CMPE 200
Computer Architecture & Design

Lecture 5.
Parallel Processing (1)

Haonan Wang

SJSU SAN JOSÉ STATE UNIVERSITY

Processor Performance Analysis

- **Performance** $= \dfrac{1}{CPU\ Time} = \dfrac{1}{\#\ Instructions \times \underline{CPI \times Cycle\ Time}}$

  Decided by Microarchitecture design

- **CPI = 1 is ideal**
  - But, practically not achievable due to hazards in pipelined architecture

- **Cycle Time is determined by**
  - The longest instruction execution time in single-cycle CPU
  - The longest stage execution time in pipelined CPU
    - If execution times of stages are well-balanced, the speedup of N-stage pipeline is N time

Can We Do Better?

- **Latency vs. throughput**

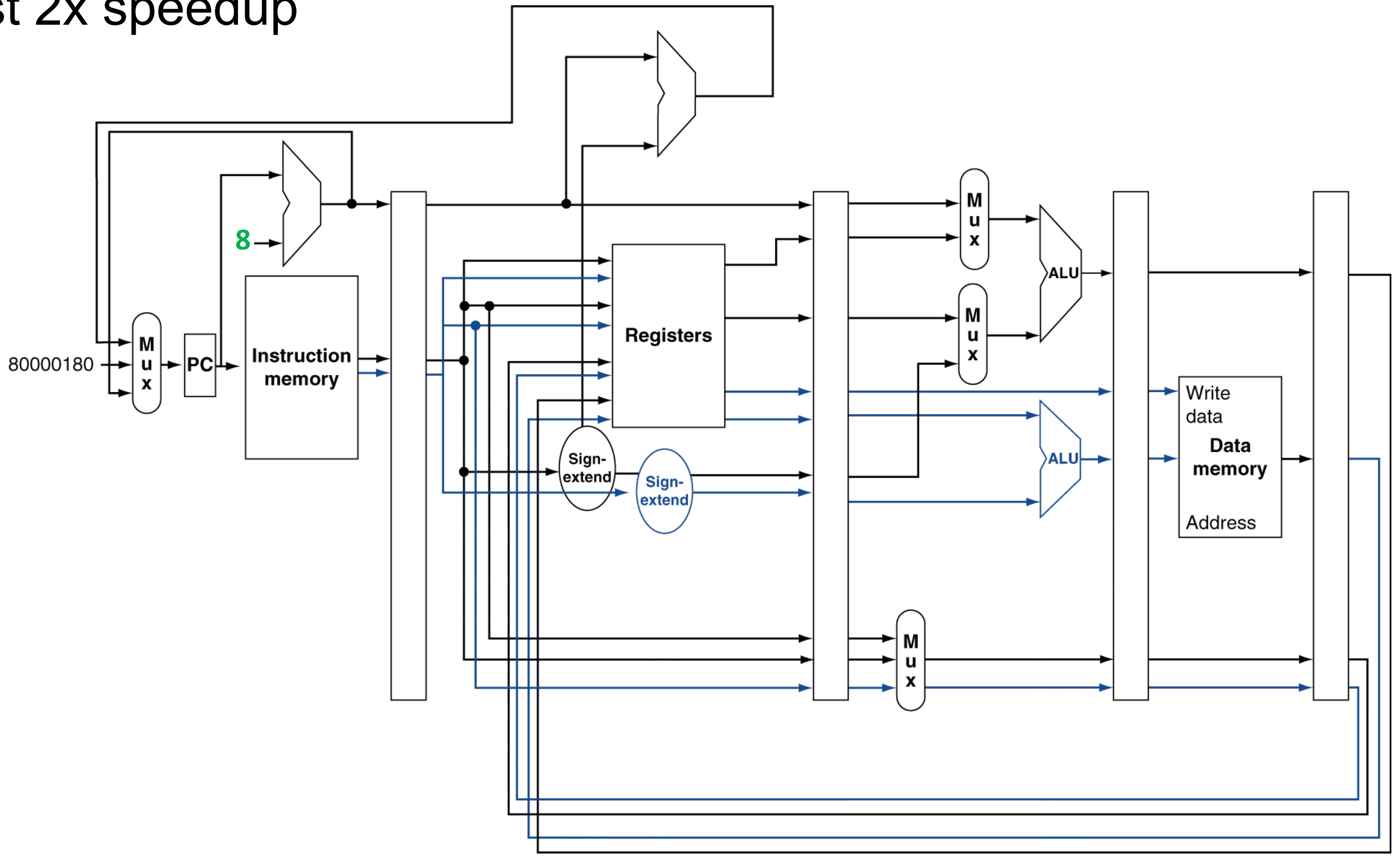
- **Levels of Parallelism:**
  - Instruction-level Parallelism (ILP)
    - Executing independent instructions (in one thread) in parallel
  - Data-level Parallelism (DLP)
    - Executing the same instruction on different data subsets
  - Thread-level Parallelism (or Task-level Parallelism, TLP)
    - Executing independent computing tasks in parallel (on same or different data)

