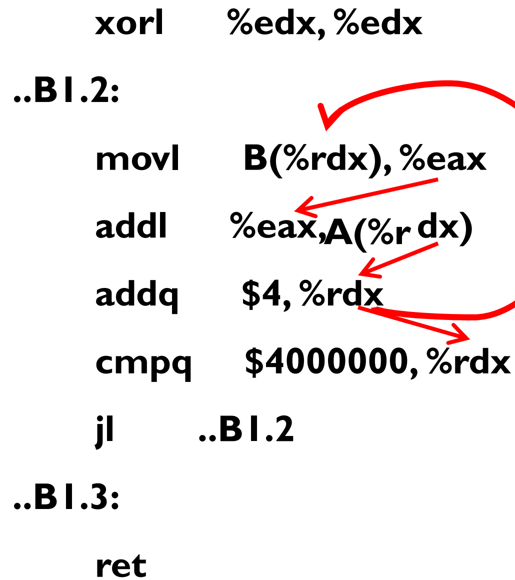


If two instructions are data dependent, they cannot execute simultaneously, be completely overlapped or execute in out-of-order

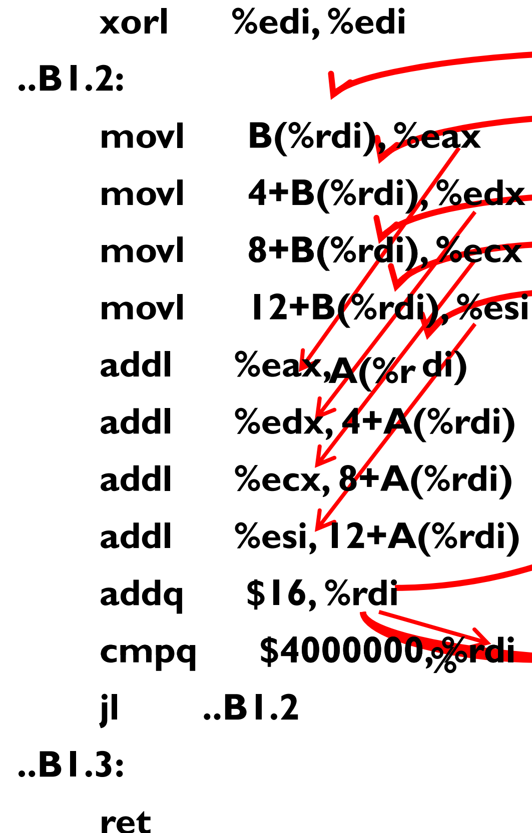
If data dependence caused a hazard in pipeline, called a Read After Write (RAW) hazard

Benefits of Unrolling

```
int A[1000000];
int B[1000000];
test()
{
    int i;
    for(i=0; i <1000000; i++)
       A[i] = A[i] + B[i];
}
```



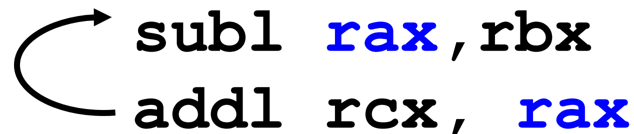

```
For(i=0; i<N; I += 4) {
  A[i] = A[i] + 1
  A[i+1] = A[i+1] + 1
  A[i+2] = A[i+2] + 1
  A[i+3] = A[i+3] + 1
}
```



Name Dependence #1: Anti-dependence

Name dependence: when 2 instructions use same register or memory location, called a **name**, but no flow of data between the instructions associated with that name; **2 versions of name dependence**

Instr$_J$ writes operand *before* Instr$_I$ reads it



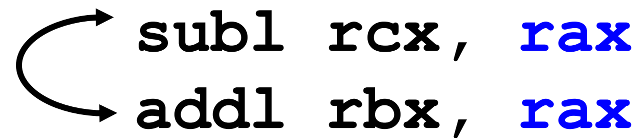
```
subl rax,rbx
addl rcx, rax
```

Called an "anti-dependence" by compiler writers.
This results from reuse of the name "rax"

If anti-dependence caused a hazard in the pipeline, called a Write After Read (WAR) hazard

Name Dependence #2: Output dependence

$Instr_J$ writes operand *before* $Instr_I$ writes it.



Called an "output dependence" by compiler writers.
This also results from the reuse of name "rax"

If anti-dependence caused a hazard in the pipeline, called a Write After Write (WAW) hazard

Instructions involved in a name dependence can execute simultaneously if name used in instructions is changed so instructions do not conflict

- Register renaming resolves name dependence for registers
- Renaming can be done either by compiler or by HW

Control Hazards

Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order

```
if p1 {
  S1;
};
if p2 {
  S2;
}
```

`S1` is control dependent on `p1`, and `S2` is control dependent on `p2` but not on `p1`.

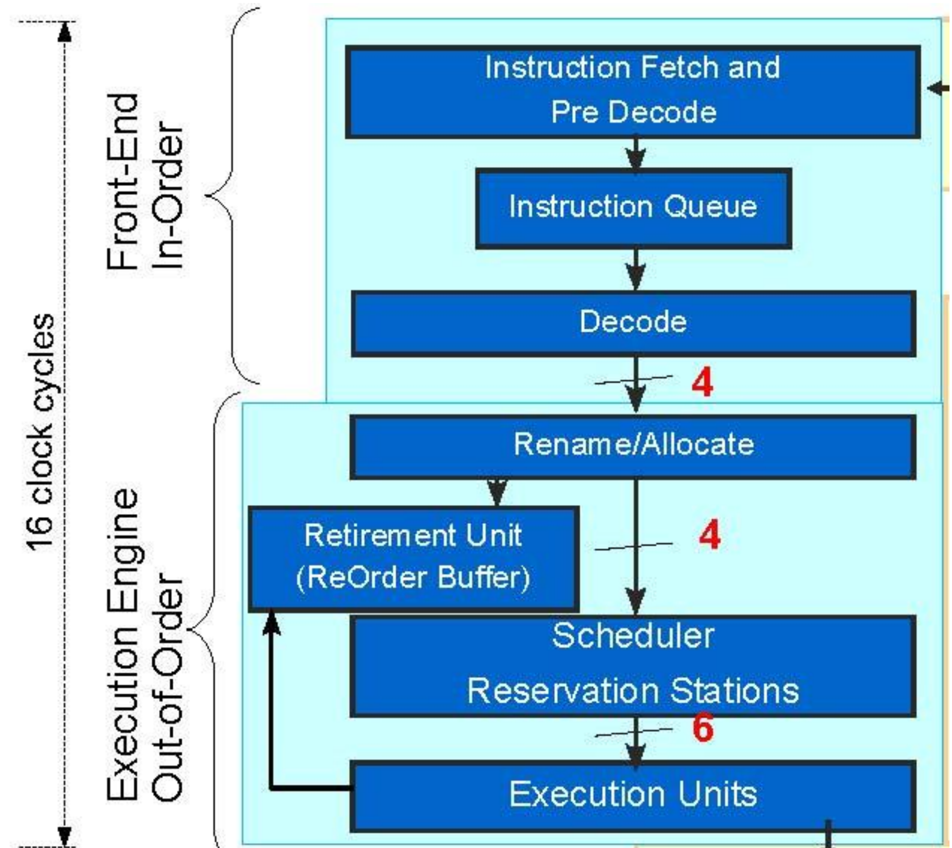
Control dependence need not be preserved

- willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program

Speculative Execution

Intel® Nehalem™ Microarchitecture – Pipelining

20-24 stage Pipeline



Reprinted with permission of Intel Corporation.

