Directions for Improving Performance

General Outlook

Fundamental performance parameters

$$T = CPI \times IC \times \tau$$

$$CPI = CPI^{ideal} + CPI^{stall}$$

Technological limitations

IC grows with software complexity

τ reaching physical limit $\approx 10^{-10}$ seconds $= \left(10 \text{ GHz}\right)^{-1}$

$CPI^{ideal} = 1$ for integer pipeline

$$CPI^{stall} = CPI^{stall}_{data\ dependency} + CPI^{stall}_{cache\ miss} + CPI^{stall}_{branch\ penalty}$$

Areas for possible improvement

Instruction and thread level parallelism to achieve $CPI^{ideal} < 1$

Reducing instruction dependency stalls to lower $CPI^{stall}_{data\ dependency}$

Reducing cache latency to lower $CPI^{stall}_{cache\ miss}$

Reducing branch stalls to lower $CPI^{stall}_{branch\ penalty}$

Processing Parallelism

Instruction Level Parallelism (ILP)

Parallel execution of single program thread by **pipelining**

Superscalar processing

Multiple independent Execution Units (EU) on single CPU

EU = { ALU / FPU / Load Access / Store Access }

Instructions per cycle ≤ ALUs

Thread Level Parallelism (TLP)

Parallel execution of single program by **multithreading**

Concurrency

Multiple threads defined in software to run independently

Interleaving

Threads execute by taking alternate turns on the same CPU core

Multiprocessing

Multiple independent cores running independent threads at same time

Processor cores = chips × (cores / chip)

Flynn Taxonomy

	Instruction	
Data	Single Instruction Single Data SISD	Multiple Instruction Single Data MISD
	Single Instruction Multiple Data SIMD	Multiple Instruction Multiple Data MIMD

SISD

Standard single execution stream machine

SIMD

Single operation on multiple operands

Vector processor or processor array

MISD

Multiple operations on single operand

Few products — IBM Watson IA applies multiple algorithms to same data

MIMD

Multiple operations on multiple operands

Multiprocessor / multithreaded / cluster computer

Ref: M.J. Flynn, "Very High-Speed Computers", Proceedings of the IEEE, Dec. 1966.

Single Instruction Multiple Data (SIMD)

Single Instruction Single Data

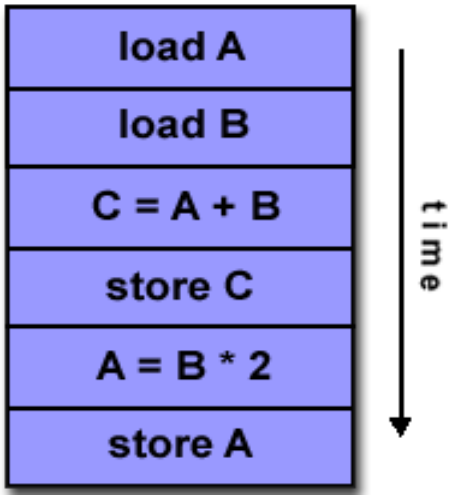
Single execution stream

Single Instruction

One instruction stream

Single Data

One input data stream



Ref: https://computing.llnl.gov/tutorials/parallel_comp/

Single Instruction Multiple Data

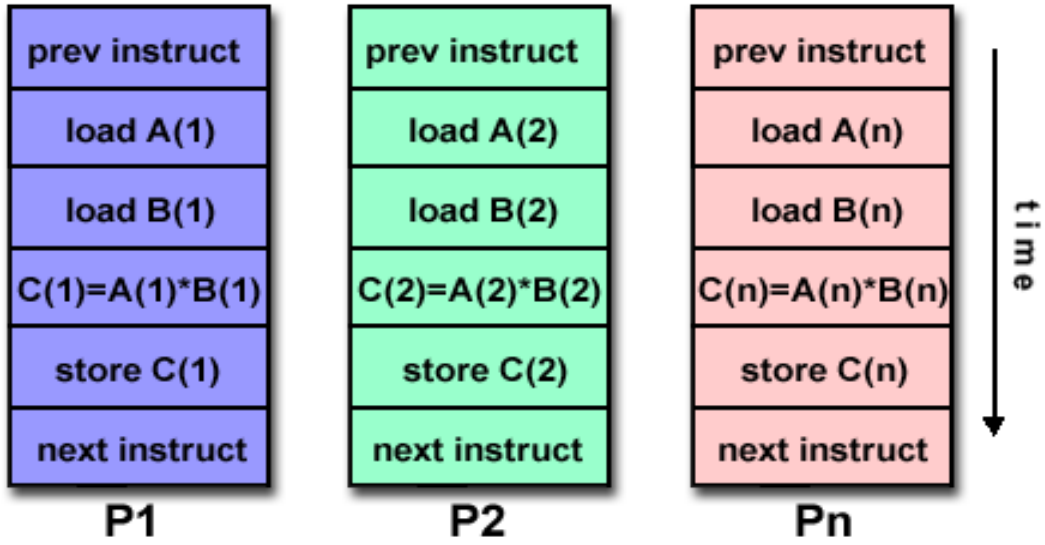
Multiple execution streams

Single Instruction

One instruction stream

Multiple Data

One instruction acts on multiple data



Multiple Instruction Single Data (MISD)

Single Instruction Single Data

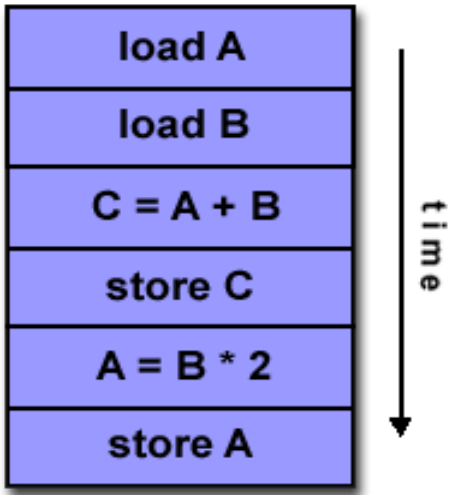
Single execution stream

Single Instruction

One instruction stream

Single Data

One input data stream



Multiple Instruction Single Data

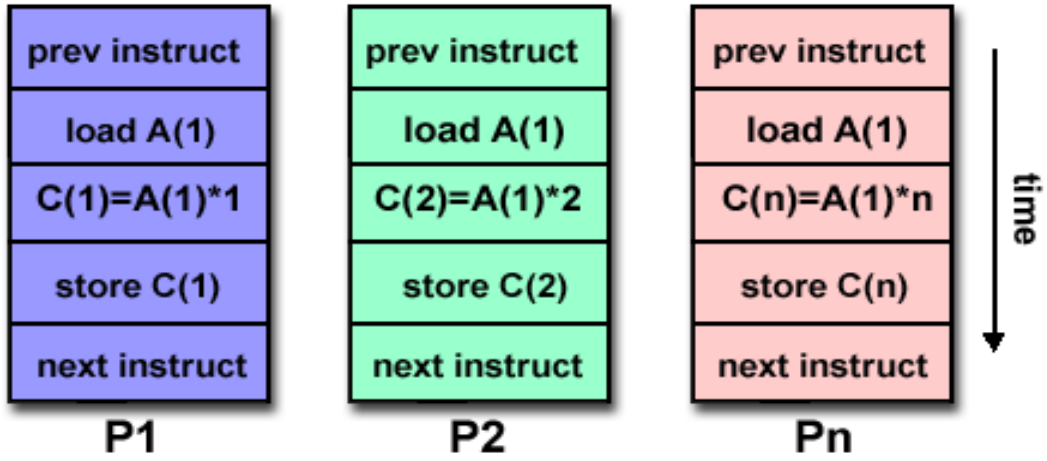
Multiple execution streams

Multiple Instruction

Multiple instruction streams act on same data

Single Data

One input data stream



Multiple Instruction Multiple Data (MIMD)