Instruction Level Parallelism (ILP)

- **Basic idea: Execute several instructions in parallel**
- **Pipelining is one example**
  - But it can only push through at most 1 inst/cycle
- **How about multiple instruction/cycle?**
  - More complicated and more transistors/logic (e.g., superscalar)
- **Simple ILP recipe**
  - Read and decode a few instructions each cycle
  - If instructions are independent, do them at the same time
  - If not, do them one at a time

Example-1: MIPS with Dual Issue

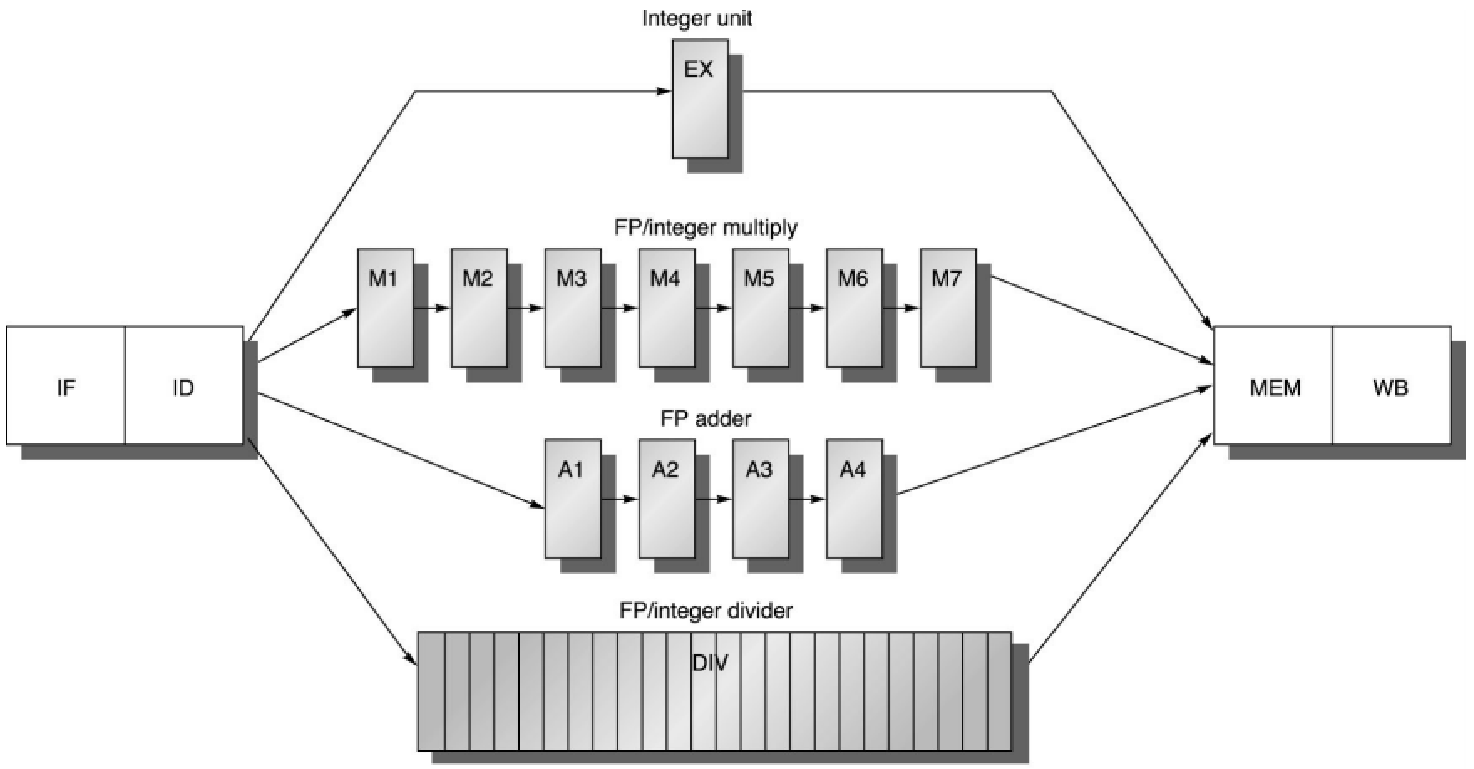
- **Issue an arithmetic and a memory instruction concurrently**
  - At most 2x speedup



