Lecture: Parallel Architecture – Thread Level Parallelism and Data Level Parallelism

CSCE 569 Parallel Computing

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
http://cse.sc.edu/~yanyh

Topics

- Introduction
- Programming on shared memory system (Chapter 7)
  - **OpenMP**
- Principles of parallel algorithm design (Chapter 3)
- Programming on large scale systems (Chapter 6)
  - **MPI (point to point and collectives)**
  - Introduction to PGAS languages, UPC and Chapel
- Analysis of parallel program executions (Chapter 5)
  - **Performance Metrics for Parallel Systems**
    - **Execution Time, Overhead, Speedup, Efficiency, Cost**
  - **Scalability of Parallel Systems**
  - **Use of performance tools**

Topics

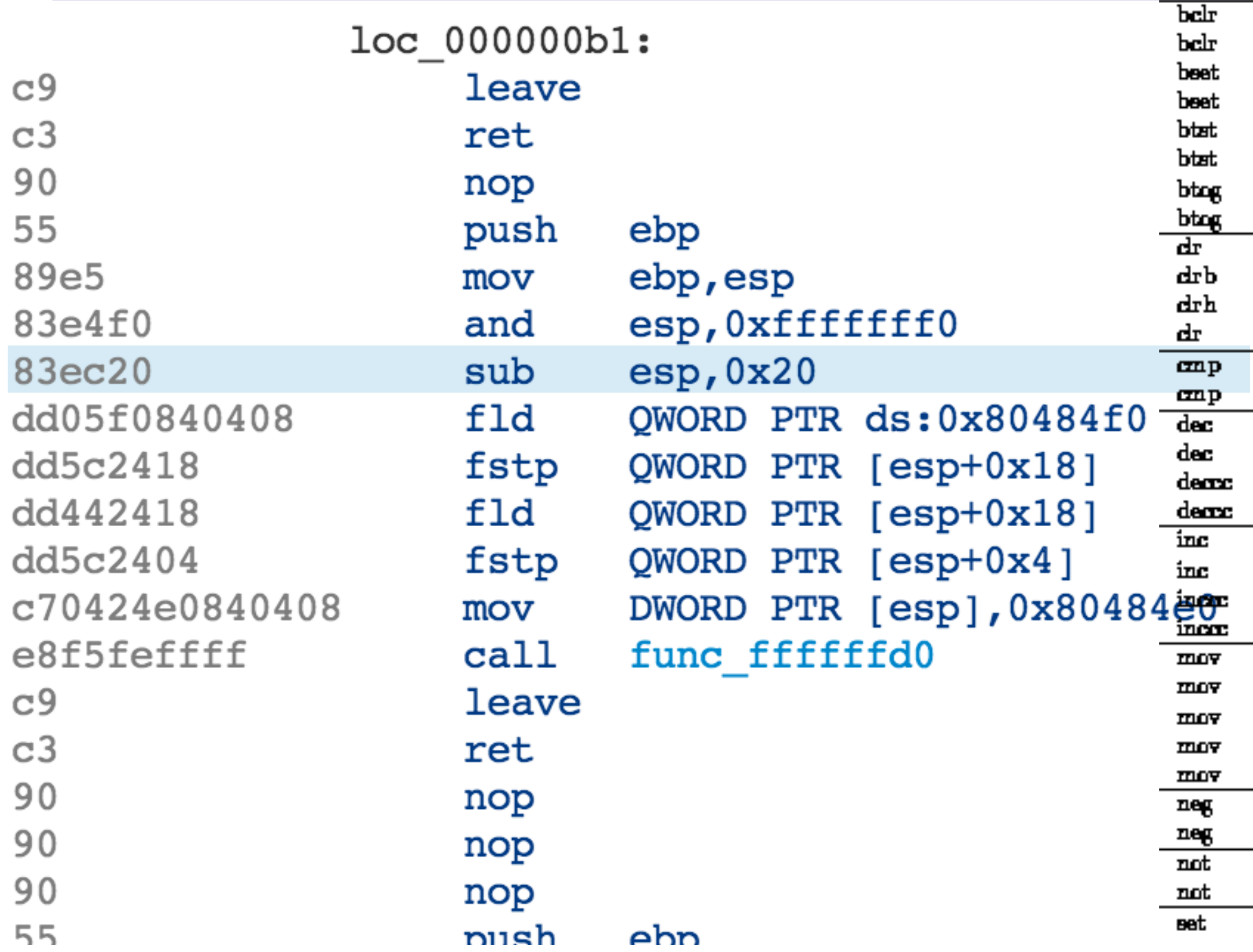
- ~~Programming on shared memory system (Chapter 7)~~
  - ~~*Cilk/Cilkplus and OpenMP Tasking*~~
  - ~~*PThread, mutual exclusion, locks, synchronizations*~~
- ☛ Parallel architectures and memory
  - **Parallel computer architectures**
    - **Thread Level Parallelism**
    - **Data Level Parallelism**
    - ~~**Synchronization**~~
  - **Memory hierarchy and cache coherency**
- Manycore GPU architectures and programming
  - **GPUs architectures**
  - **CUDA programming**
  - Introduction to offloading model in OpenMP

Lecture: Parallel Architecture – Thread Level Parallelism

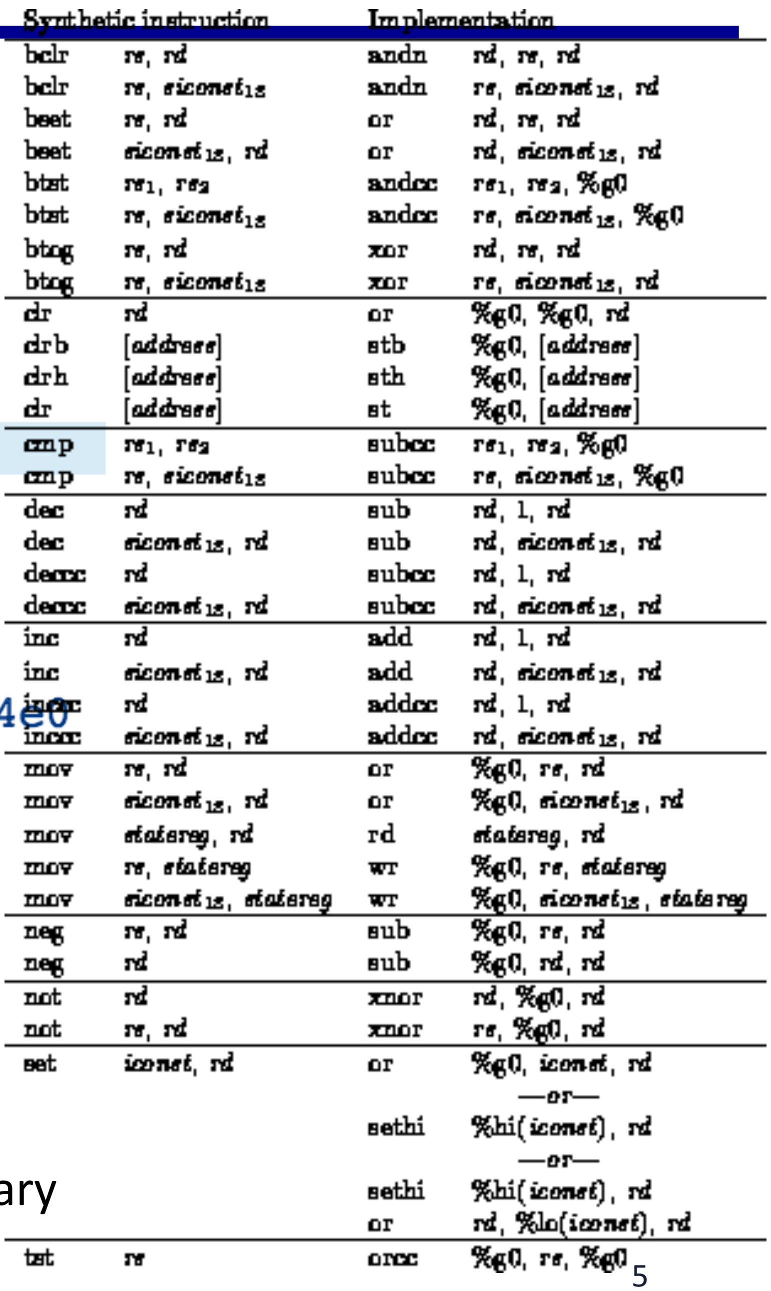
Note:

