Projects

- Work together for the implementation
  - Discussion and debugging, but not the code itself
- Each submit your own implementation and report
- Presentation
  - One presentation
- Additional meeting time

Name	
Aditi Patil	Project 1
Aparna Puram	Project 1
Erik Hoggard	Project 1
Jacques Breaux	Project 1
Karam Abughalieh	Project 2
Kevin Weinert	Project 2
Mark Easterly	Project 2
Nathan Sketch	Project 2
Shayan Mukhtar	Project 2

Lecture 16: Parallel Architecture – Thread Level Parallelism

Concurrent and Multicore Programming

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan

Topics (Part 1)

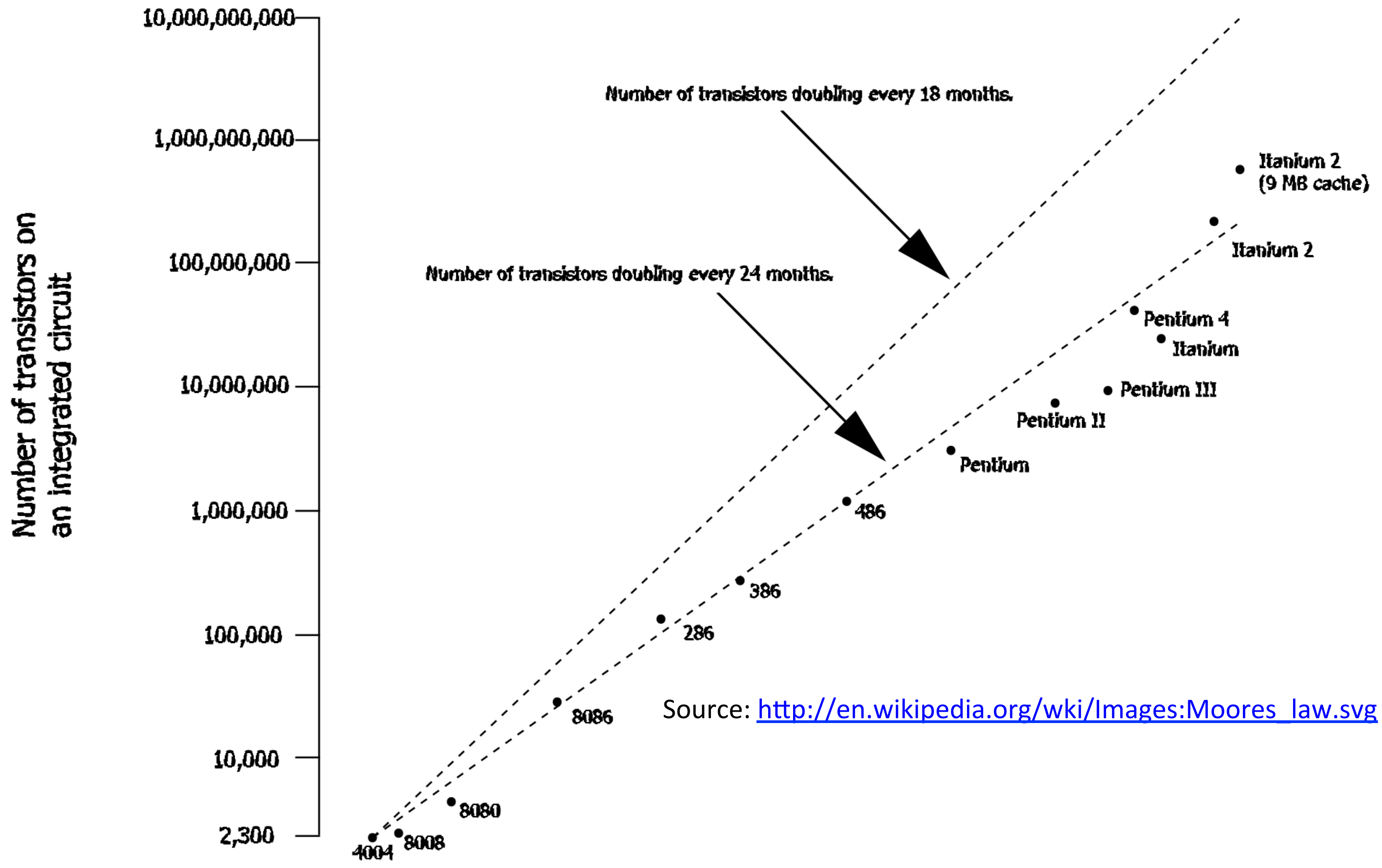
- Introduction
- Principles of parallel algorithm design (Chapter 3)
- Programming on shared memory system (Chapter 7)
  - **OpenMP**
  - **Cilk/Cilkplus**
  - **PThread, mutual exclusion, locks, synchronizations**
- Analysis of parallel program executions (Chapter 5)
  - **Performance Metrics for Parallel Systems**
    - **Execution Time, Overhead, Speedup, Efficiency, Cost**
  - **Scalability of Parallel Systems**
  - **Use of performance tools**