SJSU SAN JOSÉ STATE UNIVERSITY

Example-2: Floating-point Pipeline

- **MIPS has separate pipeline for floating-point operations**



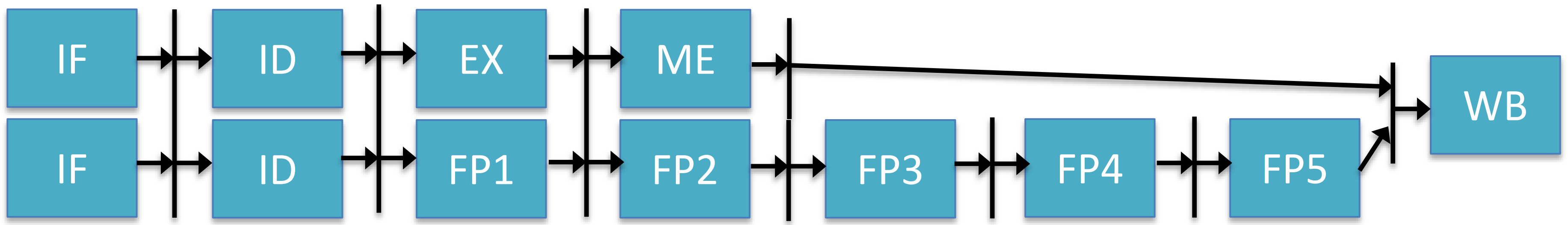
SJSU SAN JOSÉ STATE UNIVERSITY

Issue Multiple Instructions

- **Static multiple issue**
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards

- **Dynamic multiple issue**
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

2-way Superscalar Example



Forwarding paths:
MEM → EX & FP1
WB → EX & FP1

Code

```
ld.s    $f0, 0($t1)
ld.s    $f1, 0($t2)
subi    $t3, $t3, 1
add.s   $f2, $f1, $f0
addi    $t1, $t1, 4
```

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11
ld.s $f0, 0($r1)	IF	ID	EXE	MEM	WB						
ld.s $f1, 0($t2)		IF	ID	EXE	MEM	WB					
subi $t3, $t3, 1			IF	ID	EXE	MEM	WB				
add.s $f2, $f1, $f0			IF	ID	ID	FP1	FP2	FP3	FP4	FP5	WB
addi $t1, $t1, 4				IF	ID	EXE	MEM	WB			

ld.s	: load single precision data to fp reg
st.s	: store single precision data from fp reg
add.s	: add two single precision fp data

Enhance Compiler Scheduling Performance

- **Loop Unrolling: transforms an M-iteration loop to a M/N-iteration loop**
  - "Loop has been unrolled N times"
  - Effectively decrease the number of branches from M to M/N

```
for(i=0;i<100;i++)
  a[i]*=2;
```

$\rightarrow$

```
for(i=0;i<100;i+=4){
  a[i]*=2;
  a[i+1]*=2;
  a[i+2]*=2;
  a[i+3]*=2;
}
```

  - If the loop is short (i.e. 4 iterations in this code), it can even be eliminated

More Aggressive Method?