- Parallelism in hardware
- Not (just) multi-/many-core architecture

Binary Code and Instructions



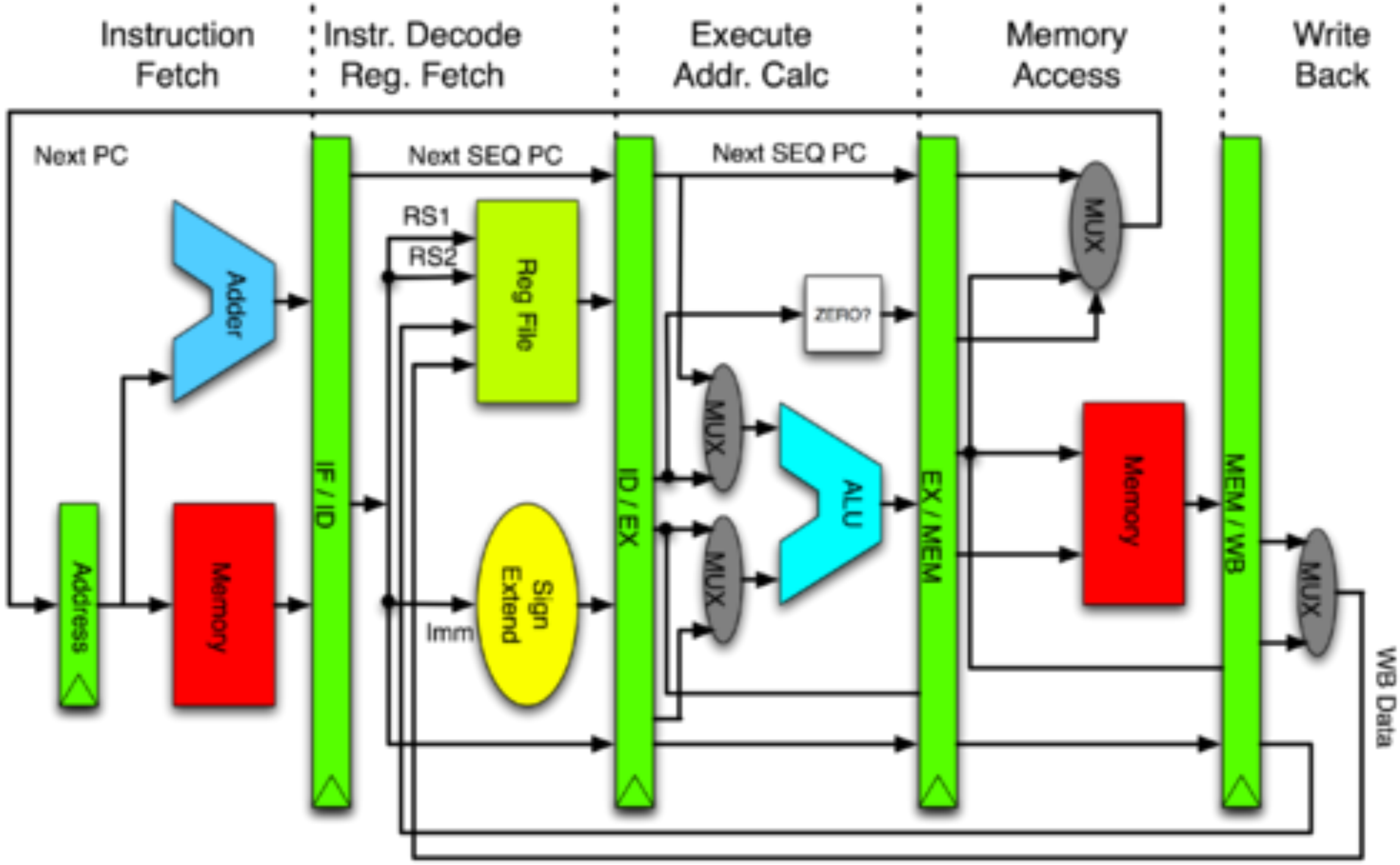
```
                loc_000000b1:
c9                      leave
c3                      ret
90                      nop
55                      push    ebp
89e5                    mov     ebp,esp
83e4f0                  and     esp,0xfffffff0
83ec20                  sub     esp,0x20
dd05f0840408            fld     QWORD PTR ds:0x80484f0
dd5c2418                fstp    QWORD PTR [esp+0x18]
dd442418                fld     QWORD PTR [esp+0x18]
dd5c2404                fstp    QWORD PTR [esp+0x4]
c70424e0840408          mov     DWORD PTR [esp],0x80484e0
e8f5feffff              call    func_ffffffd0
c9                      leave
c3                      ret
90                      nop
90                      nop
90                      nop
55                      push    ebp
```