© Saman Amarasinghe 2008

Superscalar Execution

Instruction type	1	2	3	4	5	6	7	
Integer	IF	ID	EX	MEM	WB			
Floating point	IF	ID	EX	MEM	WB			
Integer		IF	ID	EX	MEM	WB		
Floating point		IF	ID	EX	MEM	WB		
Integer			IF	ID	EX	MEM	WB	
Floating point			IF	ID	EX	MEM	WB	
Integer				IF	ID	EX	MEM	WB
Floating point				IF	ID	EX	MEM	WB

Cycles

2-issue super-scalar machine

ILP and Data Hazards

Finds Instruction Level Parallelism

- Multiple instructions issued in parallel

HW/SW must preserve program order:
order instructions would execute in if executed sequentially as determined by original source program

- Dependences are a property of programs

Importance of the data dependencies

- 1) indicates the possibility of a hazard
- 2) determines order in which results must be calculated
- 3) sets an upper bound on how much parallelism can possibly be exploited

Goal: exploit parallelism by preserving program order only where it affects the outcome of the program

Multimedia Instructions

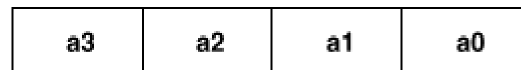
SIMD:

- In computing, ***SIMD*** (**S**ingle **I**nstruction, **M**ultiple **D**ata) is a technique employed to achieve data level parallelism, as in a vector or array processor.
- Intel calls the latest version SSE

Multimedia Instructions

Packed data type

- Separate register file



Single Instruction on Multiple Data (SIMD)

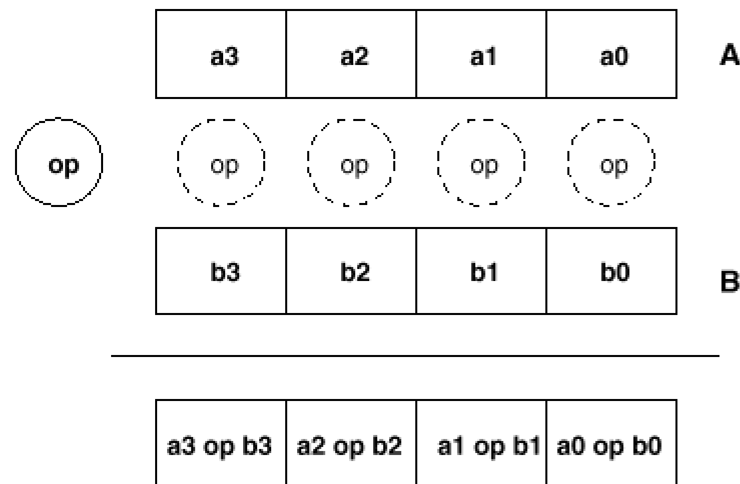


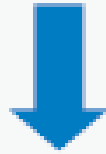
Figure 4: Packed Operation

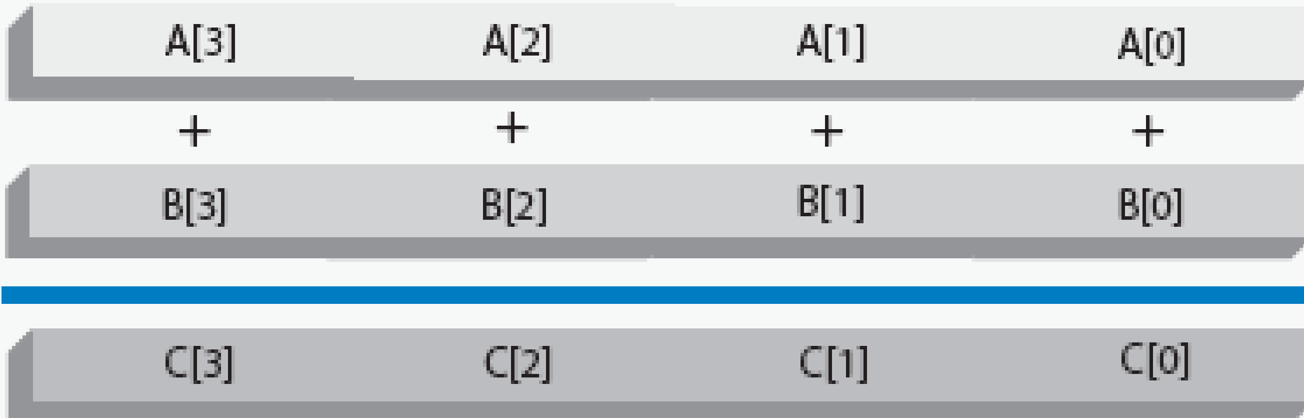
© x86.org. All rights reserved. This content is excluded from our Creative Commons license. For more information, see http://ocw.mit.edu/fairuse.

Multimedia Instructions

Vectorization Converts Loops

```
for (I=0;I<=MAX;I++)
    c [I]=a[I]+b[I];
```





Reprinted with permission of Intel Corporation.

Multimedia Instructions

```
int A[1000000];
int B[1000000];
test()
{
    int i;
    for(i=0; i <1000000; i++)
        A[i] = A[i] + B[i];
}
```