- **Static pipelines can only exploit ILP exposed by the compiler**
  - Instructions are stalled in ID stage until they are hazard free
  - Must search for the ILP across branches

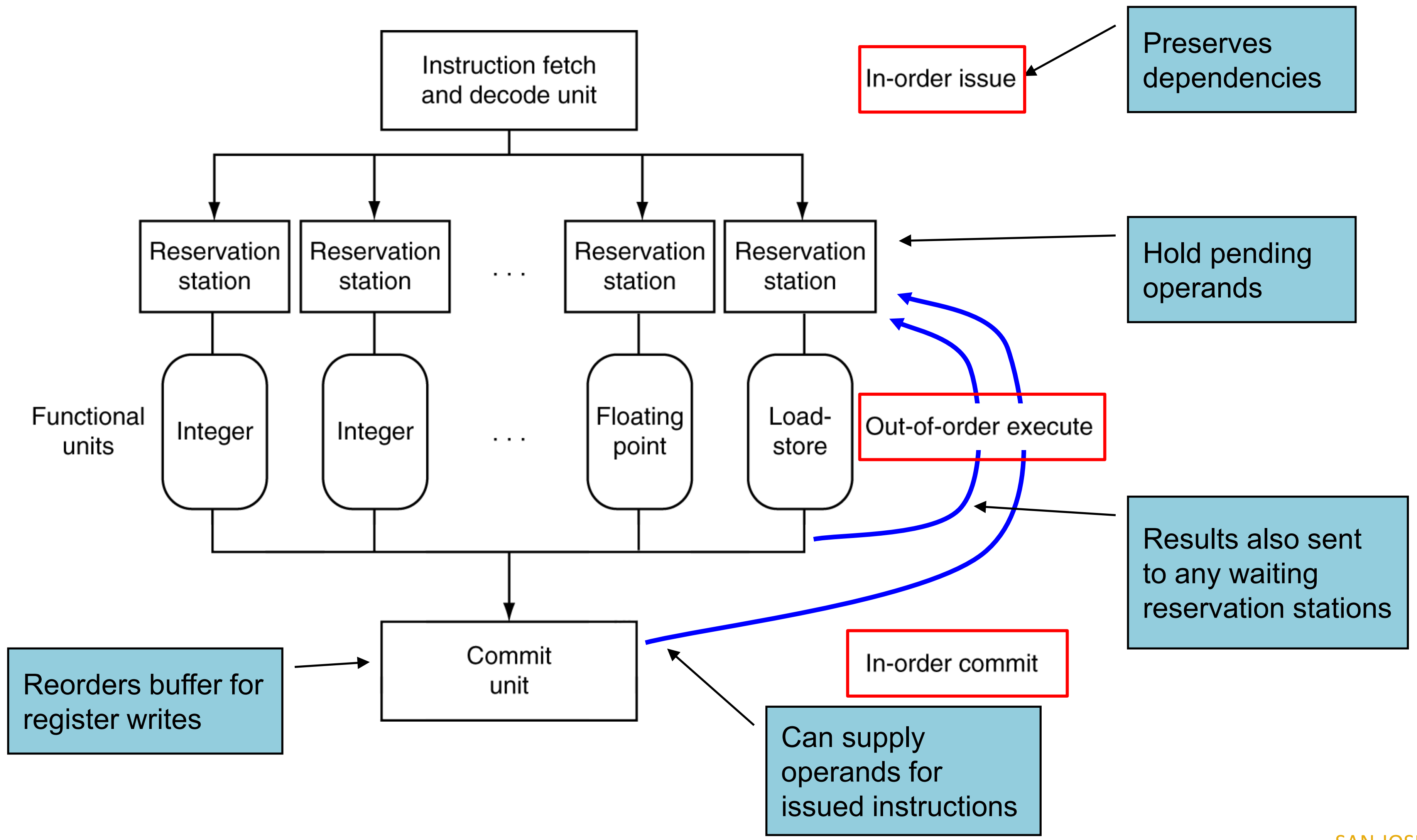
- **Dynamic pipeline scheduling**
  - Allow out-of-order (OOO) execution to avoid stalls
  - Keeps track of dependency (commit result to registers in order)
  - Decoded instructions wait in queues for operands (stall only when queues are full)