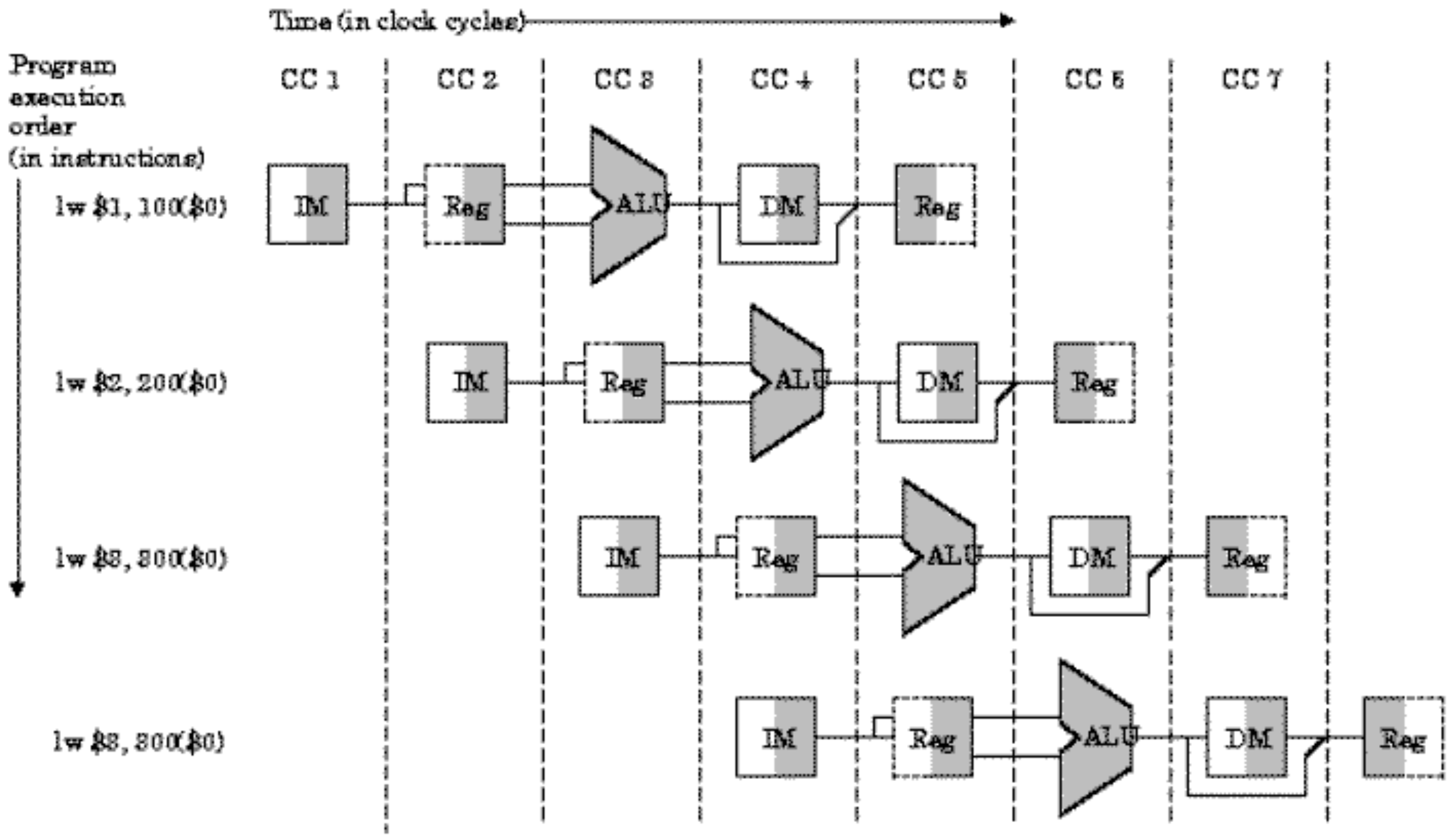
Synthetic instruction		Implementation	
bclr	*rs*, *rd*	andn	*rd*, *rs*, *rd*
bclr	*rs*, *siconst*$_{13}$	andn	*rs*, *siconst*$_{13}$, *rd*
bset	*rs*, *rd*	or	*rd*, *rs*, *rd*
bset	*siconst*$_{13}$, *rd*	or	*rd*, *siconst*$_{13}$, *rd*
btst	*rs*$_1$, *rs*$_2$	andcc	*rs*$_1$, *rs*$_2$, %g0
btst	*rs*, *siconst*$_{13}$	andcc	*rs*, *siconst*$_{13}$, %g0
btog	*rs*, *rd*	xor	*rd*, *rs*, *rd*
btog	*rs*, *siconst*$_{13}$	xor	*rs*, *siconst*$_{13}$, *rd*
clr	*rd*	or	%g0, %g0, *rd*
clrb	[*address*]	stb	%g0, [*address*]
clrh	[*address*]	sth	%g0, [*address*]
clr	[*address*]	st	%g0, [*address*]
cmp	*rs*$_1$, *rs*$_2$	subcc	*rs*$_1$, *rs*$_2$, %g0
cmp	*rs*, *siconst*$_{13}$	subcc	*rs*, *siconst*$_{13}$, %g0
dec	*rd*	sub	*rd*, 1, *rd*
dec	*siconst*$_{13}$, *rd*	sub	*rd*, *siconst*$_{13}$, *rd*
deccc	*rd*	subcc	*rd*, 1, *rd*
deccc	*siconst*$_{13}$, *rd*	subcc	*rd*, *siconst*$_{13}$, *rd*
inc	*rd*	add	*rd*, 1, *rd*
inc	*siconst*$_{13}$, *rd*	add	*rd*, *siconst*$_{13}$, *rd*
inccc	*rd*	addcc	*rd*, 1, *rd*
inccc	*siconst*$_{13}$, *rd*	addcc	*rd*, *siconst*$_{13}$, *rd*
mov	*rs*, *rd*	or	%g0, *rs*, *rd*
mov	*siconst*$_{13}$, *rd*	or	%g0, *siconst*$_{13}$, *rd*
mov	*statereg*, *rd*	rd	*statereg*, *rd*
mov	*rs*, *statereg*	wr	%g0, *rs*, *statereg*
mov	*siconst*$_{13}$, *statereg*	wr	%g0, *siconst*$_{13}$, *statereg*
neg	*rs*, *rd*	sub	%g0, *rs*, *rd*
neg	*rd*	sub	%g0, *rd*, *rd*
not	*rd*	xnor	*rd*, %g0, *rd*
not	*rs*, *rd*	xnor	*rs*, %g0, *rd*
set	*iconst*, *rd*	or	%g0, *iconst*, *rd*
			—*or*—
		sethi	%hi(*iconst*), *rd*
			—*or*—
		sethi	%hi(*iconst*), *rd*
		or	*rd*, %lo(*iconst*), *rd*
tst	*rs*	orcc	%g0, *rs*, %g0

- Compile a program using -save-temps flags to see the binary code or disassembly a binary code using objdump -D