Single Instruction Single Data

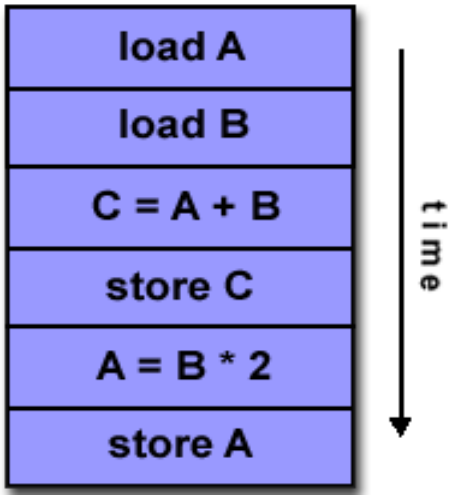
Single execution stream

Single Instruction

One instruction stream

Single Data

One input data stream



Multiple Instruction Multiple Data

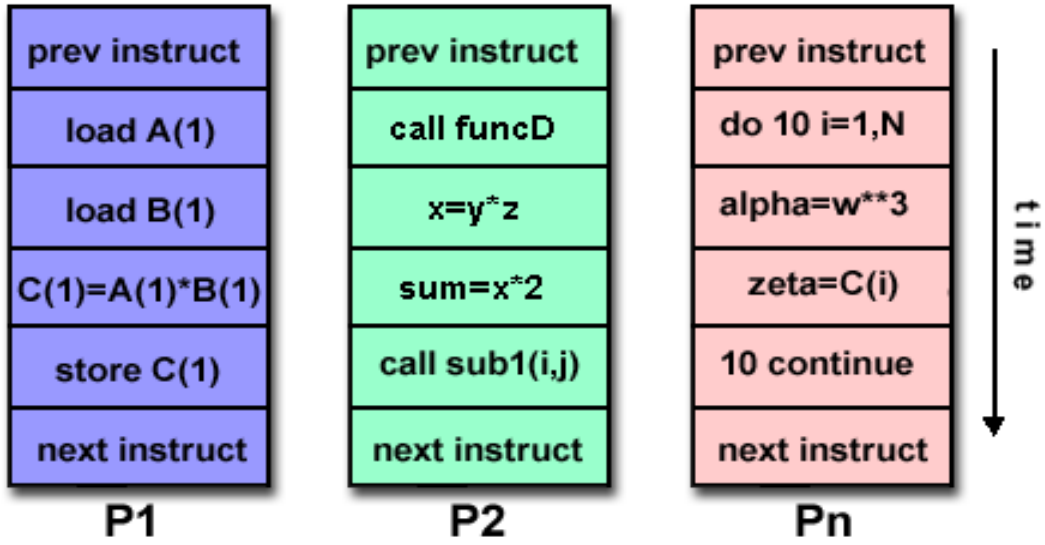
Multiple execution streams

Multiple Instruction

Multiple instruction streams run independently

Multiple Data

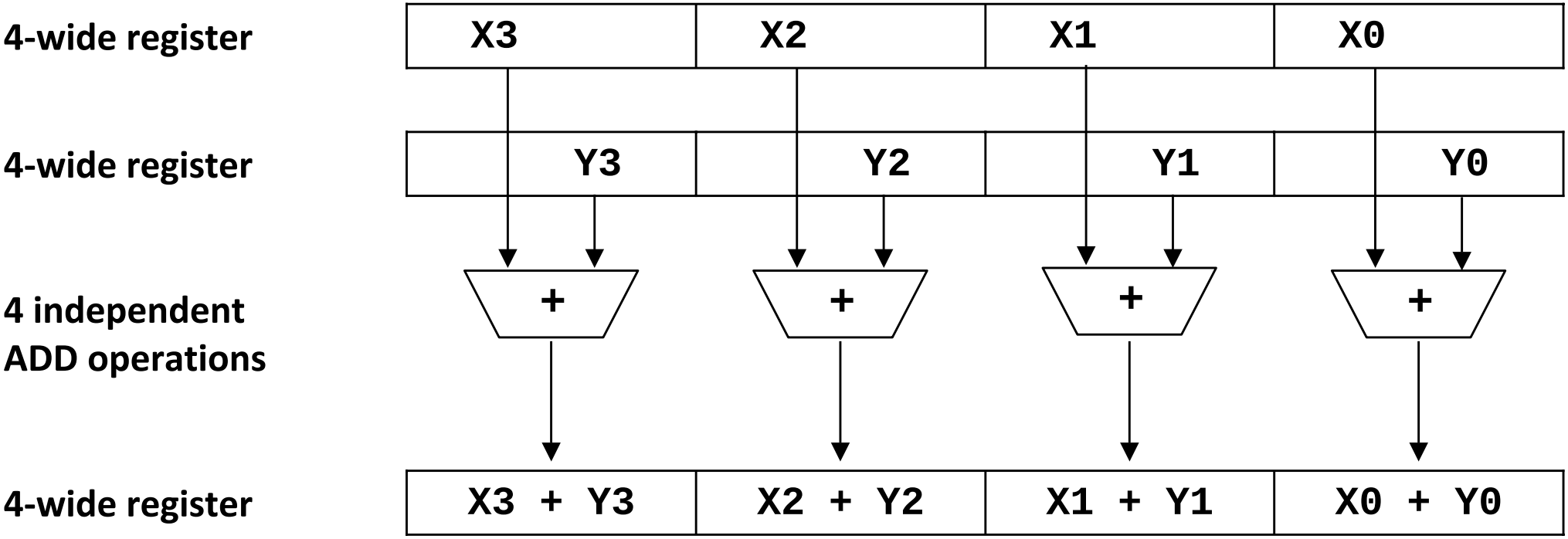
Each instruction stream acts on independent data



SIMD — Vector ALU

Example

`p_ADD = parallel add`

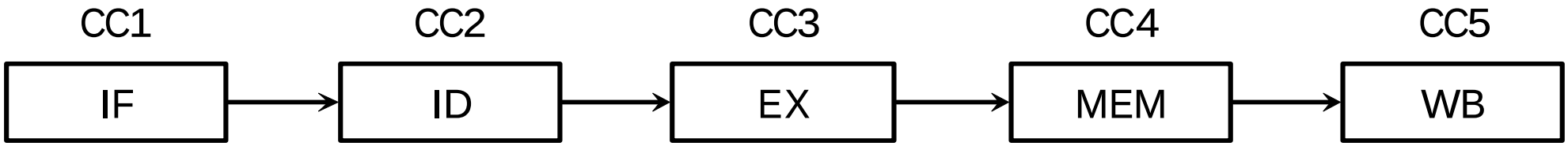


Typical applications for vector operations

Data compression / decompression

Graphics / audio processing

DLX Vector Pipeline Example



`p_LW P1, 400(R1)` Load 4 memory words (16 bytes) to register P1

`p_LW P2, 800(R1)` Load 4 memory words (16 bytes) to register P2

`p_ADD P3, P1, P2` Perform 4 word additions in parallel

	1	2	3	4	5	6	7	8
`p_LW P1, 400(R1)`	IF	ID	EX	MEM	WB			
`p_LW P2, 800(R1)`		IF	ID	EX	MEM	WB		
`p_ADD P3, P1, P2`			IF	ID	ID	EX	MEM	WB

Examples of Vector Processing

Intel MMX / SSE / AVX

Vector integer and float operations

	SSE	AVX
Register width	128 bits = 16 bytes	256 bits = 32 bytes
Integer	4	8
Single-precision FP	4	8
Double-precision FP	2	4

Power AltiVec vector processor

Similar to MMX integer operations

128-bit registers

Compiler Support

Vector instructions in processor instruction set

Reasonable compiler supports vectorization

Vectorization Example on Intel SSE

```
int i;
float a[100], red;
…
red = 0;
for (i = 0; i < 100; i++) {
     red += a[i];
}
```

Loop runs **100** times

Operates on one array element per loop iteration

```
int i;
float a[100], red;
…
red = 0;
ASM p_XOR xmm0,xmm0              /* zero accumulator */
for (i = 0; i < 25; i++) {
     ASM p_ADD xmm0,a[4*i]       /* four 32-bit additions */
}
ASM h_ADD xmm0                   /* add h_add X ⇒
                                    X3 ← X3 + X2 + X1 + X0 */
ASM p_MOV red, xmm0              /* red ← FP sum */
```

Loop runs **25** times

Operates on 4 array elements per loop iteration

Optimizing Data Structures

Data structure for 3D object

200 vertices in (x, y, z) coordinates

24-bit color (red, blue green) at each vertex

Typical data structure — array of structures

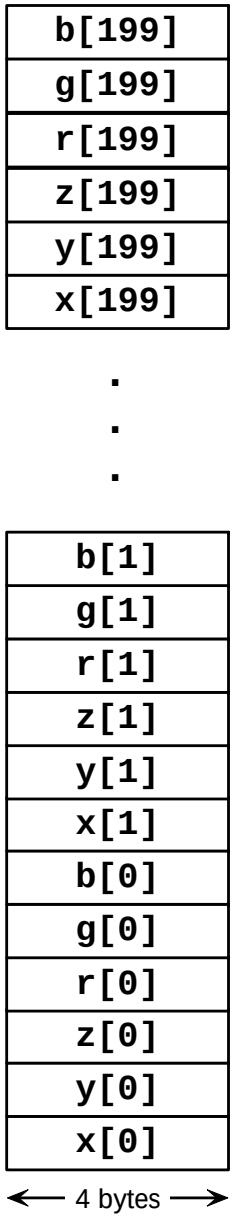
Struct = 200 consecutive groups of six 32-bit words

```
struct { float x, y, z, r, g, b
    } AoS_xyz_rgb[200];
```

SSE requires sequence

- 4 parallel **x** values
- 4 parallel **y** values
- 4 parallel **z** values
- 4 parallel **r** values
- 4 parallel **g** values
- 4 parallel **b** values