- **Example**

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s2, $s4, $t3
slti  $t0, $s2, 20
```

Why should `sub` wait for `lw` and `addu`?

Dynamically Scheduled CPU



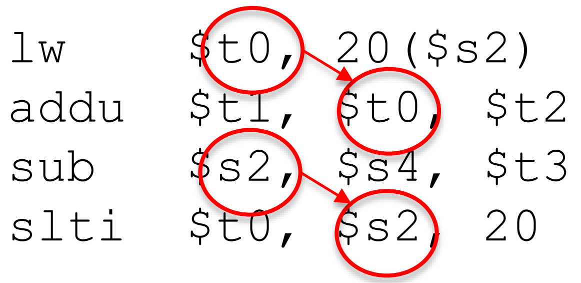
Dynamic Pipeline Scheduling

- **Challenges**
  - All data hazards (memory and registers) must be resolved by hardware

- **Data Dependencies**
  - RAW (Read After Write) : True dependency
    - Dynamic pipeline preserves only this dependency
  - WAR (Write After Read) & WAW (Write After Write) : False dependencies
    - Dynamic pipeline removes false dependencies by using register renaming

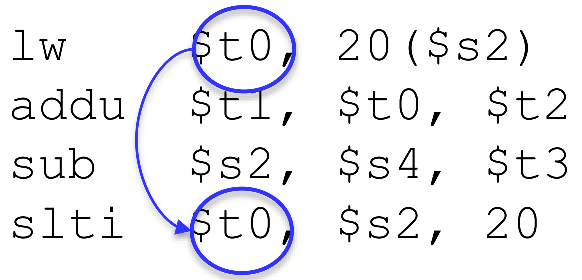
Dependencies Among Instructions

RAW



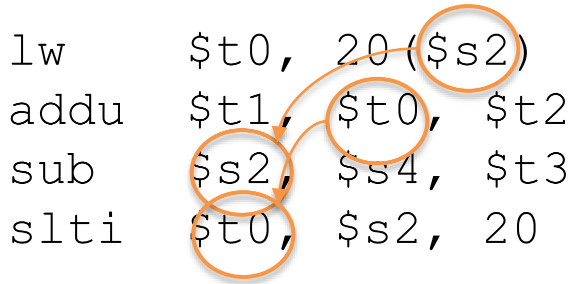
```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s2, $s4, $t3
slti    $t0, $s2, 20
```

WAW



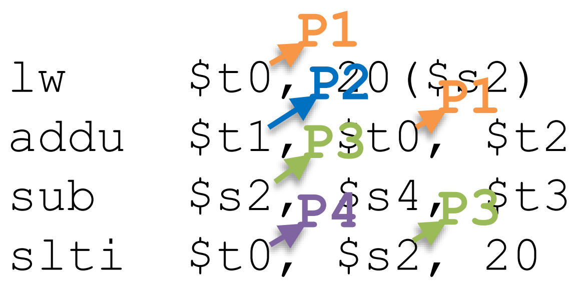
```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s2, $s4, $t3
slti    $t0, $s2, 20
```

WAR



```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s2, $s4, $t3
slti    $t0, $s2, 20
```

Preserve only RAW and remove false dependencies by renaming destination registers of each instruction



```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s2, $s4, $t3
slti    $t0, $s2, 20
```