Stages to Execute an Instruction

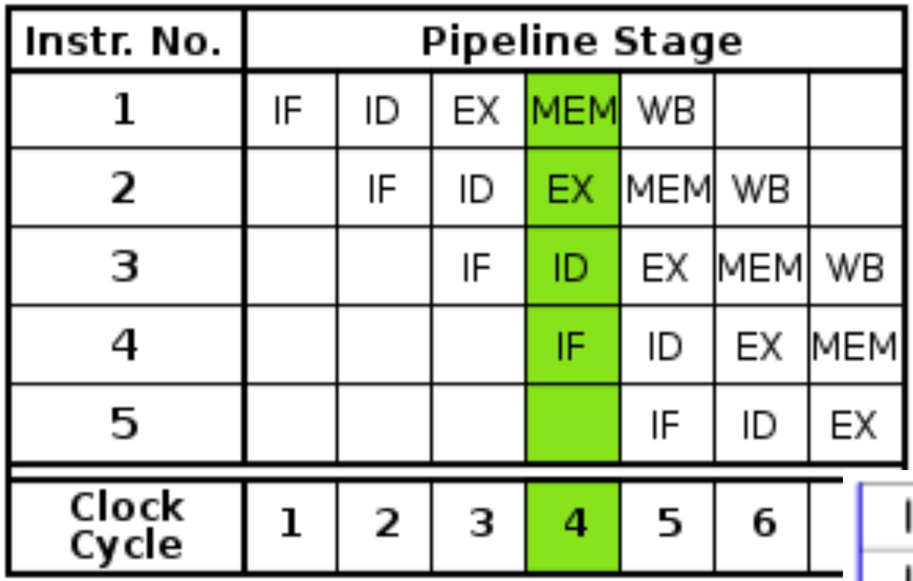


Pipeline

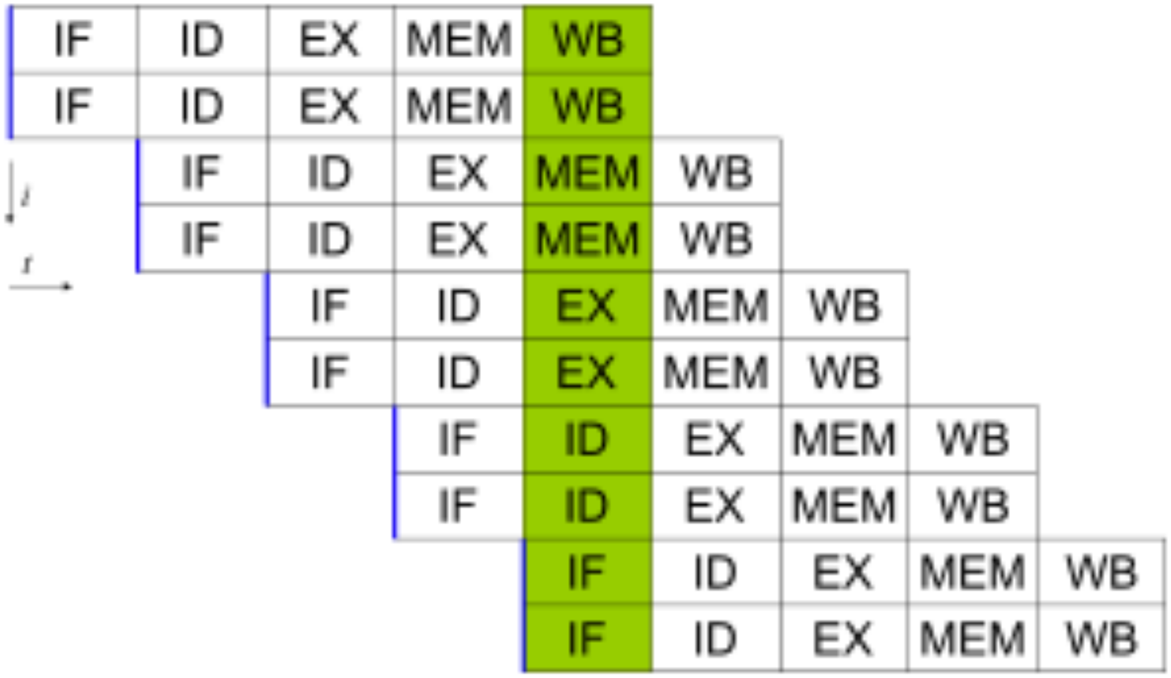


Time (in clock cycles)
Program execution order (in instructions)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
lw $3, 300($0)
IM
Reg
ALU
DM
Reg

Pipeline and Superscalar



Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	



IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

What do we do with that many transistors?

- Optimizing the execution of a single instruction stream through
  - Pipelining
    - Overlap the execution of multiple instructions
    - Example: all RISC architectures; Intel x86 underneath the hood
  - Out-of-order execution:
    - Allow instructions to overtake each other in accordance with code dependencies (RAW, WAW, WAR)
    - Example: all commercial processors (Intel, AMD, IBM, SUN)
  - Branch prediction and speculative execution:
    - Reduce the number of stall cycles due to unresolved branches
    - Example: (nearly) all commercial processors

What do we do with that many transistors? (II)

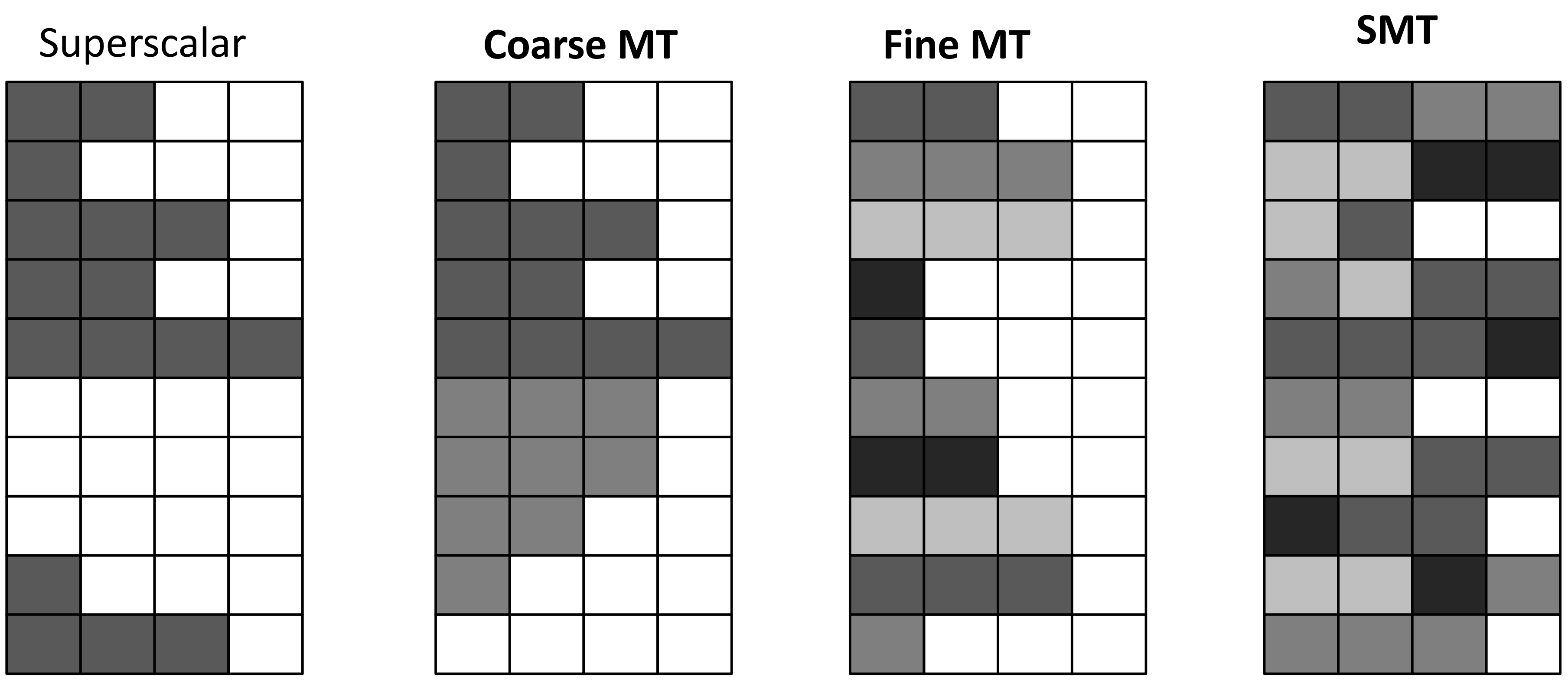
- Multi-issue processors:
  - Allow multiple instructions to start execution per clock cycle
  - Superscalar (Intel x86, AMD, …) vs. VLIW architectures
- VLIW/EPIC architectures:
  - Allow compilers to indicate independent instructions per issue packet
  - Example: Intel Itanium
- Vector units:
  - Allow for the efficient expression and execution of vector operations
  - Example: SSE - SSE4, AVX instructions