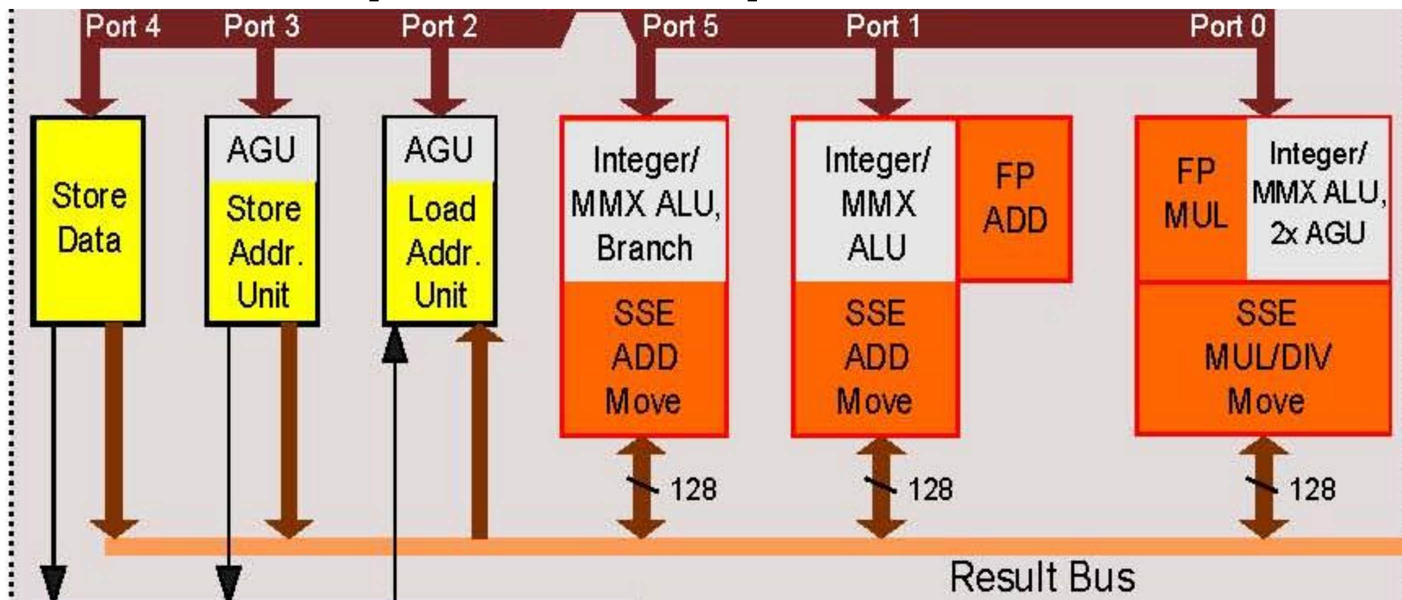
```
        xorl     %edx, %edx
..B1.2:
        movl     B(%rdx), %eax
        addl     %eax, A(%rdx)
        addq     $4, %rdx
        cmpq     $4000000, %rdx
        jl       ..B1.2
..B1.3:
        ret
```

```
        xorl     %eax, %eax
..B1.2:
        movdqa   A(%rax), %xmm0
        paddd    B(%rax), %xmm0
        movdqa   16+A(%rax), %xmm1
        paddd    16+B(%rax), %xmm1
        movdqa   32+A(%rax), %xmm2
        paddd    32+B(%rax), %xmm2
        movdqa   48+A(%rax), %xmm3
        paddd    48+B(%rax), %xmm3
        movdqa   64+A(%rax), %xmm4
        paddd    64+B(%rax), %xmm4
        movdqa   80+A(%rax), %xmm5
        paddd    80+B(%rax), %xmm5
        movdqa   96+A(%rax), %xmm6
        paddd    96+B(%rax), %xmm6
        movdqa   112+A(%rax), %xmm7
        paddd    112+B(%rax), %xmm7
        movdqa   %xmm0, A(%rax)
        movdqa   %xmm1, 16+A(%rax)
        movdqa   %xmm2, 32+A(%rax)
        movdqa   %xmm3, 48+A(%rax)
        movdqa   %xmm4, 64+A(%rax)
        movdqa   %xmm5, 80+A(%rax)
        movdqa   %xmm6, 96+A(%rax)
        movdqa   %xmm7, 112+A(%rax)
        addq     $128, %rax
        cmpq     $4000000, %rax
        jl       ..B1.2
..B1.3:
        ret
```

Intel® Nehalem™ Microarchitecture – Superscalar Execution

Can execute 6 Ops per cycle

- 3 Memory Operations
  - 1 Load
  - 1 Sore address
  - 1 Store data
- 3 Computational Operations



Reprinted with permission of Intel Corporation.

© Saman Amarasinghe 2008

Out of Order Execution

Issue varying numbers of instructions per clock

- dynamically scheduled
  - Extracting ILP by examining 100's of instructions
  - Scheduling them in parallel as operands become available
  - Rename registers to eliminate anti and dependences
  - out-of-order execution
  - Speculative execution

© Saman Amarasinghe 2008

Speculation

Different predictors

- Branch Prediction
- Value Prediction
- Prefetching (memory access pattern prediction)

Greater ILP: Overcome control dependence by hardware speculating on outcome of branches and executing program as if guesses were correct

- Speculation ⇒ fetch, issue, and execute instructions as if branch predictions were always correct
- Dynamic scheduling ⇒ only fetches and issues instructions

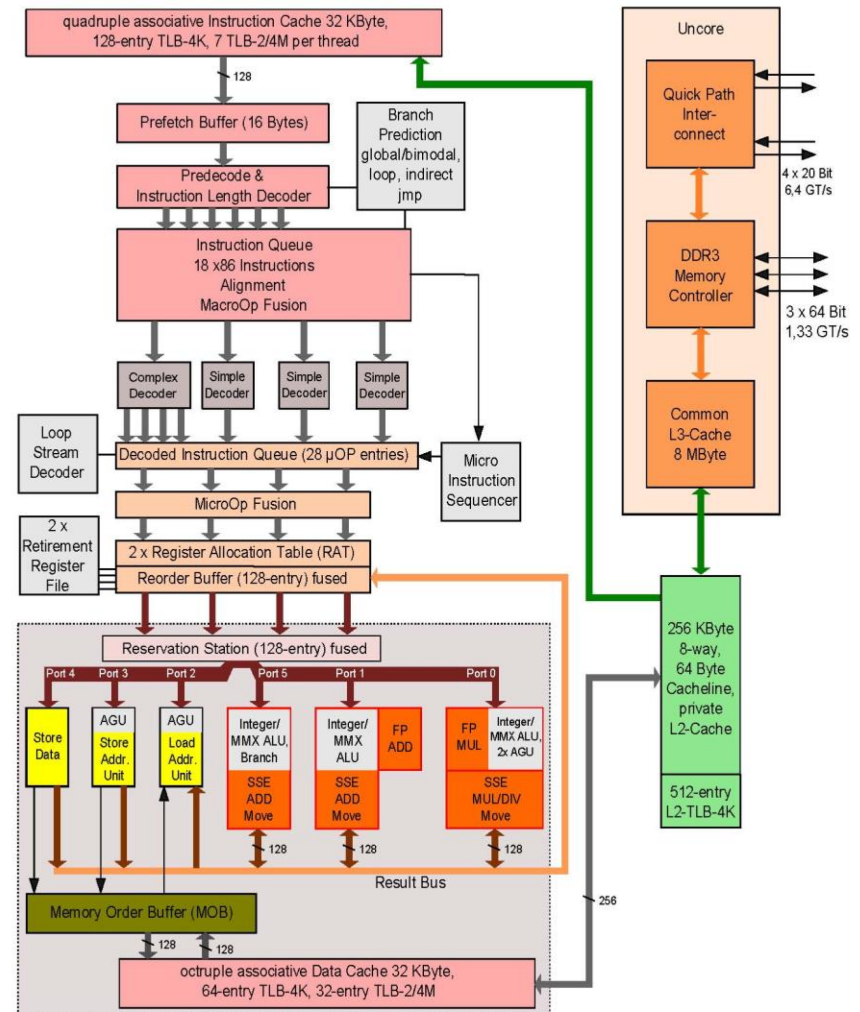
Essentially a data flow execution model: Operations execute as soon as their operands are available