SSE cannot operate on AoS data structure



Hybrid Data Structures

Hybrid Structure

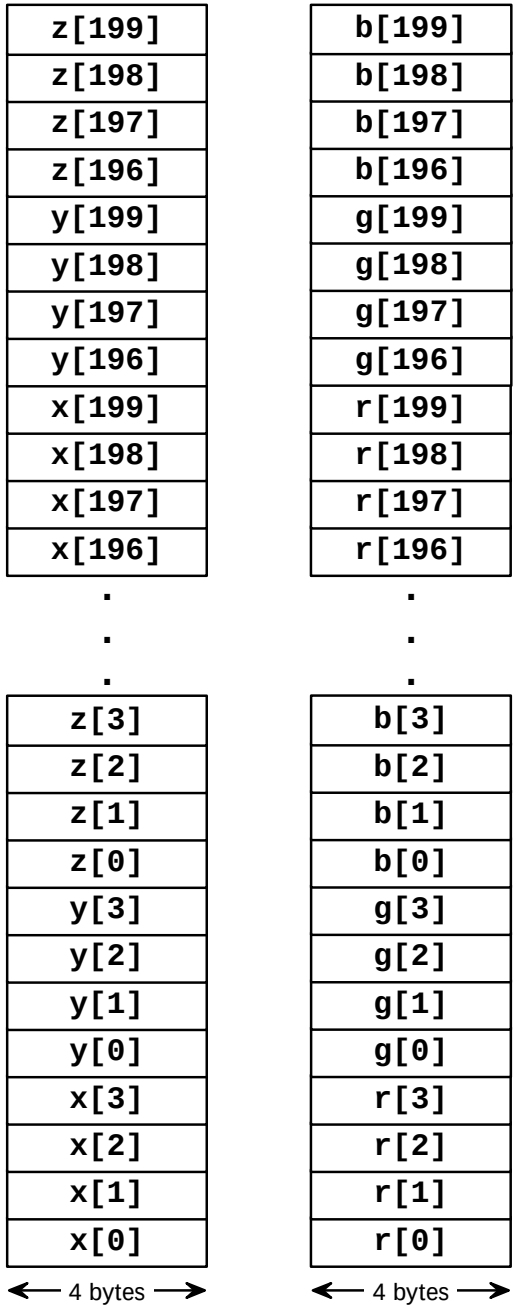
```
struct { float x[4], y[4], z[4] ;
} H_xyz[50] ;
struct {float r[4], g[4], b[4] ;
} H_rgb[50] ;
```

Vertex data and colors are loaded separately

SSE thread-1 loads 16 bytes in parallel

- Operates on vector of four 32-bit FP values
- Parallel operation on 4 **x** values
- Parallel operation on 4 **y** values
- Parallel operation on 4 **z** values

SSE thread-2 operates on rgb



Graphics Processing Unit — Hardwired GPU

Application of chip real estate (≈ 3 billion transistors)

CPU — efficient processing of sequential code

GPU — large number of execution cores

CPU offloads graphics operations to GPU as auxiliary processor

Hardwired graphics pipeline

Stage	Input	Output	
Geometry	3D vertices	2D triangles	Area enclosed by lines between vertices
Rasterization	Triangles	Fragments	Screen coordinates covered by triangle
Fragment	Fragments	Colored Fragments	Fragment colored by interpolation of vertices or from texture memory
Composition	Textures	Image	Pixel coordinates and colors

Theoretical SIMD performance

SSE on 3.7 GHz Pentium 4 (2000)

1 pipeline × 4-wide SIMD × 3.7 GHz = 14.8 GFLOPS

Nvidia GeForce 6800 Ultra (2005)

16 pipelines × 8-wide SIMD × 520 MHz =63 GFLOPS

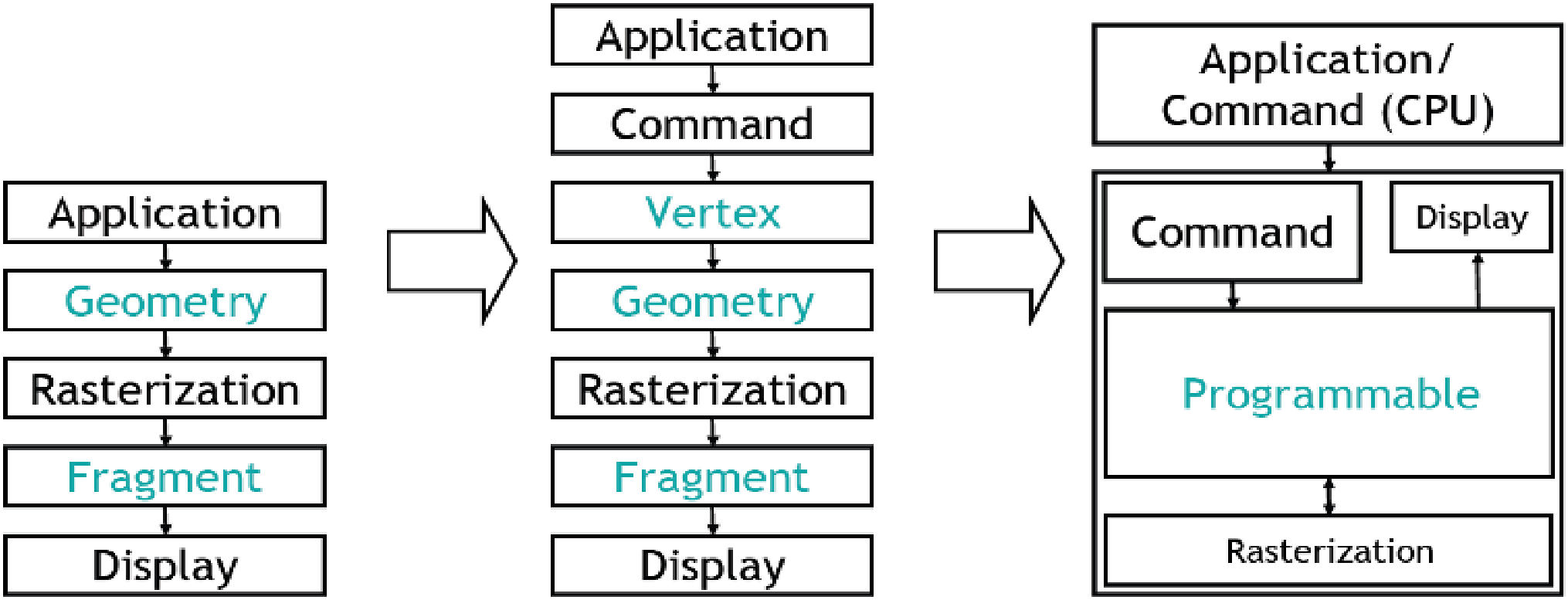
Graphics Processing Unit — Programmable Pipeline

GPU evolution

Hardwired pipeline stage → programmable pipeline stage

Hardwired pipeline organization → **SHADER**

- Programmable processor core
- Optimized to support graphics operations



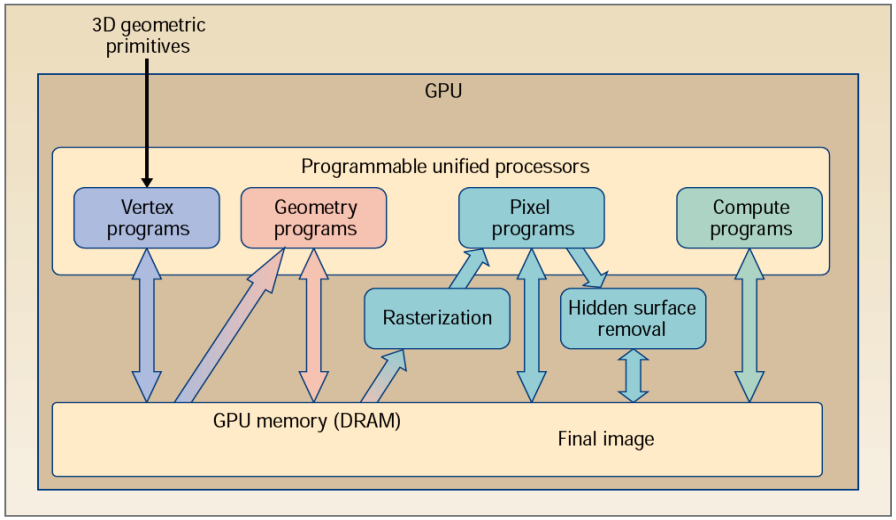
John Owens, *GPU Architecture Overview*, UC Davis, SIGGRAPH 2007

Graphics Processing Unit — GPGPU

General Purpose GPU (GPGPU)