Limitations of optimizing a single instruction stream (II)

- Problem: within a single instruction stream we do not find enough independent instructions to execute simultaneously due to
  - data dependencies
  - limitations of speculative execution across multiple branches
  - difficulties to detect memory dependencies among instruction (alias analysis)
- Consequence: significant number of functional units are idling at any given time
- Question: Can we maybe execute instructions from another instructions stream
  - Another thread?
  - Another process?

Hardware Multi-Threading (SMT)

- Three types of hardware multi-threading (single-core only):
  - Coarse-grained MT
  - Fine-grained MT
  - Simultaneous Multi-threading



Thread-level parallelism

- Problems for executing instructions from multiple threads at the same time
  - The instructions in each thread might use the same register names
  - Each thread has its own program counter
- Virtual memory management allows for the execution of multiple threads and sharing of the main memory
- When to switch between different threads:
  - Fine grain multithreading: switches between every instruction
  - Course grain multithreading: switches only on costly stalls (e.g. level 2 cache misses)

Simultaneous Multi-threading

- Convert Thread-level parallelism to instruction-level parallelism
- Dynamically scheduled processors already have most hardware mechanisms in place to support SMT (e.g. register renaming)
- Required additional hardware:
  - Register file per thread
  - Program counter per thread
- Operating system view:
  - If a CPU supports *n* simultaneous threads, the Operating System views them as *n* processors
  - OS distributes most time consuming threads ‘fairly’ across the *n* processors that it sees.

Example for SMT architectures (I)

- Intel Hyperthreading:
  - First released for Intel Xeon processor family in 2002
  - Supports two architectural sets per CPU,
  - Each architectural set has its own
    - General purpose registers
    - Control registers
    - Interrupt control registers
    - Machine state registers
  - Adds less than 5% to the relative chip size

    Reference: D.T. Marr et. al. “Hyper-Threading Technology Architecture and Microarchitecture”, Intel Technology Journal, 6(1), 2002, pp.4-15. ftp://download.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf

Example for SMT architectures (II)

- IBM Power 5
  - Same pipeline as IBM Power 4 processor but with SMT support
  - Further improvements:
    - Increase associativity of the L1 instruction cache
    - Increase the size of the L2 and L3 caches
    - Add separate instruction prefetch and buffering units for each SMT
    - Increase the size of issue queues
    - Increase the number of virtual registers used internally by the processor.

Simultaneous Multi-Threading

- Works well if
  - Number of compute intensive threads does not exceed the number of threads supported in SMT
  - Threads have highly different characteristics (e.g. one thread doing mostly integer operations, another mainly doing floating point operations)
- Does not work well if
  - Threads try to utilize the same function units
  - Assignment problems:
    - e.g. a dual processor system, each processor supporting 2 threads simultaneously (OS thinks there are 4 processors)
    - 2 compute intensive application processes might end up on the same processor instead of different processors (OS does not see the difference between SMT and real processors!)

Lecture: Parallel Architecture -- Data Level Parallelism

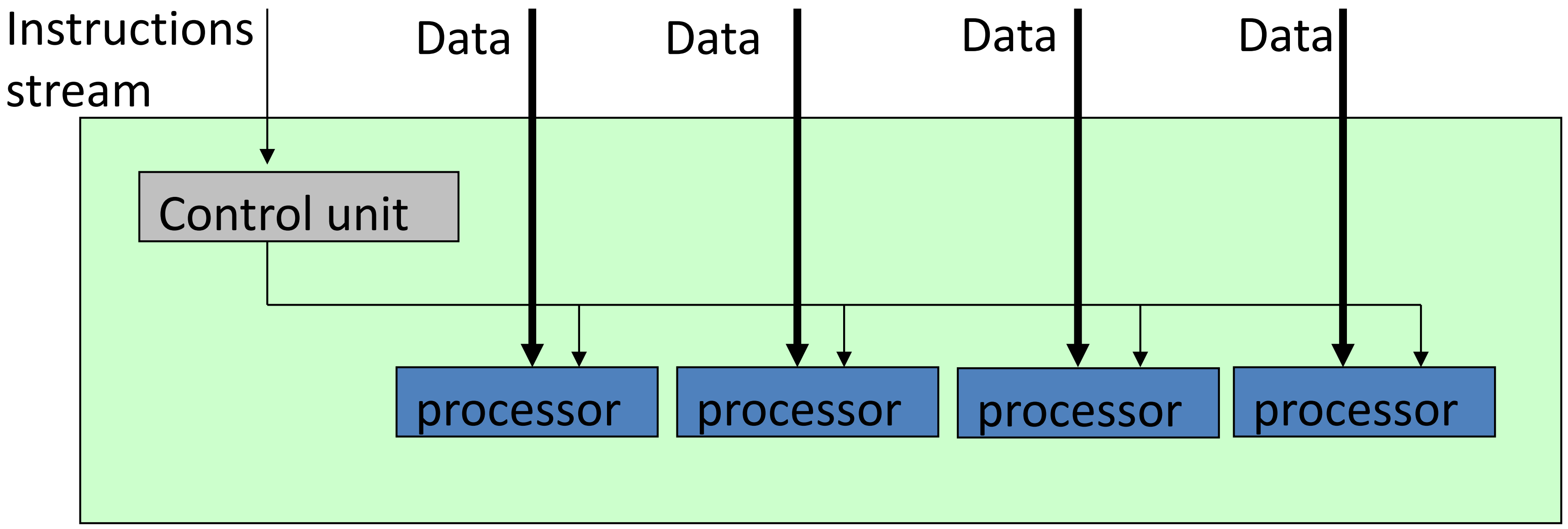
Classification of Parallel Architectures