Intel® Nehalem™ Microarchitecture -Out of Order Execution

20 to 24 stage Pipeline

6 micro-ops issued at a time

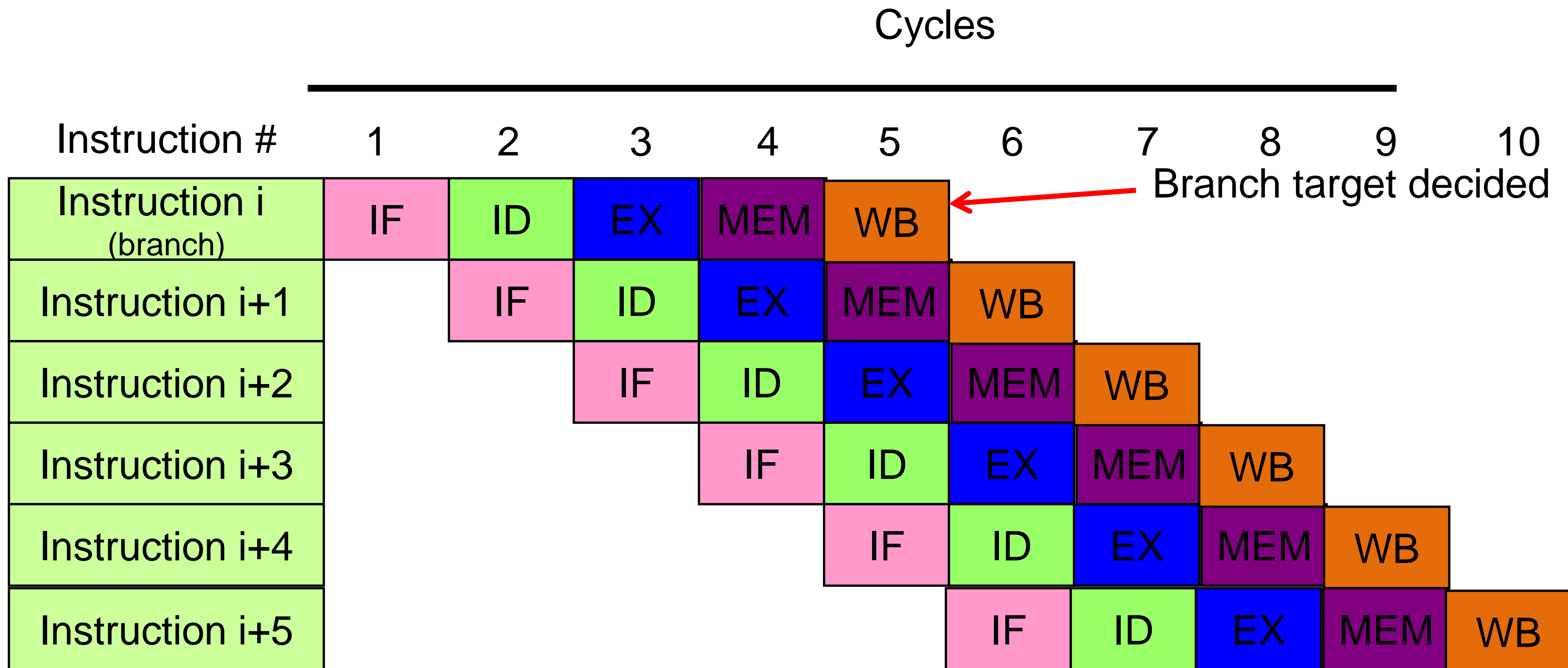
128 micro-ops waiting to be executed



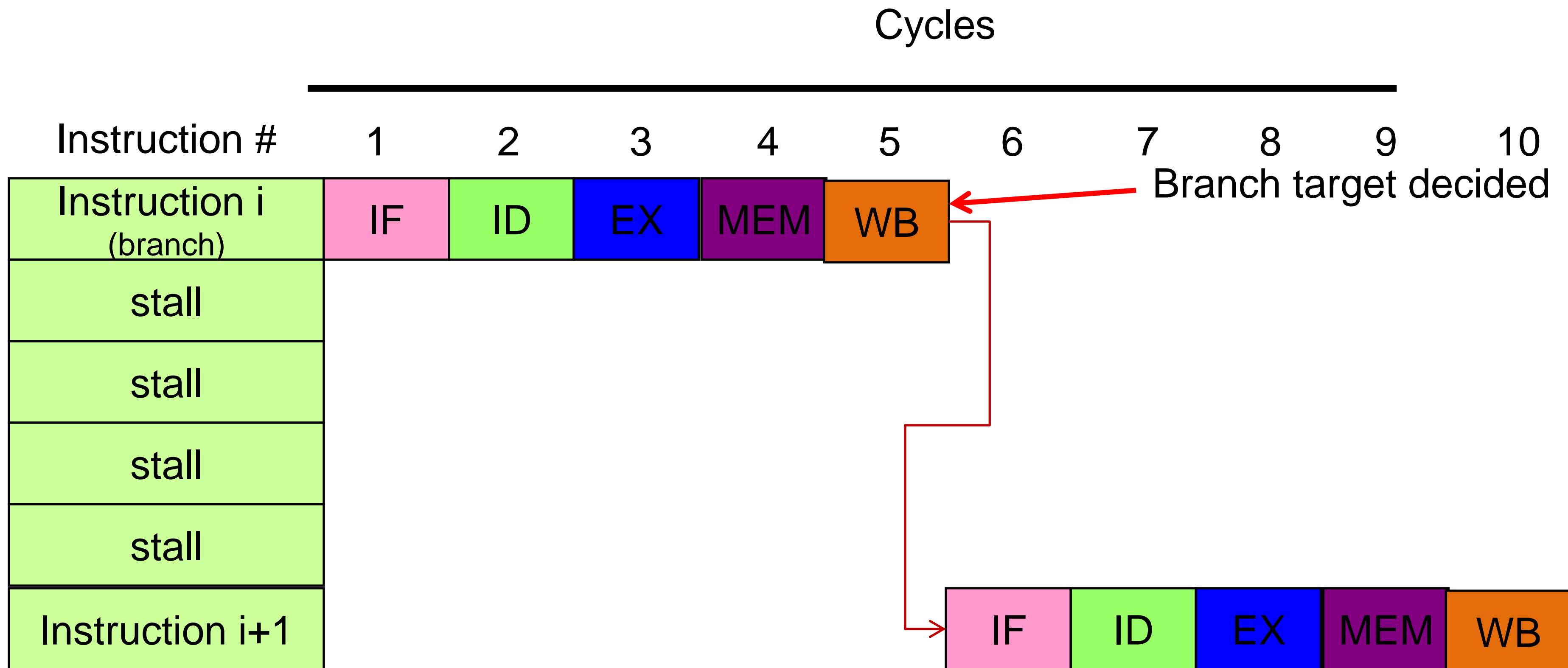
Reprinted with permission of Intel Corporation.