Topics (Part 2)

- ☛ Parallel architectures and hardware
  - **Parallel computer architectures**
    - **Thread level parallelism and data level parallelism**
  - **Memory hierarchy and cache coherency**
- Manycore GPU architectures and programming
  - **GPUs architectures**
  - **CUDA programming**
  - Introduction to offloading model in OpenMP
- Programming on large scale systems (Chapter 6)
  - **MPI (point to point and collectives)**
  - Introduction to PGAS languages, UPC and Chapel
- Parallel algorithms (Chapter 8,9 &10)

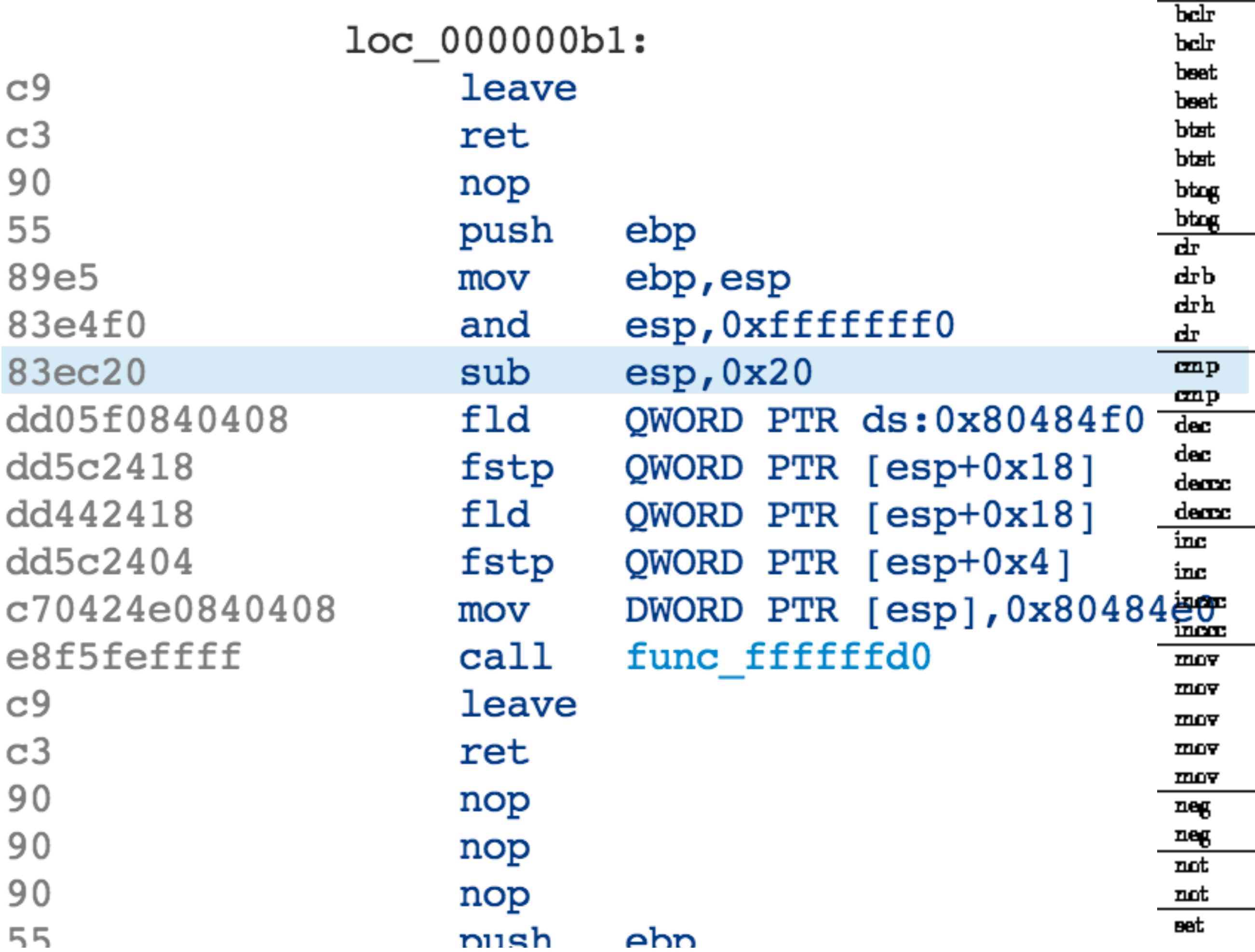
Moore's Law

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

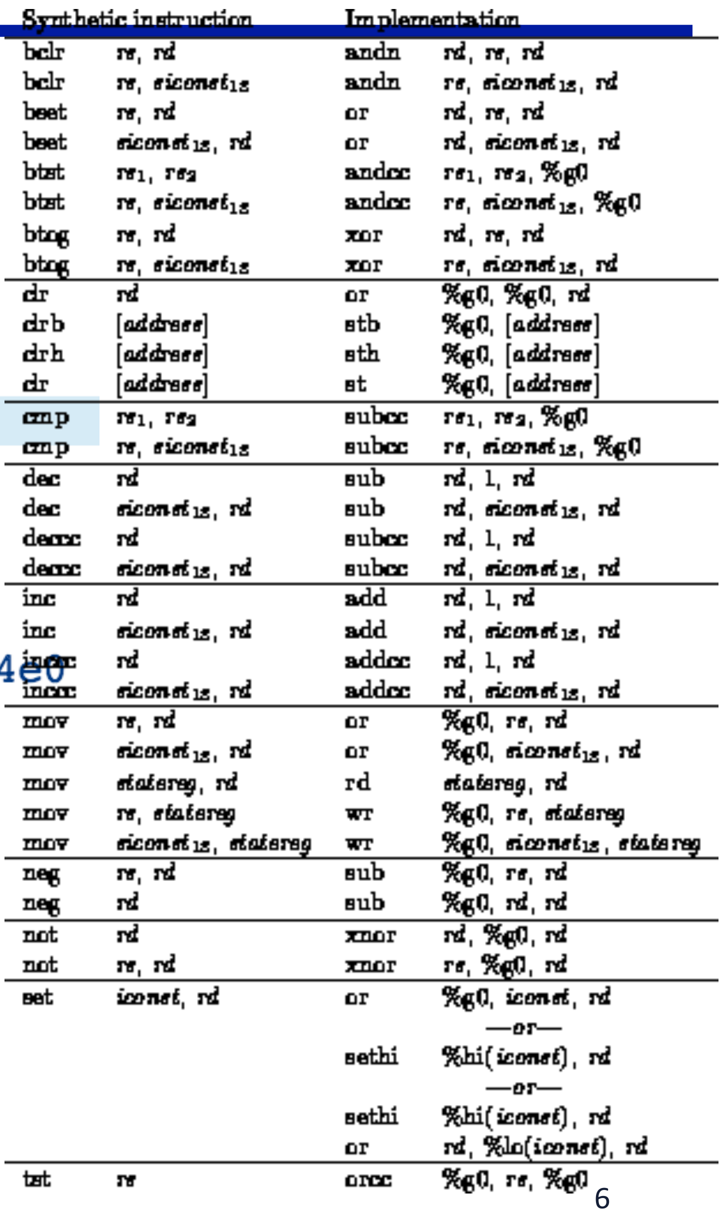


Source: http://en.wikipedia.org/wki/Images:Moores_law.svg

Binary Code and Instructions