Flynn's Taxonomy

- **SISD: Single instruction single data**
  - **Classical von Neumann architecture**
- **SIMD: Single instruction multiple data**
  - **Vector, GPU, etc**
- MISD: Multiple instructions single data
  - Non existent, just listed for completeness
- **MIMD: Multiple instructions multiple data**
  - **Most common and general parallel machine**
  - **Multi-/many- processors/cores/threads/computers**

Single Instruction Multiple Data

- Also known as Array-processors
- A single instruction stream is broadcasted to multiple processors, each having its own data stream
  - Still used in some graphics cards today



SIMD In Real

- **Hardware for data-level parallelism**

```
for (i=0; i<n; i++) {
 a[i] = b[i] + c[i];
}
```

- **Three major implementations**
  - **SIMD extensions to conventional CPU**
  - **Vector architectures**
  - ***GPU variant***

SIMD Instructions

- Originally developed for Multimedia applications
- Same operation executed for multiple data items
- Uses a fixed length register and partitions the carry chain to allow utilizing the same functional unit for multiple operations
  - E.g. a 64 bit adder can be utilized for two 32-bit add operations simultaneously
- Instructions originally not intended to be used by compiler, but just for handcrafting specific operations in device drivers
- All elements in a register have to be on the same memory page to avoid page faults within the instruction

Intel SIMD Instructions

- MMX (Mult-Media Extension) - 1996
  - Existing 64 bit floating point register could be used for eight 8-bit operations or four 16-bit operations
- SSE (Streaming SIMD Extension) – 1999
  - Successor to MMX instructions
  - Separate 128-bit registers added for sixteen 8-bit, eight 16-bit, or four 32-bit operations
- SSE2 – 2001, SSE3 – 2004, SSE4 - 2007
  - Added support for double precision operations
- AVX (Advanced Vector Extensions) - 2010
  - 256-bit registers added

Vector Processors

- Vector processors abstract operations on vectors, e.g. replace the following loop

```
for (i=0; i<n; i++) {
 a[i] = b[i] + c[i];
}
```

  by

```
a = b + c; ➡ ADDV.D V10, V8, V6
```

- Some languages offer high-level support for these operations (e.g. Fortran90 or newer)

AVX Instructions

AVX Instruction	Description
`VADDPD`	Add four packed double-precision operands
`VSUBPD`	Subtract four packed double-precision operands
`VMULPD`	Multiply four packed double-precision operands
`VDIVPD`	Divide four packed double-precision operands
`VFMADDPD`	Multiply and add four packed double-precision operands
`VFMSUBPD`	Multiply and subtract four packed double-precision operands
`VCMPxx`	Compare four packed double-precision operands for `EQ, NEQ, LT, LTE, GT, GE…`
`VMOVAPD`	Move aligned four packed double-precision operands
`VBROADCASTSD`	Broadcast one double-precision operand to four locations in a 256-bit register

Main concepts

- Advantages of vector instructions
  - A single instruction specifies a great deal of work
  - Since each loop iteration must not contain data dependence to other loop iterations
    - No need to check for data hazards between loop iterations
    - Only one check required between two vector instructions
    - Loop branches eliminated

Basic vector architecture

- A modern vector processor contains
  - Regular, pipelined scalar units
  - Regular scalar registers
  - Vector units – (inventors of pipelining!)
  - Vector register: can hold a fixed number of entries (e.g. 64)
  - Vector load-store units

Comparison MIPS code vs. vector code

Example: `Y=aX+Y` for 64 elements

```
   L.D    F0, a           /* load scalar a*/
   DADDIU R4, Rx, #512    /* last address */
L: L.D    F2, 0(Rx)       /* load X(i) */
   MUL.D  F2, F2, F0      /* calc. a times X(i)*/
   L.D    F4, 0(Ry)       /* load Y(i) */
   ADD.D  F4, F4, F2      /* aX(I) + Y(i) */
   S.D    F4, 0(Ry)       /* store Y(i) */
   DADDIU Rx, Rx, #8      /* increment X*/
   DADDIU Ry, Ry, #8      /* increment Y */
   DSUBU  R20, R4, Rx     /* compute bound */
   BNEZ   R20, L
```

Comparison MIPS code vs. vector code (II)

Example: `Y=aX+Y` for 64 elements

```
L.D     F0, a              /* load scalar a*/
LV      V1, 0(Rx)          /* load vector X */
MULVS.D   V2, V1, F0       /* vector scalar mult*/
LV      V3, 0(Ry)          /* load vector Y */
ADDV.D  V4, V2, V3         /* vector add */
SV      V4, 0(Ry)          /* store vector Y */
```

Vector length control

- What happens if the length is not matching the length of the vector registers?
- A vector-length register (VLR) contains the number of elements used within a vector register
- *Strip mining*: split a large loop into loops less or equal the maximum vector length (MVL)

Vector length control (II)

```
low =0;
VL  = (n mod MVL);
for (j=0; j < n/MVL; j++ ) {
  for (i=low; i < low + VL; i++ ) {
       Y(i) = a * X(i) + Y(i);
  }
  low += VL;
  VL   = MVL;
}
```

Vector stride

- Memory on typically organized in multiple banks
  - Allow for independent management of different memory addresses
  - Memory bank time an order of magnitude larger than CPU clock cycle
- Example: assume 8 memory banks and 6 cycles of memory bank time to deliver a data item
  - Overlapping of multiple data requests by the hardware

Vector stride (II)

- What happens if the code does not access subsequent elements of the vector

```
for (i=0; i<n; i+=2) {
 a[i] = b[i] + c[i];
}
```

  – Vector load ‘compacts’ the data items in the vector register (gather)
    - No affect on the execution of the loop
    - You might however use only a subset of the memory banks -> longer load time
    - Worst case: stride is a multiple of the number of memory banks

Conditional execution

- Consider the following loop

```
for (i=0; i< N; i++ ) {
    if ( A(i) != 0 ) {
    A(i) = A(i) - B(i);
    }
}
```

- Loop can usually not been vectorized because of the conditional statement
- Vector-mask control: boolean vector of length MLV to control whether an instruction is executed or not
  - Per element of the vector

Conditional execution (II)

```
LV        V1, Ra    /* load vector A into V1 */
LV        V2, Rb    /* load vector B into V2 */
L.D       F0, #0    /* set F0 to zero */
SNEVS.D   V1, F0    /* set VM(i)=1 if V1(i)!=F0 */
SUBV.D    V1, V1, V2  /* sub using vector mask*/
CVM                /* clear vector mask to 1 */
SV     V1, Ra   /* store V1 */
```

Support for sparse matrices

- Access of non-zero elements in a sparse matrix often described by

  `A(K(i)) = A(K(i)) + C (M(i))`

  - K(i) and M(i) describe which elements of A and C are non-zero
  - Number of non-zero elements have to match, location not necessarily
- Gather-operation: take an index vector and fetch the according elements using a base-address
  - Mapping from a non-contiguous to a contiguous representation
- Scatter-operation: inverse of the gather operation