© Saman Amarasinghe 2008

Branch Prediction and Speculative Execution



Branch Prediction and Speculative Execution



Branch Prediction and Speculative Execution

Build a predictor to figure out which direction branch is going

- Today we have complex predictors with 99+% accuracy
- Even predict the address in indirect branches / returns

Fetch and speculatively execute from the predicted address

- No pipeline stalls

When the branch is finally decided, the speculative execution is confirmed or squashed

Intel® Core™ Microarchitecture – Branch Prediction

Complex predictor

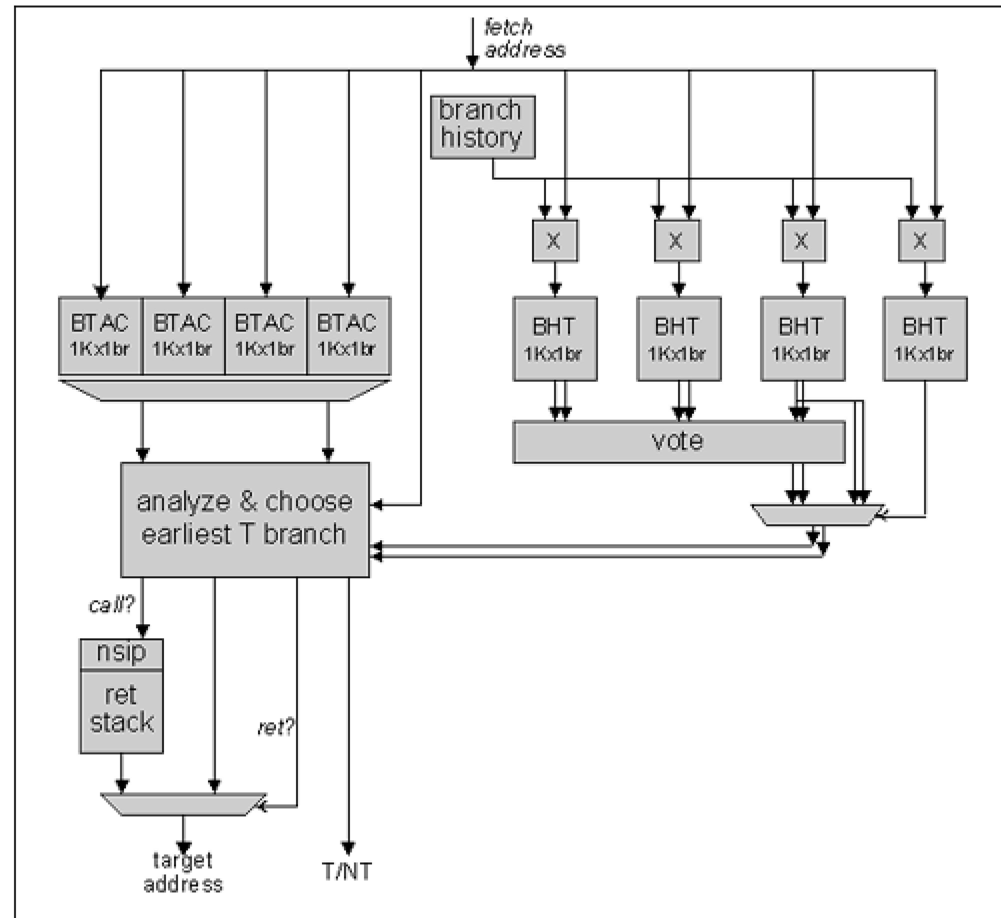
Multiple predictors

- Use branch history
- Different algorithms
- Vote at the end

Indirect address predictor

Return address predictor

Nehalem is even more complicated!



Reprinted with permission of Intel Corporation.

© Saman Amarasinghe 2008

Memory System

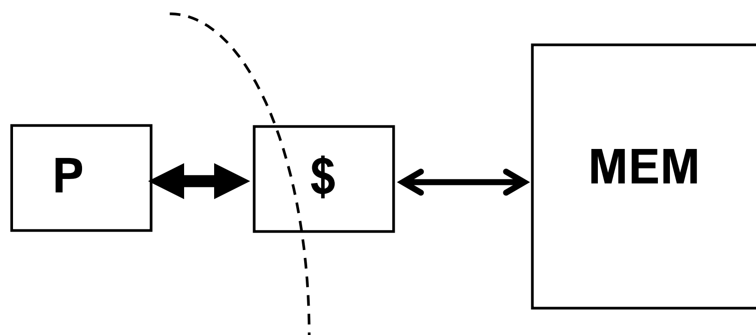
The Principle of Locality:

- Program access a relatively small portion of the address space at any instant of time.