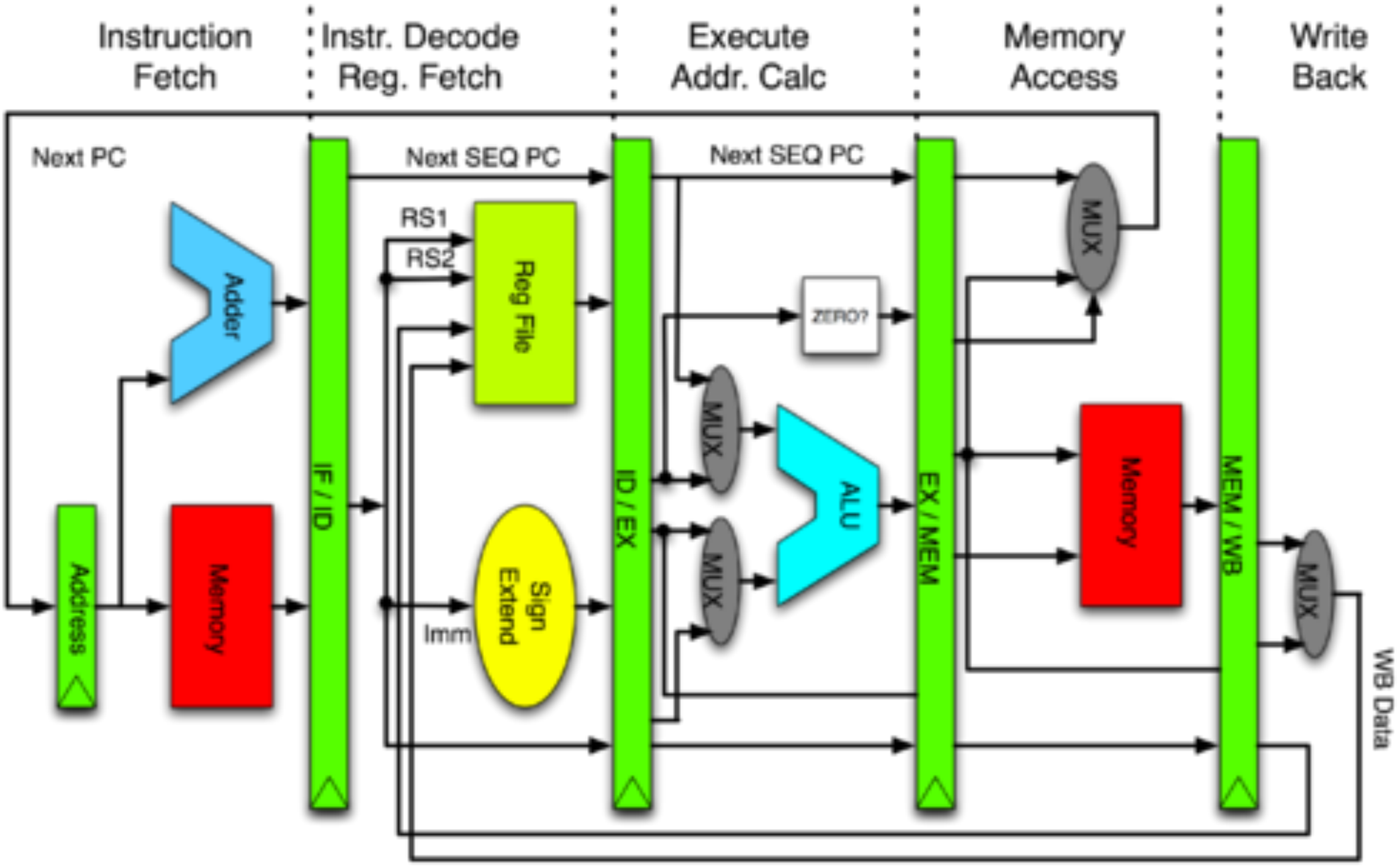


```
                    loc_000000b1:
c9                         leave
c3                         ret
90                         nop
55                         push    ebp
89e5                       mov     ebp,esp
83e4f0                     and     esp,0xfffffff0
83ec20                     sub     esp,0x20
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
```