Support for sparse matrices (II)

```
LV     Vk, Rk    /* load index vector K into V1 */
LVI    Va, (Ra+Vk) /* Load vector indexed A(K(i)) */
LV     Vm, Rm    /* load index vector M into V2 */
LVI    Vc, (Rc+Vm) /* Load vector indexed C(M(i)) */
ADDV.D Va, Va, Vc /* set VM(i)=1 if V1(i)!=F0 */
SVI    Va, (Ra+Vk) /* store vector indexed A(K(i)) */
```

- Note:
  - Compiler needs the explicit hint, that each element of K is pointing to a distinct element of A
  - Hardware alternative: a hash table keeping track of the address acquired
    - Start of a new vector iteration (convoy) as soon as an address appears the second time

Lecture: Parallel Architecture – Synchronization between processors, cores and HW threads

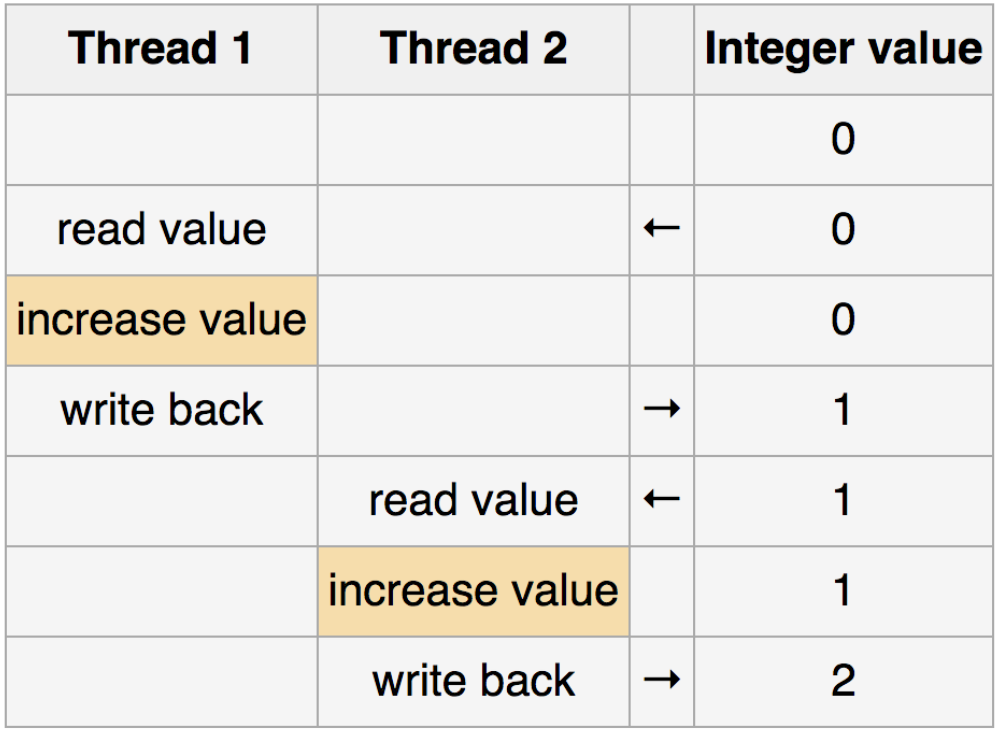
Synchronization between processors

- Required on all levels of multi-threaded programming
  - Lock/unlock
  - Mutual exclusion
  - Barrier synchronization

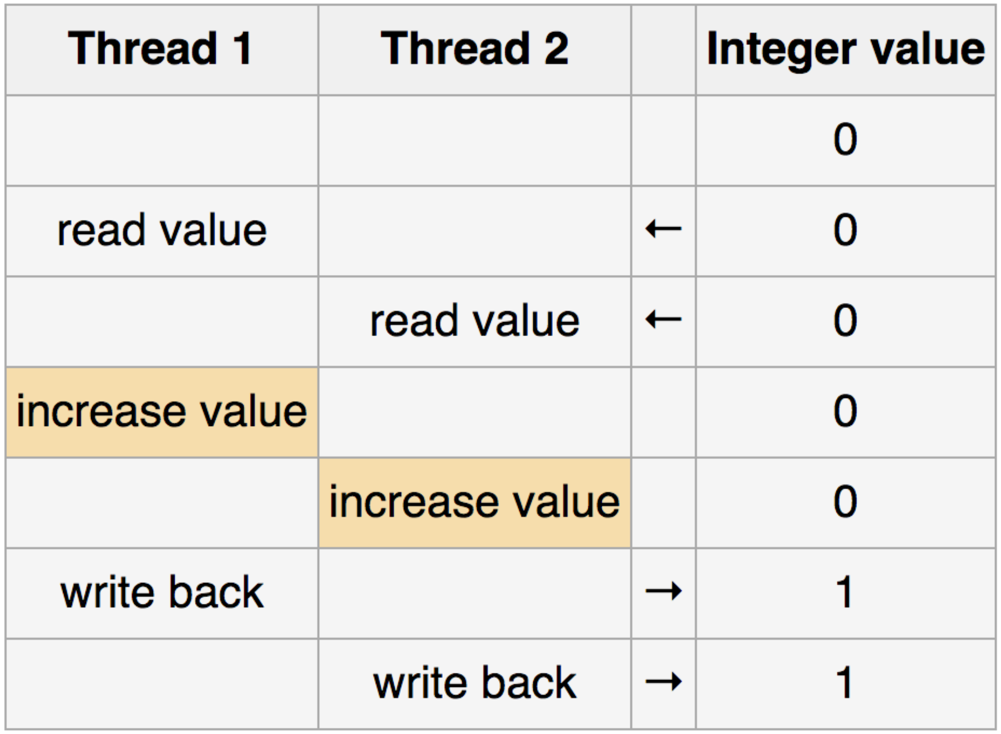
- Key hardware capability: *cp++
  - Uninterruptable instruction capable of automatically retrieving or changing a value

Race Condition

```
int count = 0;
int * cp = &count;
....
*cp++; /* by two threads */
```



Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2



Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Pictures from wikipedia: http://en.wikipedia.org/wiki/Race_condition

Simple Example (IIIb)

```
void *thread_func (void *arg){
   int * cp (int *) arg;

   pthread_mutex_lock (&mymutex);
     *cp++;       // read, increment and write shared variabl
   pthread_mutex_unlock (&mymutex);

   return NULL;
}
```

Synchronization

- Lock/unlock operations on the hardware level, e.g.
  - Lock returning 1 if lock is free/available
  - Lock returning 0 if lock is unavailable
- Implementation using *atomic exchange (compare and swap)*
  - Process sets the value of a register/memory location to the required operation
  - Setting the value must not be interrupted in order to avoid race conditions
  - Access by multiple processes/threads will be resolved by write serialization

Synchronization (II)

- Other synchronization primitives:
  - Test-and-set
  - Fetch-and-increment
- Problems with all three algorithms:
  - Require a read and write operation in a single, uninterruptable sequence
  - Hardware can not allow any operations between the read and the write operation
  - Complicates cache coherence
  - Must not deadlock