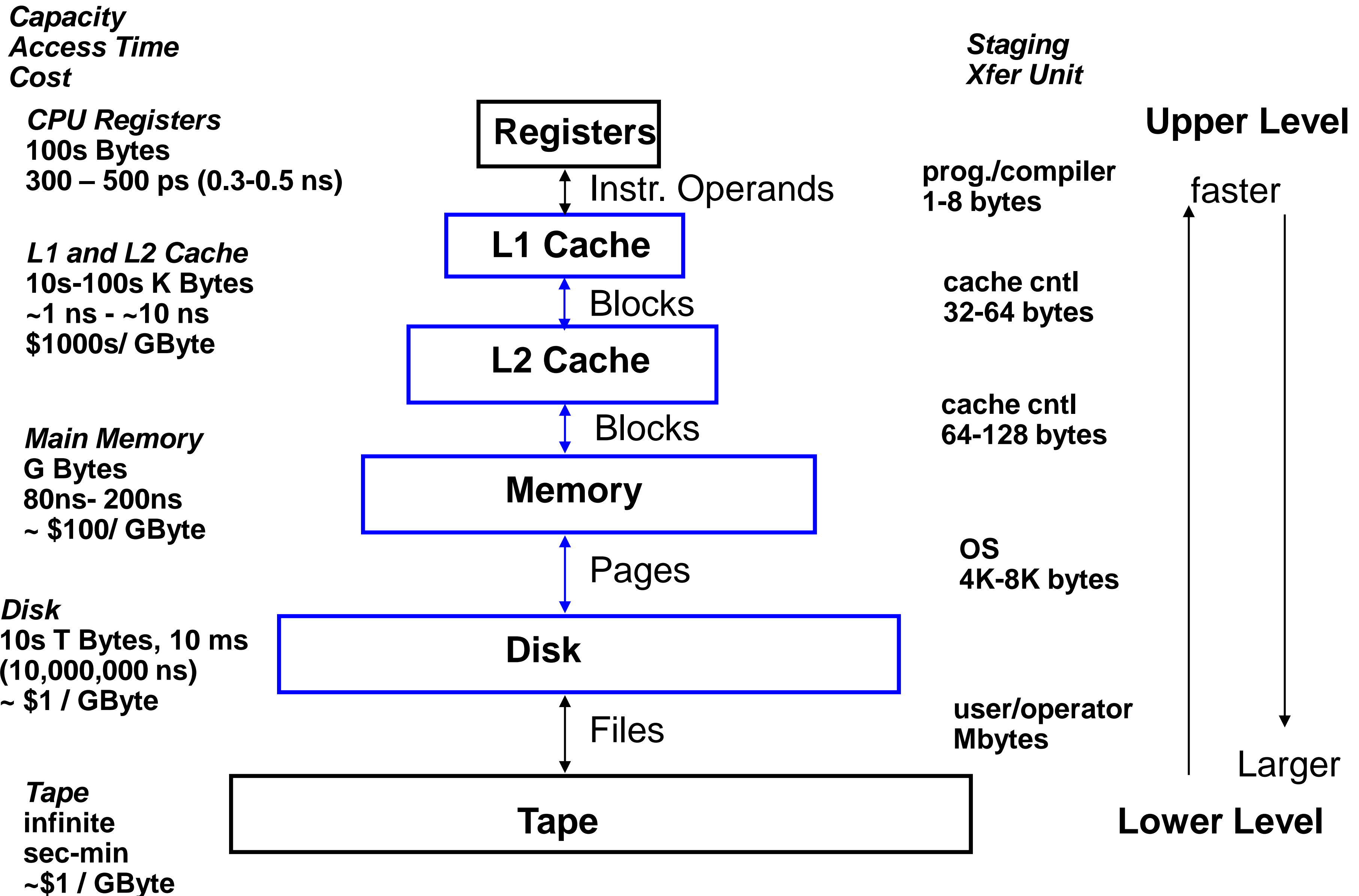
Two Different Types of Locality:

- Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
- Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon
(e.g., straight-line code, array access)

Last 30 years, HW relied on locality for memory perf.



Levels of the Memory Hierarchy



Cache Issues

Cold Miss

- The first time the data is available
- Prefetching may be able to reduce the cost

Capacity Miss

- The previous access has been evicted because too much data touched in between
- "Working Set" too large
- Reorganize the data access so reuse occurs before getting evicted.
- Prefetch otherwise

Conflict Miss

- Multiple data items mapped to the same location. Evicted even before cache is full
- Rearrange data and/or pad arrays

True Sharing Miss

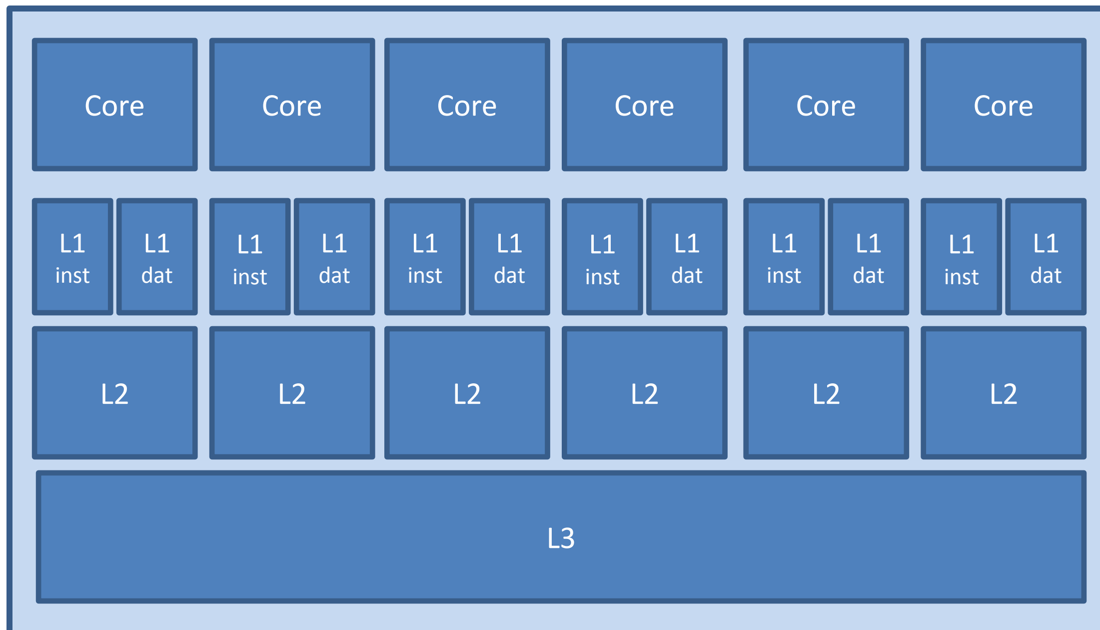
- Thread in another processor wanted the data, it got moved to the other cache
- Minimize sharing/locks

False Sharing Miss

- Other processor used different data in the same cache line. So the line got moved
- Pad data and make sure structures such as locks don't get into the same cache line

Intel® Nehalem™ Microarchitecture – Memory Sub-system

Intel 6 Core Processor



L1 Data Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	4 ns	8-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	4 ns	4-way
L2 Cache			
Size	Line Size	Latency	Associativty
256 KB	64 bytes	10 ns	8-way
L3 Cache			
Size	Line Size	Latency	Associativty
12 MB	64 bytes	50 ns	16-way
Main Memory			
Size	Line Size	Latency	Associativty
	64 bytes	75 ns	

Outline

Overview of Computer Architecture

Profiling a Program

Set of Example Programs

© Saman Amarasinghe 2008

Performance Analyzer

Helps you identify and characterize performance issues by:

- Collecting performance data from the system running your application.
- Organizing and displaying the data in a variety of interactive views, from system-wide down to source code or processor instruction perspective.
- Identifying potential performance issues and suggesting improvements.

- **Example: Intel Vtune, gprof, oprofile, perf**

Copyright © 2004 Intel Corporation

What Is a Hotspot?