Heterogeneous multi-core processor

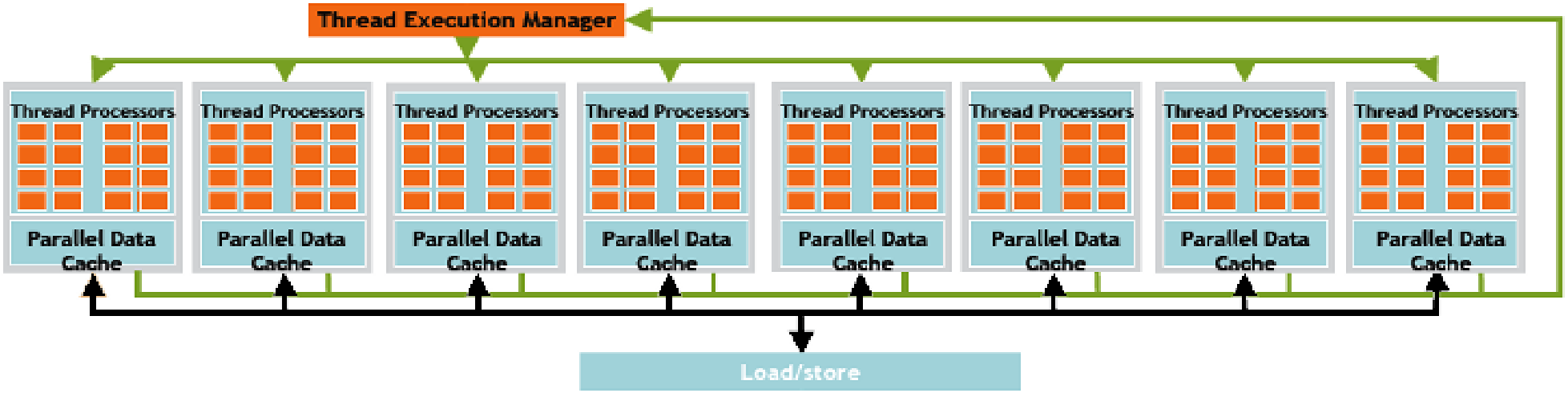
Grid of shaders



Nvidia shader

Nvidia GeForce 8800 (2005)

8 thread processors × 16 shaders/processor = 128 shaders

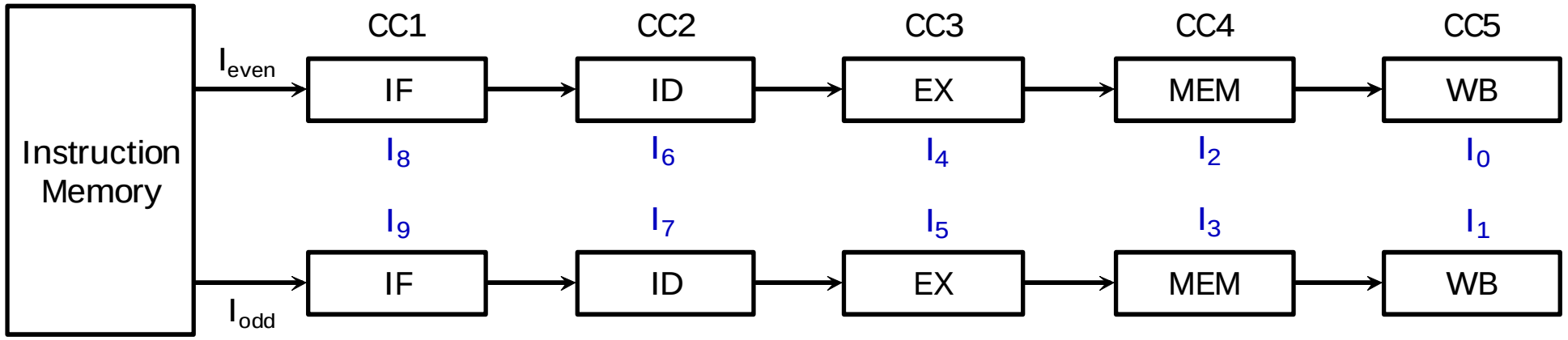


GeForce RTX 3090 Ti (2022)

10752 shaders

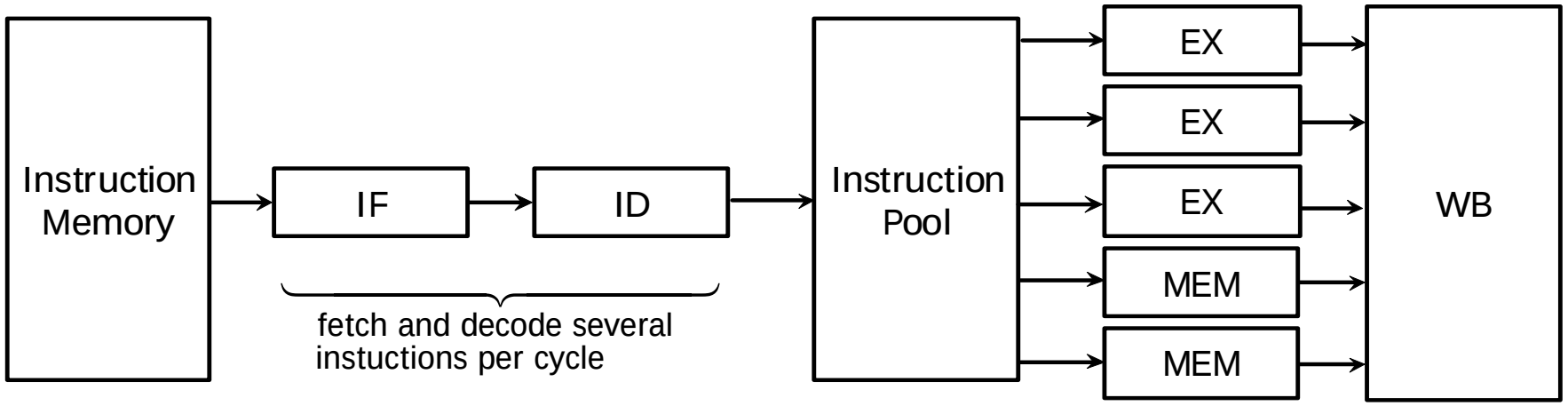
Two Superscalar Pipeline Models for DLX

Replicate entire pipeline



Replicate EX and MEM execution units (EUs)

Instruction Pool holds decoded instructions



Intel 486 — Single Pipeline Processor

Instruction Fetch (IF)

Stage-1 Decode (D1)

Identify operation

Read register sources

Calculate register part of effective address for memory source

Stage-2 Decode (D2)

Calculate physical data address from logical address

Generate execution microcode

Execution (EX)

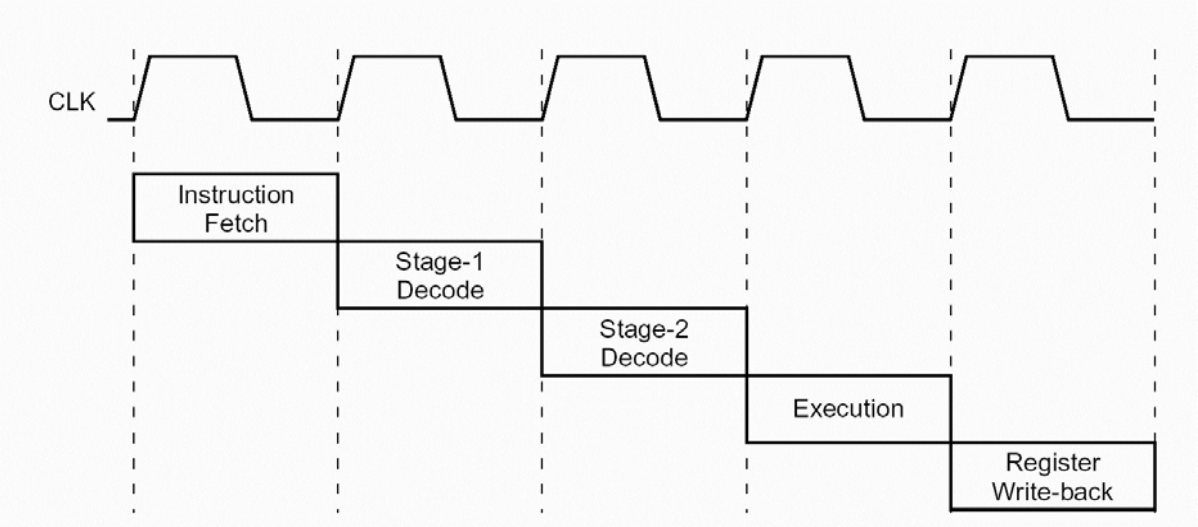
Integer ALU operations

FPU operations

Data memory accesses

Register Writeback (WB)

Update destination registers



"Embedded Intel486™ Processor Hardware Reference Manual" (July 1997)

Intel Pentium — Dual Pipeline Processor

Ideal pipeline progress in Intel 486

	0	1	2	3	4	5	6	7
I_0	PF	D1	D2	EX	WB			
I_1		PF	D1	D2	EX	WB		
I_2			PF	D1	D2	EX	WB	
I_3				PF	D1	D2	EX	WB
I_4					PF	D1	D2	EX
I_5						PF	D1	D2
I_6							PF	D1
I_7								PF

Ideal pipeline progress in Intel Pentium

Pipeline stages as in 486

	0	1	2	3	4	5	6	7
I_0	PF	D1	D2	EX	WB			
I_1	PF	D1	D2	EX	WB			
I_2		PF	D1	D2	EX	WB		
I_3		PF	D1	D2	EX	WB		
I_4			PF	D1	D2	EX	WB	
I_5			PF	D1	D2	EX	WB	
I_6				PF	D1	D2	EX	WB
I_7				PF	D1	D2	EX	WB

$$T = CPI \times IC \times \tau$$

$$CPI = \frac{\text{clock cycles}}{IC} \Rightarrow CPI^{ideal} = \frac{N \text{ clock cycles}}{2 \times N} = 0.5 = \frac{CPI^{\text{one-pipeline ideal}}}{\text{number of pipelines}}$$