Load linked/store conditional

- Pair of instructions where the second instruction returns a value indicating, whether the pair of instructions was executed as if the instructions were atomic
- Special pair of load and store operations
  - *Load linked (LL)*
  - *Store conditional (SC):* returns 1 if successful, 0 otherwise
- Store conditional returns an error if
  - Contents of memory location specified by LL changed before calling SC
  - Processor executes a context switch

Load linked/store conditional (II)

- Assembler code sequence to atomically exchange the contents of register R4 and the memory location specified by R1

```
try:  MOV   R3, R4
   LL R2, 0(R1)
   SC R3, 0(R1)
   BEQZ  R3, try
   MOV   R4, R2
```

Load linked/store conditional (III)

- Implementing fetch-and-increment using load linked and conditional store

```
try: LL     R2, 0(R1)
     DADDUI R3, R2, #1
     SC     R3, 0(R1)
     BEQZ   R3, try
```

- Implementation of LL/SC by using a special Link Register, which contains the address of the operation

Spin locks

- A lock that a processor continuously tries to acquire, spinning around in a loop until it succeeds.
- Trivial implementation

```
        DADDUI   R2, R0, #1
lockit:  EXCH       R2, 0(R1)    !atomic exchange
        BNEZ     R2, lockit
```

- Since the EXCH operation includes a read and a modify operation
  - Value will be loaded into the cache
    - Good if only one processor tries to access the lock
    - Bad if multiple processors in an SMP try to get the lock (cache coherence)
  - EXCH includes a write attempt, which will lead to a write-miss for SMPs

Spin locks (II)

- For cache coherent SMPs, slight modification of the loop required

```
lockit:  LD   R2, 0(R1)   !load the lock
      BNEZ   R2, lockit !lock available?
      DADDUI R2, R0, #1 !load locked value
      EXCH   R2, 0(R1)  !atomic exchange
      BNEZ   R2, lockit !EXCH successful?
```

Spin locks (III)

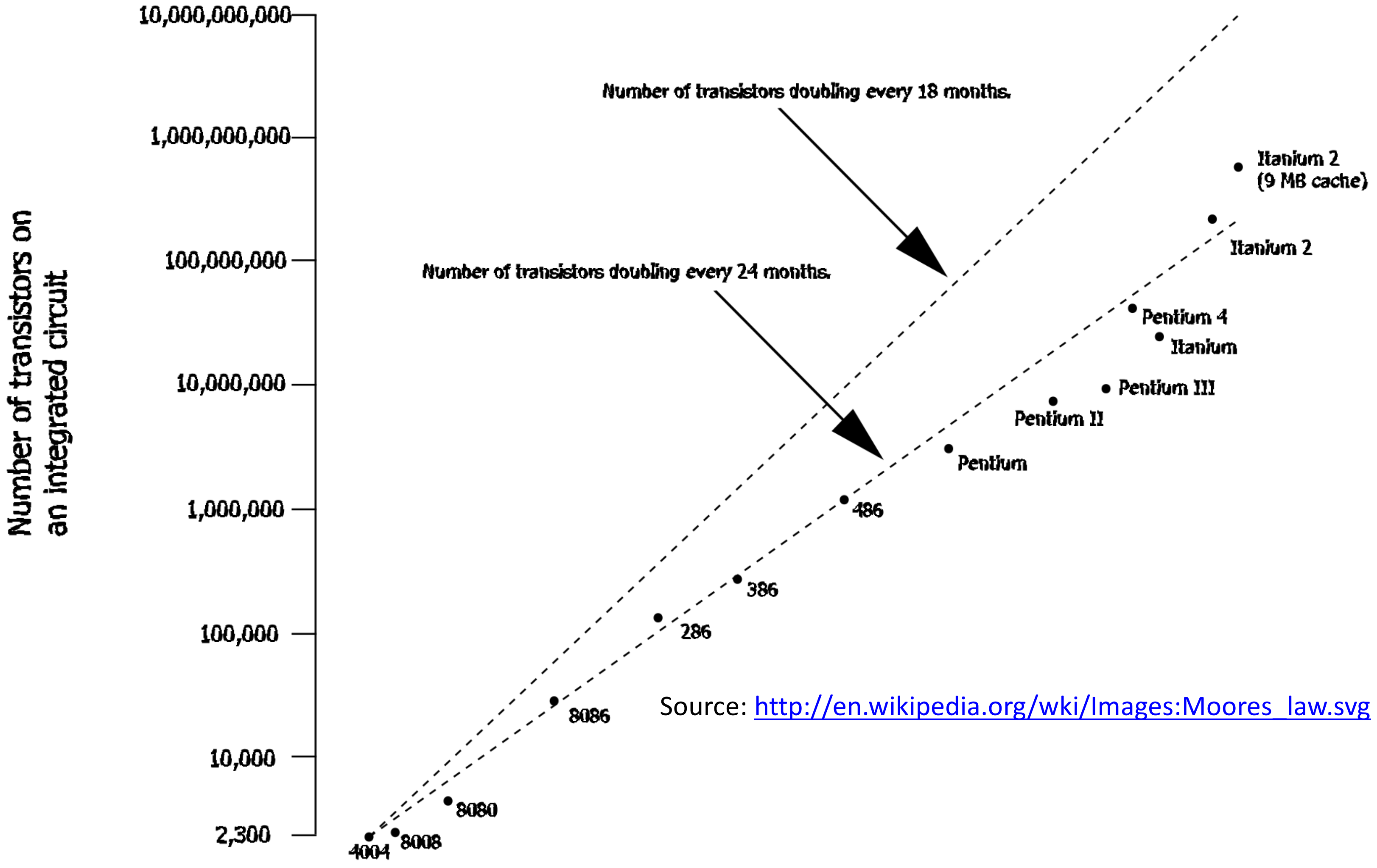
- …or using LL/SC

```
lockit:  LL  R2, 0(R1)   !load the lock
      BNEZ   R2, lockit !lock available?
      DADDUI R2, R0, #1 !load locked value
      SC  R2, 0(R1)   !atomic exchange
      BNEZ   R2, lockit !SC successful?
```

Lecture: Parallel Architecture – Moore’s Law

Moore’s Law

- Long-term trend on the number of transistor per integrated circuit
- Number of transistors double every ~18 month



Source: http://en.wikipedia.org/wki/Images:Moores_law.svg

The “Future” of Moore’s Law

- The chips are down for Moore’s law
  - http://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338
- Special Report: 50 Years of Moore's Law
  - http://spectrum.ieee.org/static/special-report-50-years-of-moores-law
- Moore’s law really is dead this time
  - http://arstechnica.com/information-technology/2016/02/moores-law-really-is-dead-this-time/
- Rebooting the IT Revolution: A Call to Action (SIA/SRC, 2015)
  - https://www.semiconductors.org/clientuploads/Resources/RITR%20WEB%20version%20FINAL.pdf