```
lw      P1, 20($s2)
addu    P2, P1, $t2
sub     P3, $s4, $t3
slti    P4, P3, 20
```

Now can `sub` execute before `lw` and `addu`?

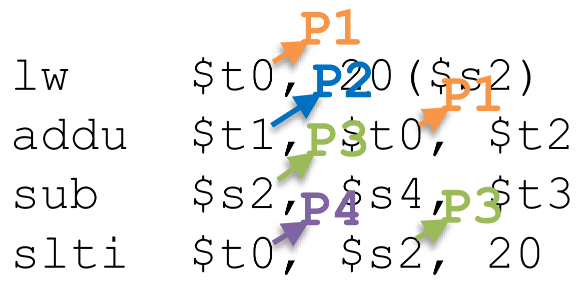
Dependencies Among Instructions

Mapping table status

Logical	initial	lw	addu	sub	slti
l1	t0	P1	P1	P1	P4
l2	t1	t1	P2	P2	P2
l3	t2	t2	t2	t2	t2
l4	t3	t3	t3	t3	t3
l5	s2	s2	s2	P3	P3
l6	s4	s4	s4	s4	s4

Free list status

initial	lw	addu	sub	slti
P1	P2	P3	P4	P5
P2	P3	P4	P5	…
P3	P4	P5	…	
P4	P5	…		
P5	…			

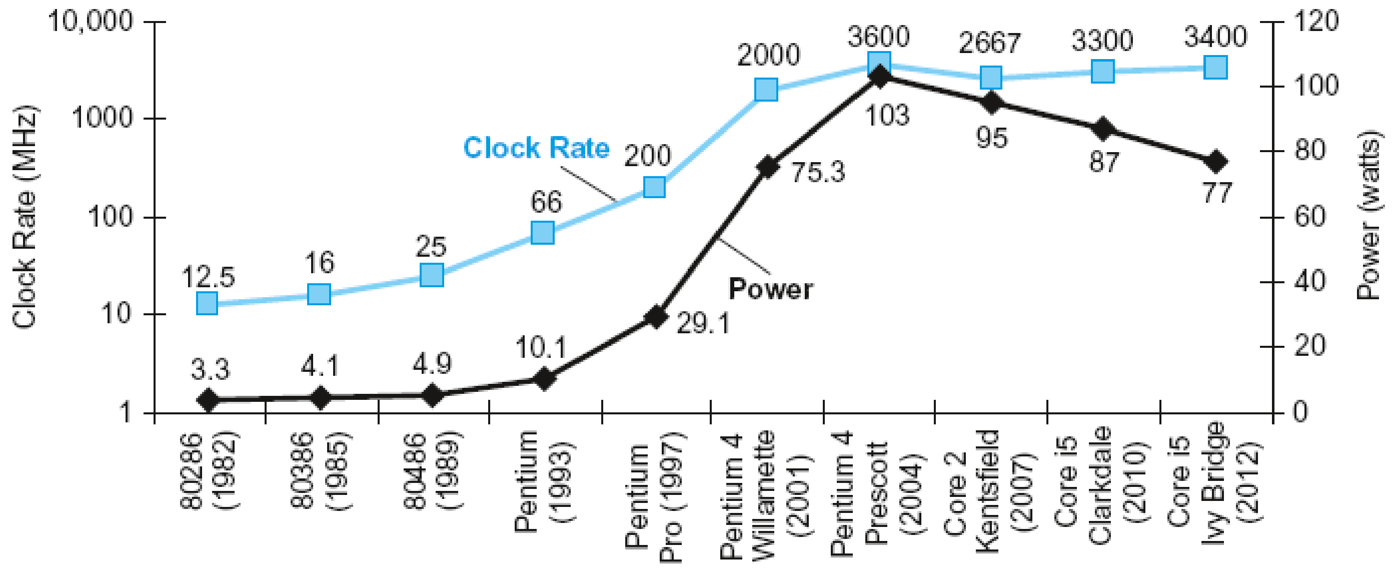


```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s2, $s4, $t3
slti    $t0, $s2, 20
```

```
lw      P1, 20($s2)
addu    P2, P1, $t2
sub     P3, $s4, $t3
slti    P4, P3, 20
```

Recall Power Wall

- With super-pipelining, super-scalar, and dynamic pipelining, CPU performance was easily increased alongside higher clock frequency until early 2000s



Power = Capacitive load x Voltage^2 x Frequency

Solution: Add more cores with the same frequency instead (DLP & TLP)

Concurrency vs. Parallelism

- **Programs: sequential or concurrent**
  - Sequential: only one activity during a period of time
  - Concurrent: multiple independent or cooperating activities during a period, but not necessarily simultaneous (e.g., multithreading on 1 core)
- **Computers: serial or parallel**
  - Parallel: multiple instructions or multiple operations per cycle (ILP, DLP, TLP)
- **Key challenge: Scalability of programs**
  - Issues: scheduling, load balancing, synchronization, communication overhead

Concurrency and Parallelism Examples

- **Matrix addition**
  - matrix A, B, C; // dimensions m × n
  - C = A + B;
  - for (i = 0; i < m; i++) for (j = 0; j < n; j++) C[i][j] = A[i][j] + B[i][j];

- **Database search**
  - find an item with a given property by examining all items

- **Web search**
  - Google's MapReduce algorithm

Conclusion Time

What is the key idea behind register renaming?

Prevent overwriting

What are the three levels of parallelism?

ILP, DLP, TLP

SAN JOSÉ STATE UNIVERSITY *powering* SILICON VALLEY