Problems with Pipeline Replication

Dependent instructions cannot be paired

I_N and I_{N+1} cannot be paired ⇒ run in usual pipeline sequence

Paired instructions stall together

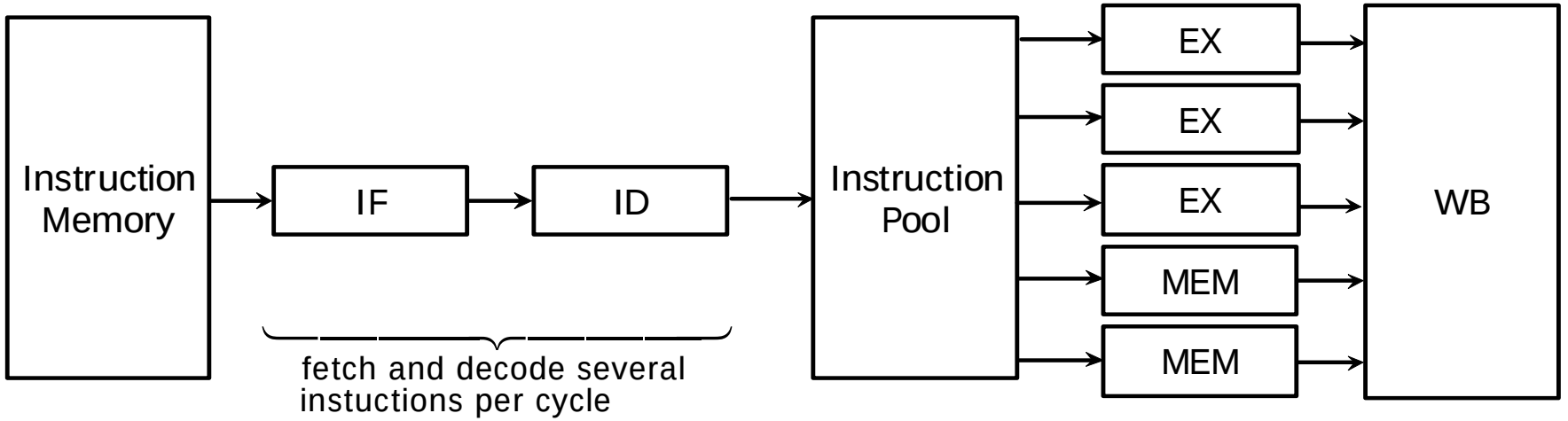
I_S stalls ⇒ I_{S+1} stalls

Example

```
I0   MOV EAX, 100          ; I0 and I1 cannot be paired
I1   ADD ESP, EAX
I2   PUSH EAX              ; stack pointer ESP not forwarded
                           ; I2 waits for I1 write-back
I3   ADD ECX, EAX          ; I2 stalls ⇒ I3 stalls
```

	0	1	2	3	4	5	6	7	8
I_0	IF	D1	D2	EX	WB				
I_1	IF	D1	D1	D2	EX	WB			
I_2		IF	IF	D1	D1	D1	D2	EX	WB
I_3		IF	IF	D1	D1	D1	D2	EX	WB

Instruction Pool Example



			CC	Utilization

I1
I2
I3
	ADDI R1, R0, #10
	ADDI R2, R0, #87
	LW R6, 100(R4)
		No dependencies
		Run on 2 EX + 1 MEM
			1	3

I4	ADD R4, R2, R6	Dependency — runs in EX	2	1
I5	ADD R7, R4, R6	Dependency — runs in EX	3	1
I6	SW 100(R4), R7	Dependency — runs in MEM	4	1
I7	SW 200(R0), R8	1 store per CC — runs in MEM	5	1

I7 has no dependency — could **RESCHEDULE** to CC1, CC2, CC3

Instruction Rescheduling Strategies

Instruction dependencies

RAW hazard — instruction $\mathbf{I_{k+j}}$ reads operand before $\mathbf{I_k}$ writes it

WAW hazard — instruction $\mathbf{I_{k+j}}$ writes operand before $\mathbf{I_k}$ writes it

WAR hazard — instruction $\mathbf{I_{k+j}}$ writes operand before $\mathbf{I_k}$ reads it

Instruction dependency stall

Delay $\mathbf{I_{k+j}}$ until $\mathbf{I_k}$ completes

Register renaming

Allocate separate register sets to independent instruction blocks

Eliminates dependencies between independent blocks using same registers

Instruction rescheduling

Move independent instructions between $\mathbf{I_k}$ and $\mathbf{I_{k+j}}$

Increase instruction "distance" in pipeline to resolve dependence

Compiler optimization

Compiler reschedules object code (machine code listing) in second pass

Dynamic rescheduling (out-of-order execution)

Hardware executes instructions in most efficient order

Chooses instructions with no unresolved dependencies for execution

Efficient Register Reuse

Instruction block N

Allocate **R1**

Overwrite for use as destination operand

Instructions access **R1**

Access as source and destination operand

De-allocate (**RETIRE**) **R1**

After final source access in instruction block

Instruction block N+1

Re-allocate **R1**

Register Reuse	
`LW R1, 0(R2)`	`; allocate R1`
`ADDI R1, R1, #1`	`; R1 is source and destination`
`SW 0(R2), R1`	`; R1 is source — de-allocate R1`
`ADD R1, R2, R3`	`; re-allocate of R1`

Register Reuse versus Register Renaming

Register renaming

- Instruction block N
  - Allocate **R1**
  - Instructions access **R1**
  - De-allocate (**RETIRE**) **R1**
- Instruction block N+1
  - Allocate new register instead of reuse

Eliminates fictitious data dependence

- Instruction blocks use independent registers
- Enables rescheduling

Example

Reuse	Renaming	Rescheduling
`LW R1, 0(R2)`	`LW R1, 0(R2)`	`LW R1, 0(R2)`
`ADDI R1, R1, #1`	`ADDI R1, R1, #1`	`ADD R10, R2, R3`
`SW 0(R2), R1`	`SW 0(R2), R1`	`ADDI R1, R1, #1`
`ADD R1, R2, R3`	`ADD R10, R2, R3`	`SW 0(R2), R1`

Instruction Rescheduling for DLX

Compiler reschedules instruction issue

- Respect real data flow dependencies
- Eliminate fictitious data dependencies by register renaming
- Enhance efficiency
  - Insert independent work between dependent instructions

High Level Code	First Pass Assembly Code	Rescheduled Code for DLX

`X = X + 123`	`LW R2,[X]`
	`ADD R2,R2,#123`
	`SW [X],R2`
		`LW R2,[X]`
		`LW R3,[Y] ; Rename R2`
		`LW R4,[Z] ; Rename R2`
		`ADD R2,R2,#123`
		`SUB R3,R3,#456`
		`SUB R4,R4,#789`
		`SW [X],R2`
		`SW [Y],R3`
		`SW [Z],R4`

`Y = Y – 456`	`LW R2,[Y] ; Reuse R2`
	`SUB R2,R2,#456`
	`SW [Y],R2`	

`Z = Z - 789`	`LW R2,[Z] ; Reuse R2`
	`SUB R2,R2,#789`
	`SW [Z],R2`	

Advantage of Rescheduled Code

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
LW R2,[X]	F	D	X	M	W											
ADD R2,R2,#123		F	D	D	X	M	W									
SW [X],R2			F	F	D	X	M	W								
LW R3,[Y]					F	D	X	M	W							
SUB R3,R3,#456						F	D	D	X	M	W					
SW [Y],R3							F	F	D	X	M	W				
LW R4,[Z]									F	D	X	M	W			
SUB R4,R4,#789										F	D	D	X	M	W	
SW [Z],R4											F	F	D	X	M	W

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
LW R2,[X]	F	D	X	M	**W**											
LW R3,[Y]		F	D	X	M	**W**										
LW R4,[Z]			F	D	X	M	**W**									
ADD R2,R2,#123				F	**D**	X	M	**W**								
SUB R3,R3,#456					F	**D**	X	M	**W**							
SUB R4,R4,#789						F	**D**	X	M	**W**						
SW [X],R2							F	**D**	X	M	W					
SW [Y],R3								F	**D**	X	M	W				
SW [Z],R4									F	**D**	X	M	W			

Scheduling Algorithm — Instruction Scoreboard

Status field assigned to instructions in program listing

NR	Not Ready	At least one source operand not available
R	Ready	All source operands available
X	Executed	Instruction executed — destination operand not available
F	Finished	Instruction executed — all destination operand(s) available

Instructions executed according to

- Status fields
  - Only instructions marked Ready can be executed
- Scheduling policy
  - Depends on hardware organization

Update scoreboard after each clock cycle

- Completed instructions marked Finished
- Instructions marked Ready as operands become available

Scoreboard Example for Renamed DLX Code

Instruction status after EX clock cycle

Instruction	1	2	3	4	5	6	7	8	9	10	11	12
LW R2,[X]	X	F	F	F	F	F	F	F	F	F	F	F
ADD R2,R2,#123	NR	R	F	F	F	F	F	F	F	F	F	F
SW [X],R2	NR	NR	R	F	F	F	F	F	F	F	F	F
LW R3,[Y]	R	R	R	R	X	F	F	F	F	F	F	F
SUB R3,R3,#456	NR	NR	NR	NR	NR	R	F	F	F	F	F	F
SW [Y],R3	NR	NR	NR	NR	NR	NR	R	F	F	F	F	F
LW R4,[Z]	R	R	R	R	R	R	R	R	X	F	F	F
SUB R4,R4,#789	NR	NR	NR	NR	NR	NR	NR	NR	NR	R	F	F
SW [Z],R4	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	R	F