Where in an application or system there is a significant amount of activity

- Where = address in memory
  => OS process
  => OS thread
  => executable file or *module*
  => user function (requires symbols)
  => line of source code (requires symbols with line numbers) or assembly instruction
- Significant = activity that occurs infrequently probably does not have much impact on system performance
- Activity = time spent or other internal processor event
  - Examples of other events: Cache misses, branch mispredictions, floating-point instructions retired, partial register stalls, and so on.

Copyright © 2004 Intel Corporation.

Two Ways to Track Location

Problem: I need to know where you spend <u>most</u> of your time.

Statistical Solution: I call you on your cellular phone every 30 minutes and ask you to report your location. Then I plot the data as a histogram.

Instrumentation Solution: I install a special phone booth at the entrance of every site you plan to visit. As you enter or exit every site, you first go into the booth, call the operator to get the exact time, and then call me and tell me where you are and when you got there.

Copyright © 2004 Intel Corporation.

Sampling Collector

Periodically interrupt the processor to obtain the execution context

- Time-based sampling (TBS) is triggered by:
  - Operating system timer services.
  - Every *n* processor clockticks.
- Event-based sampling (EBS) is triggered by processor event counter overflow.
  - These events are processor-specific, like L2 cache misses, branch mispredictions, floating-point instructions retired, and so on.

Copyright © 2004 Intel Corporation.

The Statistical Solution: Advantages

No Installation Required

- No need to install a phone everywhere you want a man in the field to make a report.

Wide Coverage

- Assuming all his territory has cellular coverage, you can track him wherever he goes.

Low Overhead

- Answering his cellular telephone once in a while, reporting his location, and returning to other tasks do not take much of his time.

Copyright © 2004 Intel Corporation.

The Statistical Solution:
Disadvantages

Approximate Precision:

- A number of factors can influence exactly how long he takes to answer the phone.

Limited Report:

- Insufficient time to find out how he got to where he is or where he has been since you last called him.

Statistical Significance: There are some places you might not locate him, if he does not go there often or he does not stay very long. Does that really matter?

Copyright © 2004 Intel Corporation.

The Instrumentation Solution: Advantages

Perfect Accuracy

- I know where you were immediately before and after your visit to each customer.
- I can calculate how much time you spent at each customer site.
- I know how many times you visited each customer site.

Copyright © 2004 Intel Corporation.

The Instrumentation Solution:
Disadvantages

Low Granularity

- Too coarse; the site is the site.

High Overhead

- You spend valuable time going to phone booths, calling operators, and calling me.

High Touch

- I have to build all those phone booths, which expands the space in each site you visit.

Copyright © 2004 Intel Corporation.

Events

Intel provide 100's of types of events

- Can be very confusing (ex: "number of bogus branches")
- Some useful event categories
  - Total instruction count and mix
  - Branch events
  - Load/store events
  - L1/L2 cache events
  - Prefetching events
  - TLB events
  - Multicore events

Copyright © 2004 Intel Corporation.

Use Event Ratios

In isolation, events may not tell you much.

Event ratios are dynamically calculated values based on events that make up the formula.

- Cycles per instruction (CPI) consists of clockticks and instructions retired.

There are a wide variety of predefined event ratios.

Copyright © 2004 Intel Corporation.

Outline

Overview of Computer Architecture

Profiling a Program

Set of Example Programs

© Saman Amarasinghe 2008

```
#define MAXA 10000
int maxa_half = MAXA/2;
int32_t  A[MAXA];

  //                [0, 1, 2, 3, 4, …]
int32_t  incA[MAXA];

//                  [ 0..MAXA-1 randomly]
int32_t  rndA[MAXA];
```

© Saman Amarasinghe 2008

Assembly listings

multiple passes over data

```
for(j=0; j<MAXA; j++)
    A[j] = A[j]+1;
```

```
        movl    $A, %eax
        movl    $A+40000, %edx
..B3.3:
        incl    (%rax)
        addq    $4, %rax
        cmpq    %rdx, %rax
        jl      ..B3.3
```

test j < maxa_half

```
for(j=0; j<MAXA; j++) {
    if(j < maxa_half)
        A[j] = A[j]+1;
}
```

```
        movl    maxa_half(%rip), %edx
        xorl    %ecx, %ecx
..B3.2:
        cmpl    %edx, %ecx
        jge     ..B3.4
..B3.3:
        movslq  %ecx, %rax
        incl    A(,%rax,4)
..B3.4:
        incl    %ecx
        cmpl    $10000, %ecx
        jl      ..B3.2
```

© Saman Amarasinghe 2008

Assembly listings

test div by 4

```
for(j=0; j<MAXA; j++) {
    if((j & 0x03) == 0)
        A[j] = A[j]+1;
}
```

```
        xorl     %edx, %edx
..B2.2:
        testb    $3, %dl
        jne      ..B2.4
..B2.3:
        movslq   %edx, %rax
        incl     A(,%rax,4)
..B2.4:
        incl     %edx
        cmpl     $10000, %edx
        jl       ..B2.2
```

test incA[i] < maxa_half

```
for(j=0; j<MAXA; j++) {
    if(incA[j] < maxa_half)
        A[j] = A[j]+1;
}
```

```
        movslq   maxa_half(%rip), %rdx
        xorl     %ecx, %ecx
        xorl     %eax, %eax
..B1.2:
        cmpq     incA(,%rcx,8),%rdx
        jle      ..B1.4
..B1.3:
        incl     A(%rax)
..B1.4:
        addq     $4, %rax
        addq     $1, %rcx
        cmpq     $10000, %rcx
        jl       ..B1.2
```

© Saman Amarasinghe 2008

results

	Runtime (ms)
multi pass over	**1.00**
test j < maxa_half	**1.26**
test div by 4	**2.21**
test incA[i] < maxa_half	**1.33**
test rndA[i] <maxa_half	**6.80**

© Saman Amarasinghe 2008

results

	Runtime (ms)	INST_RETIRED.ANY events
multi pass over	**1.00**	**1.00**
test j < maxa_half	**1.26**	**1.50**
test div by 4	**2.21**	**1.37**
test incA[i] < maxa_half	**1.33**	**1.63**
test rndA[i] <maxa_half	**6.80**	**1.63**

INST_RETIRED.ANY Instructions retired.

This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.