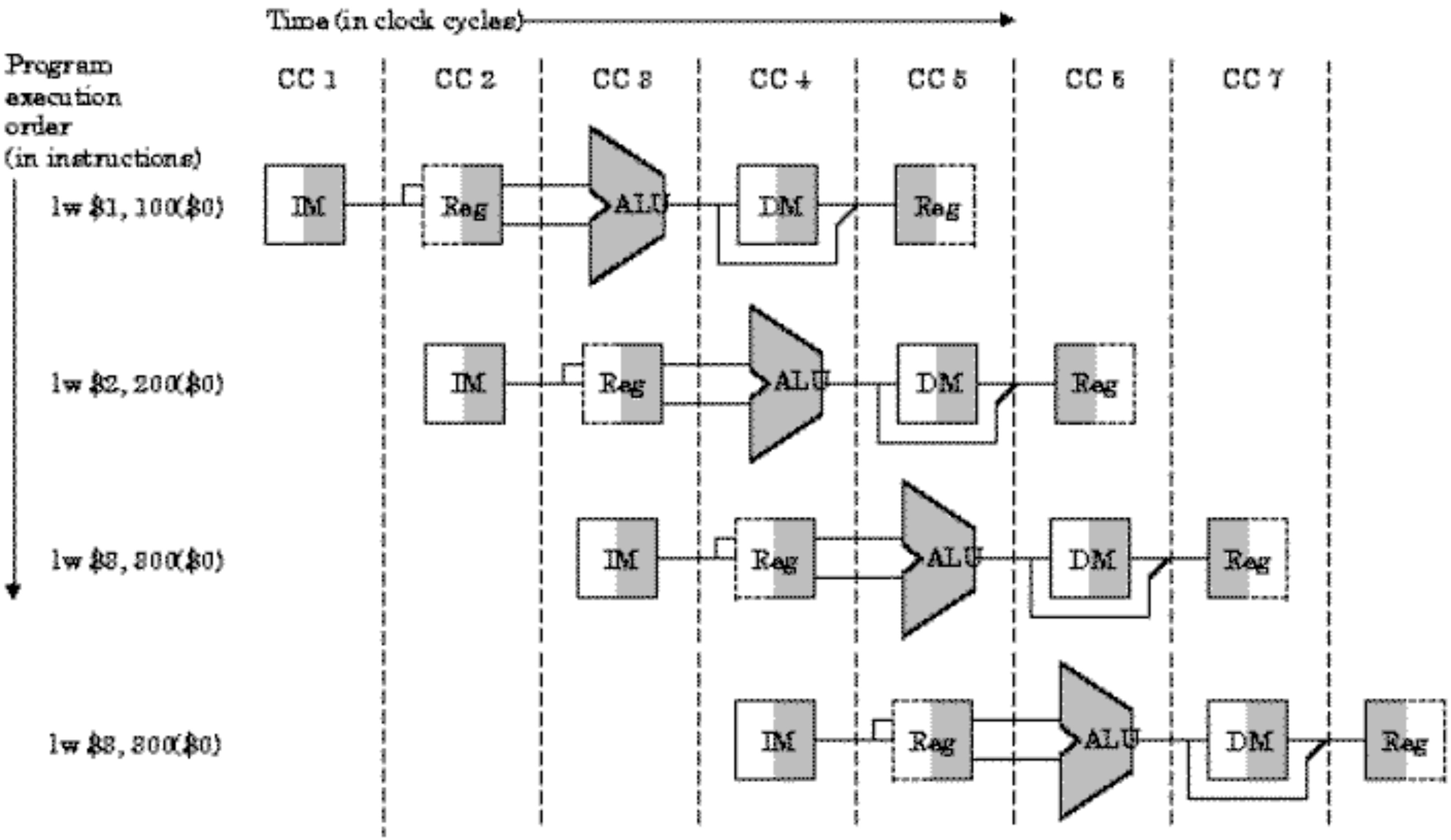


Synthetic instruction		Implementation	
bclr	rs, rd	andn	rd, rs, rd
bclr	rs, siconst$_{13}$	andn	rs, siconst$_{13}$, rd
bset	rs, rd	or	rd, rs, rd
bset	siconst$_{13}$, rd	or	rd, siconst$_{13}$, rd
btst	rs$_1$, rs$_2$	andcc	rs$_1$, rs$_2$, %g0
btst	rs, siconst$_{13}$	andcc	rs, siconst$_{13}$, %g0
btog	rs, 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