Scheduling rules

Issue only ready instructions

Choose instructions in **ORIGINAL PROGRAM ORDER**

Scoreboard generates inefficient code

NR	Not Ready
R	Ready
X	Executed
F	Executed and destination operand available

Scoreboard Example for Renamed DLX Code

Instruction status after EX clock cycle

Instruction	1	2	3	4	5	6	7	8	9	Execution Order
LW R2,[X]	X	F	F	F	F	F	F	F	F	1
ADD R2,R2,#123	NR	R	R	F	F	F	F	F	F	4
SW [X],R2	NR	NR	NR	R	R	R	F	F	F	7
LW R3,[Y]	R	X	F	F	F	F	F	F	F	2
SUB R3,R3,#456	NR	NR	R	R	F	F	F	F	F	5
SW [Y],R3	NR	NR	NR	NR	R	R	R	F	F	8
LW R4,[Z]	R	R	X	F	F	F	F	F	F	3
SUB R4,R4,#789	NR	NR	NR	R	R	F	F	F	F	6
SW [Z],R4	NR	NR	NR	NR	NR	R	R	R	F	9

Scheduling rules

Issue only ready instructions

Choose instructions in **ORDER THEY BECOME READY**

Scoreboard generates

compiler-rescheduled code

NR	Not Ready
R	Ready
X	Executed
F	Executed and destination operand available

Virtual Register Architecture

Architectural register set (AR)

Defined in Instruction Set Architecture (ISA) of CPU

DLX: R0, R1, … , R31, F0, …, F31

x86: EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP, EIP

Visible to programmer

Atomic read / write at level of ISA semantics

Virtual register set (VR)

Physical hardware registers

VR0, VR1, VR2, …, VRn-1

Physical register set > architectural register set

Pentium 4: n = 128

Not visible to programmer

VRs allocated to ARs per instruction block

Instruction blocks identified by block ID

VR Allocation — Automatic Register Renaming in CPU

CPU recodes each machine instruction

- Destination AR ← new VR
- Stable source AR ← literal value stored in AR
- Variable source AR ← last VR assigned to AR

Recoded instruction

- Stored in instruction pool
- Executed in some specified order

Commit to state (RETIREMENT)

- Copy AR ← VR in original order
- De-allocate VR

Example

Machine Code	Recoded	Execution CC
`LW R1, 0(R2)`	`LW VR1, 0(reg[R2])`	1
`ADDI R1, R1, #1`	`ADDI VR2, VR1, #1`	2
`SW 0(R2), R1`	`SW 0(reg[R2]), VR2`	3
`ADD R1, R2, R3`	`ADD VR3, reg[R2], reg[R3]`	1

VRs retired in order (R1 ← VR1, R1 ← VR2, R1 ← VR3)

Bookkeeping — Register Allocation Table (RAT)

Cross-reference method

- Index
  - VR name
- Entry
  - AR allocated to VR
  - Instruction block ID
  - Status (allocated / not allocated)

AR look-up method

- Index
  - AR name
- Entry
  - Allocated VR
  - Sequence ID to record VR allocation ordering
  - Status (stable / allocated)
- Method used in Intel x86

Ref: Hinton, et. al., *The Microarchitecture of the Pentium® 4 Processor*, Intel Technology Journal Q1, 2001

Intel RAT Bookkeeping — Decoding and Retirement

Instruction decoding

- Stable source AR ← value stored in AR
- Allocated source AR ← VR with highest sequence in RAT (for AR)
- Destination AR ← new VR allocation
  - New RAT entry written
  - Available VRs found by searching RAT for unallocated VRs
  - Decoder allocates available VRs sequentially mod n
  - Maximize code distance between VR reuse
- Recoded instruction (with VRs) placed in instruction pool
  - Instructions executed from pool by reading and writing VRs

VR is retired if either:

- Instruction that overwrites allocated AR is marked finished
- VR does not appear as source for any unfinished instructions in pool
  - VR (with highest sequence number) copied to AR (commitment)
  - AR marked stable in RAT
  - Next use of AR as a source will access stable AR value

Virtual Register Example

	Instruction	AR	VR	Recoded Instruction
I1	LW R1, 0(R20)	R1	VR1	LW VR1, 0(R20)
I2	ADDI R2, R1, #1	R2	VR2	ADDI VR2, VR1, #1
I3	SW 0(R20), R2			SW 0(R20), VR2
I4	MUL R2, R3, R4	R2	VR3	MUL VR3, Regs[R3], Regs[R4]
MUL overwrites R2 ⇒ retire VR2				
I5	SUBI R3, R2, #3	R3	VR4	SUBI VR4, VR3, #3
I6	XOR R2, R3, R4	R2	VR5	XOR VR5, VR4, Regs[R4]
XOR overwrites R2 ⇒ retire VR3				
I1 and I2 complete ⇒ retire VR1 Commit Reg[R1] ← Reg[VR1] ⇒ R1 marked stable				
I7	AND R4, R1, R2	R4	VR6	AND VR6, Regs[R1], VR5

WAR and WAW Pseudo-Dependencies

WAR and WAW data dependencies

```
I1   ADD R2, R1, R3      ; WAR hazard if I1 placed after I2 or I3
I2   ADD R1, R3, R4      ; WAW hazard if I2 placed after I3
I3   ADD R1, R3, R5
```

Remove data dependencies

```
I1   ADD R2, R1, R3      ; allocate VR1 to R2
                         ; read stable values from R1, R3
I2   ADD R1, R3, R4      ; allocate VR2 to R1
                         ; read stable values from R3, R4
I3   ADD R1, R3, R5      ; allocate VR3 to R1
                         ; read stable values from R3, R5
```

Prevent hazards

1. Read source values and allocate destination VRs in program order
2. Execute I_1, I_2, I_3 in any order
3. Retire **VR2** and **VR3** in program order

Removing False Register Dependencies

Virtual registers on superscalar CPU

	Sequential code	Allocate	Rescheduled Code with Virtual Registers
CC1	LW R1, [0x4]	VR1 ← R1	VR1 (R1) ← [0x4]
CC2	MUL R1, R1, R2	VR2 ← R1	VR2 (R1) ← VR1 (R1) * regs[R2] VR4 (R1) ← [0x1234] VR5 (R2) ← 22
CC3	SUBI R1, R1, 11	VR3 ← R1	VR3 (R1) ← VR2 (R1) - 11 VR6 (R1) ← VR4 (R1) AND 33
CC4	SW [0x4], R1		[0x4] ← VR3 (R1) VR7 (R2) ← VR5 (R2) + VR6 (R1)
CC5	LW R1, [0x1234] ADDI R2, R0, 22	VR4 ← R1 VR5 ← R2	Retire in order R1 ← VR3 R1 ← VR6 R2 ← VR7
CC6	AND R1, R1, 33	VR6 ← R1	
CC7	ADD R2, R2, R1	VR7 ← R2	

Dynamic Scheduling

Hardware performs rescheduling (out-of-order execution)

- Execute program in efficient order
  - Not in program listing order
- Evolution of Tomasulo algorithm (IBM 1967)

Fast hardware decoder

- Decodes many instructions into an instruction queue
- Automatic register renaming with virtual registers

Mark instructions with status fields

NR — Not Ready / **R** — Ready / **F** — Finished

Execution

- Issue **R** instructions from instruction pool
- Return **F** instructions to instruction pool and update destination **VR**