© Saman Amarasinghe 2008

results

	Runtime (ms)	INST_RETIRED.ANY events	INST_RETIRED.LOADS events	BR_INST_RETIRED.ANY events	Inst Retired (ANY - LOAD - BR)
multi pass over	1.00	1.00	1.00	1.00	1.00
test j < maxa_half	1.26	1.50	0.50	1.99	1.75
test div by 4	2.21	1.37	0.25	1.99	1.62
test incA[i] < maxa_half	1.33	1.63	1.50	2.00	1.50
test rndA[i] <maxa_half	6.80	1.63	1.50	1.99	1.50

INST_RETIRED.LOADS — **Instructions retired, contain a load**

INST_RETIRED.STORE — **Instructions retired, contain a store**

BR_INST_RETIRED.ANY — **Number of branch instructions retired**

© Saman Amarasinghe 2008

results

	Runtime (ms)	INST_RETIRED.ANY events	Clocks per Instructions Retired - CPI	CPI*Tot Instructions
multi pass over	1.00	1.00	1.00	1.00
test j < maxa_half	1.26	1.50	0.84	1.25
test div by 4	2.21	1.37	1.60	2.19
test incA[i] < maxa_half	1.33	1.63	0.82	1.33
test rndA[i] <maxa_half	6.80	1.63	4.17	6.78

CPI **CPU_CLK_UNHALTED.CORE / INST_RETIRED.ANY**

High CPI indicates that instructions require more cycles to execute than they should. In this case there may be opportunities to modify your code to improve the efficiency with which instructions are executed within the processor. CPI can get as low as 0.25 cycles per instructions.

© Saman Amarasinghe 2008

results

	Runtime (ms)	BR_INST_RETIRED.MIS PRED %
multi pass over	1.00	1.00
test j < maxa_half	1.26	2.50
test div by 4	2.21	400.00
test incA[i] < maxa_half	1.33	2.00
test rndA[i] <maxa_half	6.80	2134.00

BR_INST_RETIRED.MISPRED

This event counts the number of retired branch instructions that were mispredicted by the processor. A branch misprediction occurs when the processor predicts that the branch would be taken, but it is not, or vice-versa. ….

© Saman Amarasinghe 2008

results

	Runtime (ms)	INST_RETIRED.ANY events	BR_INST_RETIRED.MISPRED %	"Instructions wasted" of mispredicted branches	Total "Cost"
multi pass over	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
test j < maxa_half	**1.26**	**1.50**	**2.50**	**4.97**	**1.50**
test div by 4	**2.21**	**1.37**	**400.00**	**797.51**	**2.21**
test incA[i] < maxa_half	**1.33**	**1.63**	**2.00**	**3.99**	**1.63**
test rndA[i] <maxa_half	**6.80**	**1.63**	**2134.00**	**4254.69**	**6.10**

Assume the cost of a mispredicted branch is 21 "instructions wasted"

- Number 21 got the closest answer

© Saman Amarasinghe 2008

Accessing Memory

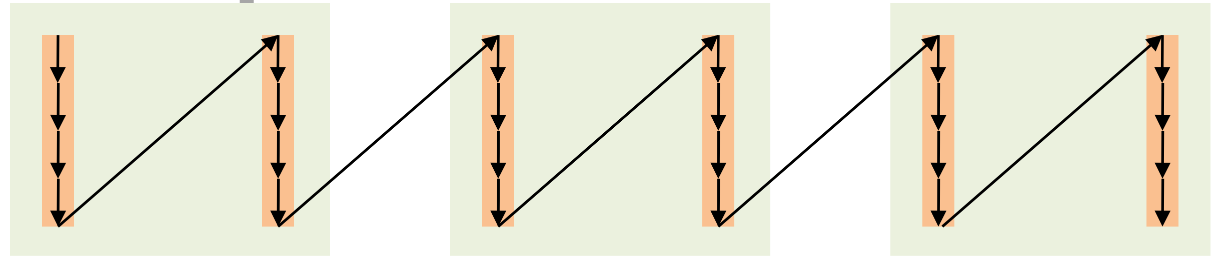
inner accumulate

```
for(j=0; j<MAXB; j++)
   for(i=0; i<DRV; i++)
      B[j] = B[j]+1;
```

© Saman Amarasinghe 2008

Memory access pattern

inner accumulate



Results

	Runtime (ms)	INST_RETIRED.ANY events	Clocks per Instructions Retired - CPI	*CPI*Tot Instructions*	INST_RETIRED.LOADS events	L1 Data Cache Miss Rate	L2_LINES_IN.SELF.DEMAND events	RESOURCE_STALLS.ANY %
Inner Accumulate	1.0	1.0	1.0	*1.0*	1	1	1	1
Multiple Passes	1.5	0.8	1.8	*1.5*	2	5	5	28
Blocked Access	0.9	0.8	1.0	*0.9*	2	5	2	10
Strided Access	8.8	0.8	10.4	*8.8*	2	167	202	220

© Saman Amarasinghe 2008

Indirect random access

```
for(i=0; i<DRV; i++)
   for(j=0; j<MAXB; j += MAXA)
      for(k=0; k<MAXA; k++)
        B[rndA[k]*MAXB/MAXA+k]
        = B[rndA[k]*MAXB/MAXA+k]+1;
```

© Saman Amarasinghe 2008

Results

	Runtime (ms)	INST_RETIRED.ANY events	Clocks per Instructions Retired - CPI	*CPI*Tot Instructions*	INST_RETIRED.LOADS events	L1 Data Cache Miss Rate	L2_LINES_IN.SELF.DEMAND events	RESOURCE_STALLS.ANY %
Inner Accumulate	1.0	1.0	1.0	*1.0*	1.0	1.0	1.0	1.0
indirect random access	8.1	2.7	3.0	*8.3*	4.0	53.3	0.0	180.0
indirect strided access	1.5	1.1	1.5	*1.6*	4.0	12.7	6.0	25.0
pointer chase	12.4	0.8	14.9	*12.5*	4.0	41.7	43.0	320.0
pointer chase -- post randomize	146	0.9	174	*149*	4.0	194	7752	3927

© Saman Amarasinghe 2008

Compiler Heroics

What else can be done

- Deeper optimizations (see lecture 2)
- Vectorization for SSE
- Loop interchange

	O1	O2		O3	
multi pass over	101.78	32.25	3.16	13.09	7.78
test j < maxa_half	127.28	34.78	3.66	37.10	3.43
test div by 3	223.43	37.24	6.00	20.53	10.88
test incA[i] < maxa_half	134.50	134.11	1.00	152.27	0.88
test rndA[i] <maxa_half	687.44	658.72	1.04	154.06	4.46
Inner Accumulate	158.59	64.02	2.48	64.03	2.48
Multiple Passes	242.08	233.43	1.04	64.20	3.77
Blocked Access	136.79	82.27	1.66	82.00	1.67
Strided Access	1392.56	1400.19	0.99	81.96	16.99
indirect random access	1315.34	1308.75	1.01	1350.17	0.97
indirect strided access	248.37	250.55	0.99	250.48	0.99
pointer chase	2005.12	2002.99	1.00	2007.83	1.00
pointer chase -- post randomize	23581.20	23603.33	1.00	23559.83	1.00

© Saman Amarasinghe 2008

MIT OpenCourseWare
http://ocw.mit.edu

6.172 Performance Engineering of Software Systems
Fall 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.