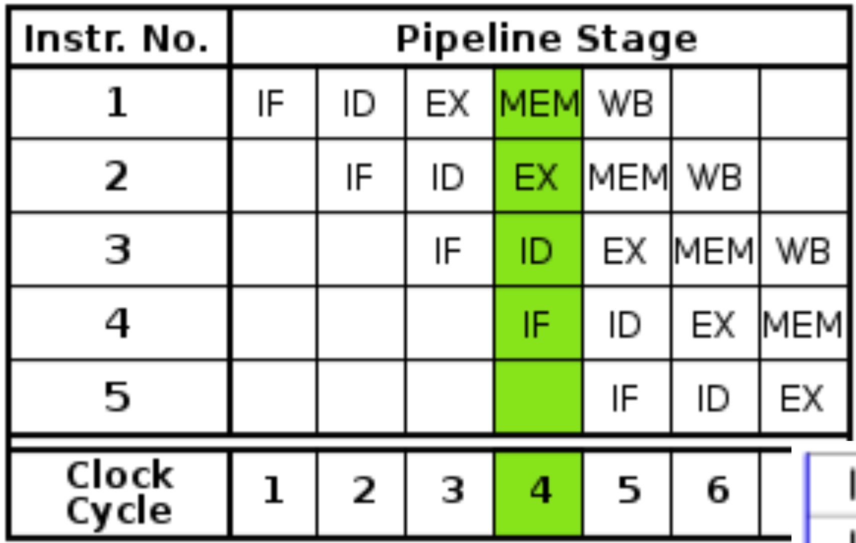
Stages to Execute an Instruction



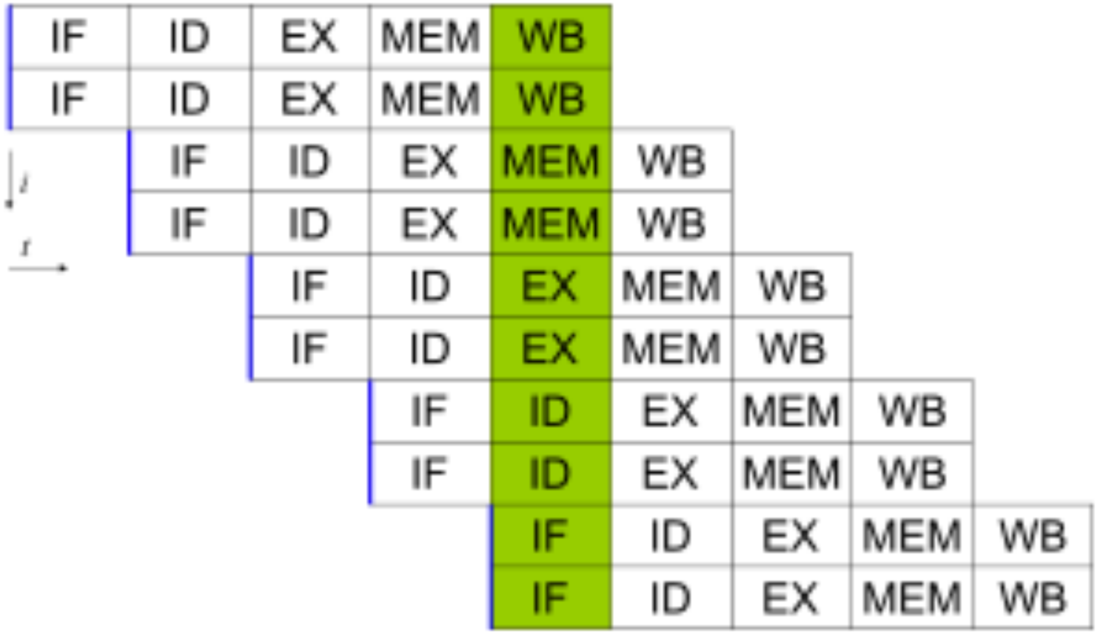
Pipeline


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?