Retirement

- Update architectural state in original program order

Tomasulo Algorithm

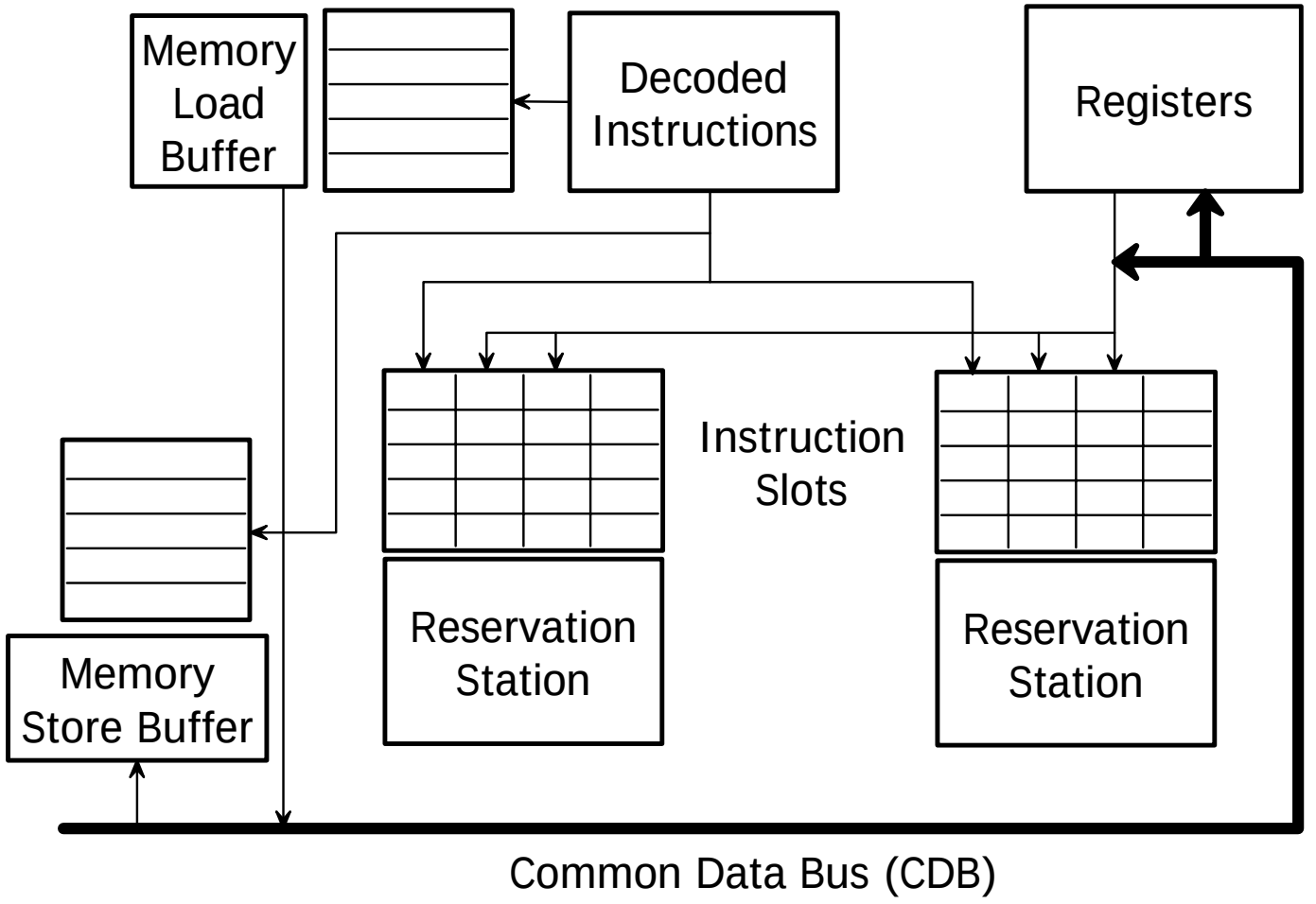
Original dynamic rescheduling algorithm

Virtual register architecture

Automatic register renaming and out-of-order execution

Distributed control mechanism

Used in IBM 360/91 floating point unit (1969)



Ref: Robert Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM Journal of Research and Development, (1967) 11.

Tomasulo Algorithm — General Description

Reservation station (RS)

- Execution unit (EU) for particular operation type
- Virtual registers (VR)
- Load/store buffers — read/write reservation stations

Decoder

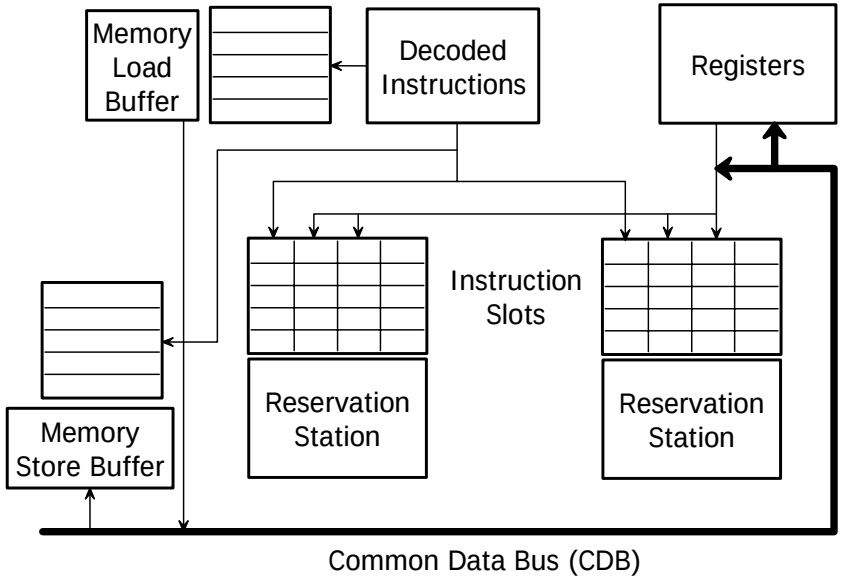
- Distributes instructions to Instruction Slot for reservation station
- Allocates VR to all destination operands (no RAT)
- Source operand
  - Ready at decode time $\Rightarrow$ operand = source value
  - Not ready $\Rightarrow$ operand = destination VR of unfinished instruction
    - VR = pointer to instruction in particular RS

Execution order

- RS executes ready instructions
- Broadcasts VR pointers and result values on data bus (CDB)
- Instruction Slots update VR values from CDB

Tomasulo Algorithm — Example

Program	Decode	Allocation / Issue
`LW R1,[0x4]`	`VR1 (R1) ← [0x4]`	**Assign value at allocation**
`MUL R1, R1, R2`	`VR2 (R1) ← VR1 (R1) * regs[R2]`	Issue with values VR1, R2
`SUBI R1, R1,0x22`	`VR3 (R1) ← VR2 (R1) - 0x22`	Issue with pointer VR2, imm
`SW [0x4],R1`	`[0x4] ← VR3 (R1)`	Issue with pointer VR3
`LW R1,[0x33]`	`VR4 (R1) ← [0x33]`	**Assign value at allocation**
`ADDI R2, R0, 0x44`	`VR5 (R2) ← 0x44`	**Assign value at allocation**
`AND R1, R1, 0x7F`	`VR6 (R1) ← VR4 (R1) AND 0x7F`	Issue with value VR4, imm
`ADD R2, R2, R1`	`VR7 (R2) ← VR5 (R2) + VR6 (R1)`	Issue with pointers VR5, VR6



Tomasulo Algorithm — Example

Program	Decoding	CC-1
`LW R1,[0x4]`	`VR1 (R1) ← [0x4]`	F
`MUL R1, R1, R2`	`VR2 (R1) ← VR1 (R1) * regs[R2]`	R
`SUBI R1, R1,0x22`	`VR3 (R1) ← VR2 (R1) – 0x22`	NR
`SW [0x4],R1`	`[0x4] ← VR3 (R1)`	NR
`LW R1,[0x33]`	`VR4 (R1) ← [0x33]`	F
`ADDI R2, R0, 0x44`	`VR5 (R2) ← 0x44`	F
`AND R1, R1, 0x7F`	`VR6 (R1) ← VR4 (R1) AND 0x7F`	R
`ADD R2, R2, R1`	`VR7 (R2) ← VR5 (R2) + VR6 (R1)`	NR

Assigned values

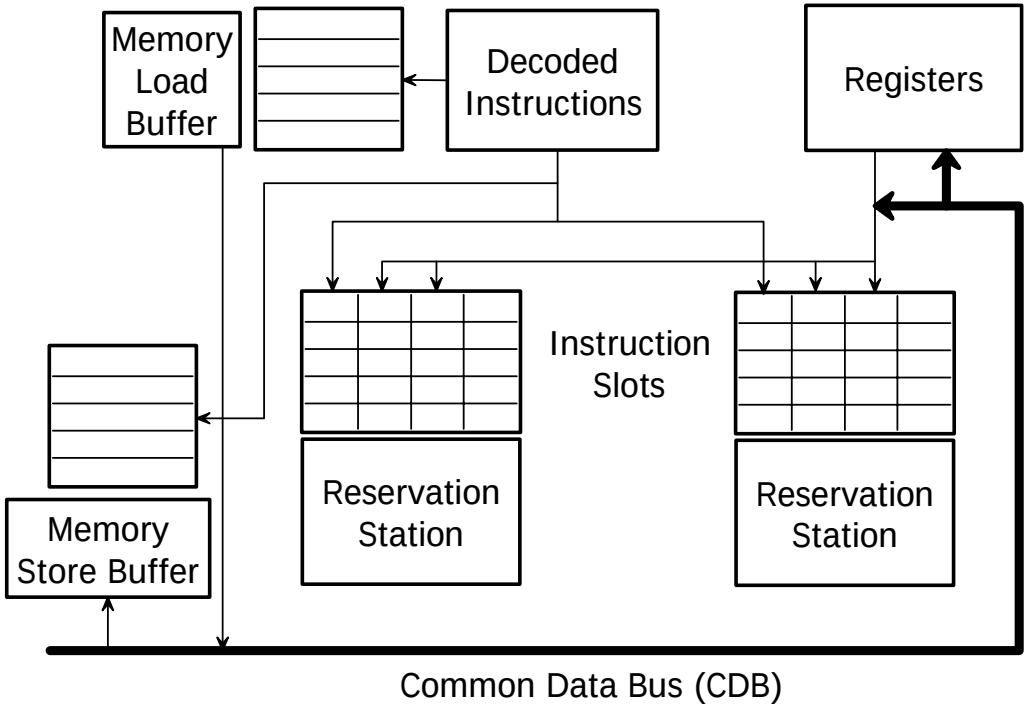
`VR1, VR4, VR5`

Issued with values

`MUL, AND`

Issued with pointers

`SUBI, SW, ADD`



Tomasulo Algorithm — Example

Program	Decoding	CC-2
`LW R1,[0x4]`	`VR1 (R1) ← [0x4]`	F
`MUL R1, R1, R2`	`VR2 (R1) ← VR1 (R1) * regs[R2]`	F
`SUBI R1, R1,0x22`	`VR3 (R1) ← VR2 (R1) – 0x22`	R
`SW [0x4],R1`	`[0x4] ← VR3 (R1)`	NR
`LW R1,[0x33]`	`VR4 (R1) ← [0x33]`	F
`ADDI R2, R0, 0x44`	`VR5 (R2) ← 0x44`	F
`AND R1, R1, 0x7F`	`VR6 (R1) ← VR4 (R1) AND 0x7F`	F
`ADD R2, R2, R1`	`VR7 (R2) ← VR5 (R2) + VR6 (R1)`	R