The “Future” of Moore’s Law

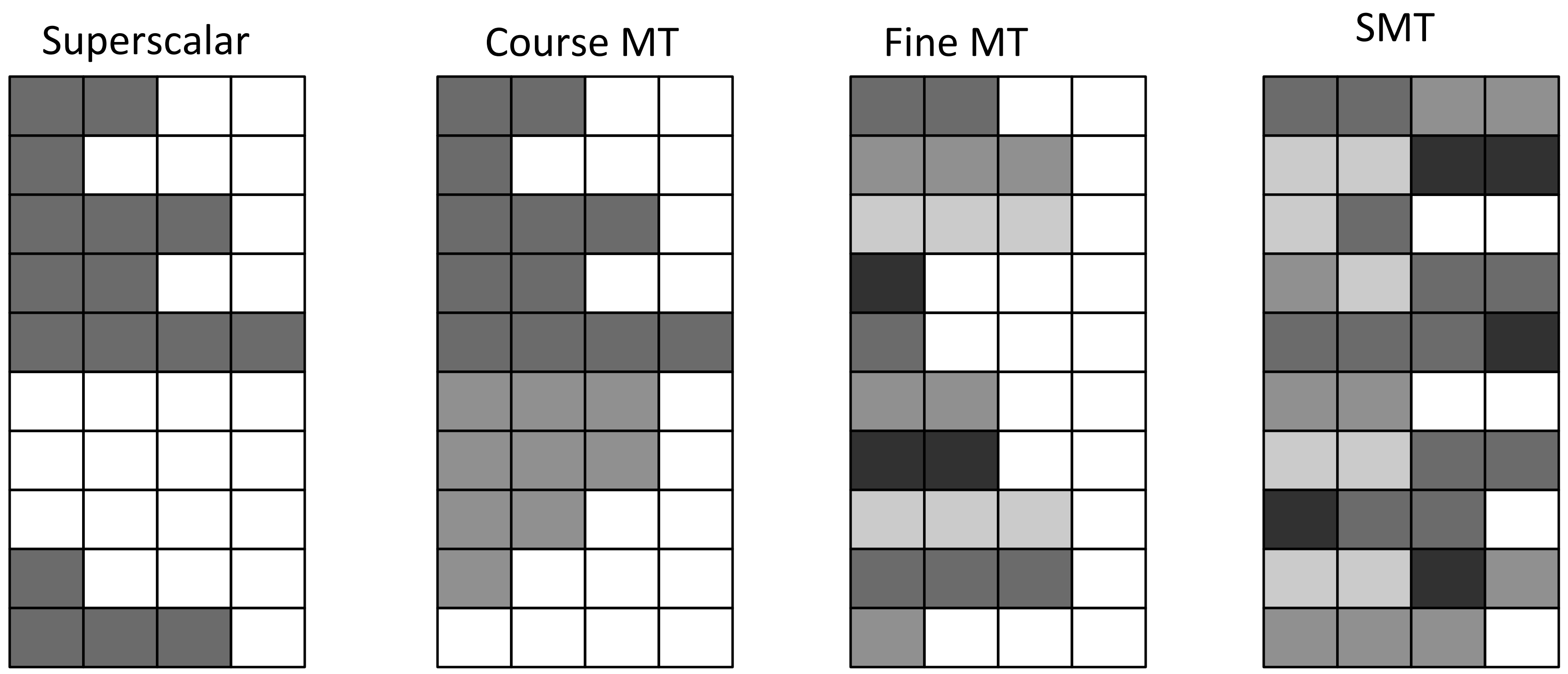
- The chips are down for Moore’s law
  - http://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338
- Special Report: 50 Years of Moore's Law
  - http://spectrum.ieee.org/static/special-report-50-years-of-moores-law
- Moore’s law really is dead this time
  - http://arstechnica.com/information-technology/2016/02/moores-law-really-is-dead-this-time/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

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 (SMT)

- Convert Thread-level parallelism to instruction-level parallelism



Simultaneous multi-threading (II)

- 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!)

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?
```