Execute

`MUL, AND`

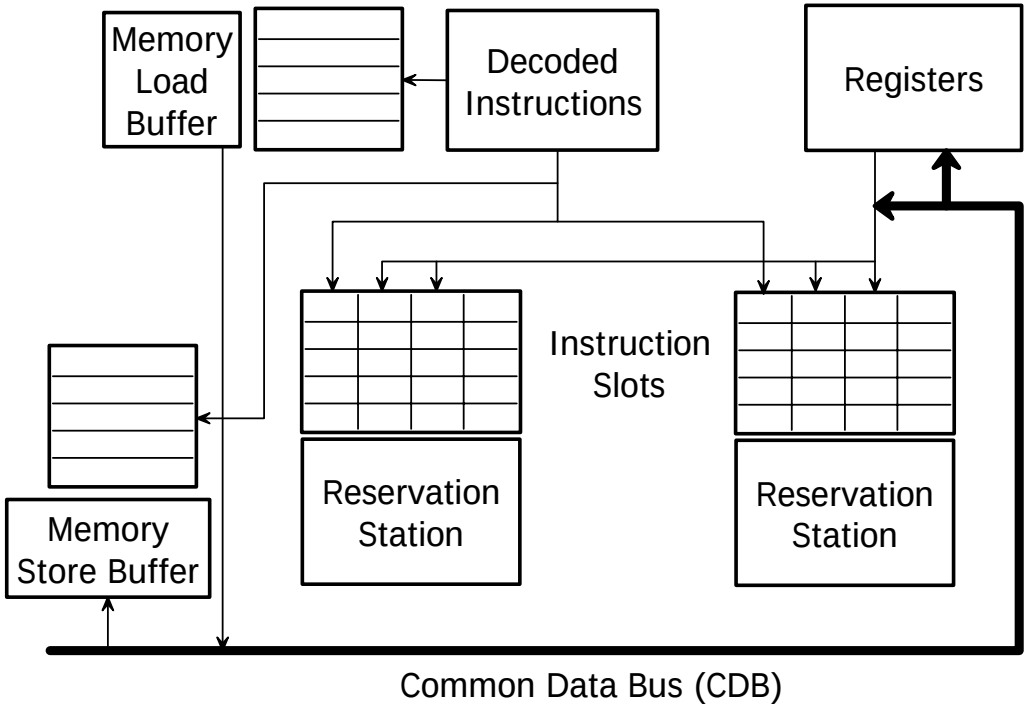
Broadcast updates on CDB

`VR2, VR6`

Replace pointers with values

`SUBI – VR2`

`ADD – VR6`



Tomasulo Algorithm — Example

Program	Decoding	CC-3
`LW R1,[0x4]`	`VR1 (R1) ← [0x4]`	F
`MUL R1, R1, R2`	`VR2 (R1) ← VR1 (R1) * regs[R2]`	F
`SUBI R1, R1,0x22`	`VR3 (R1) ← VR2 (R1) – 0x22`	F
`SW [0x4],R1`	`[0x4] ← VR3 (R1)`	F
`LW R1,[0x33]`	`VR4 (R1) ← [0x33]`	F
`ADDI R2, R0, 0x44`	`VR5 (R2) ← 0x44`	F
`AND R1, R1, 0x7F`	`VR6 (R1) ← VR4 (R1) AND 0x7F`	F
`ADD R2, R2, R1`	`VR7 (R2) ← VR5 (R2) + VR6 (R1)`	F

Execute

SUBI, ADD, SW

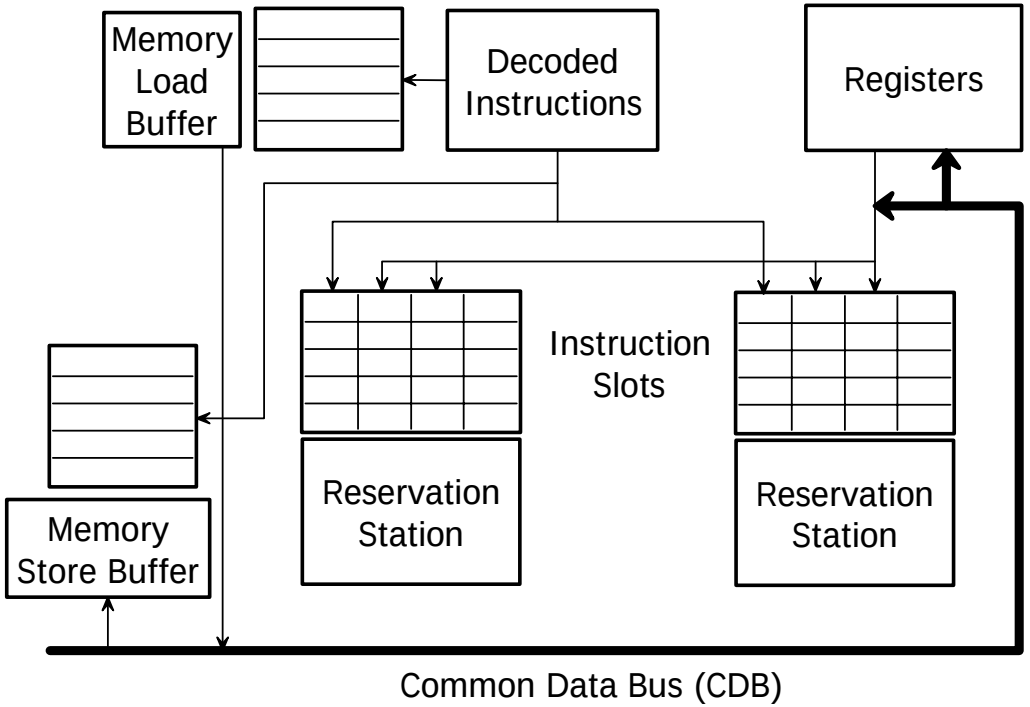
Broadcast updates on CDB

VR3, VR7

Replace pointers with values

[0x4] – VR3

R2 – VR7



Issues in Tomasulo Algorithm

Scheduling similar to scoreboarding

- Instructions execute when source operands are ready

Distributed control simplifies implementation

- Automatic register renaming
- Decoder keeps list of allocated ARs
- No centralized register allocation table (RAT)
  - RAT records all VR allocations
- No central scoreboard controller

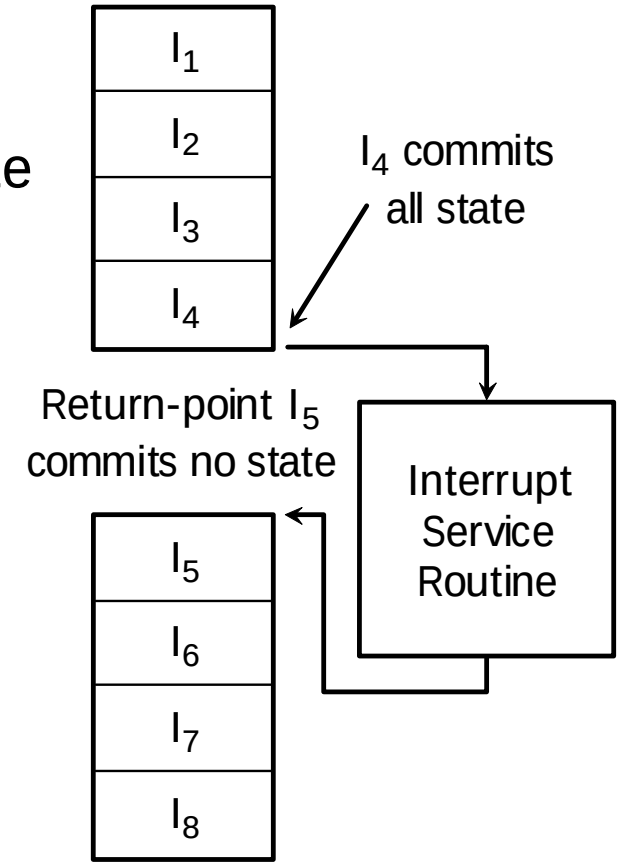
Difficulties with Tomasulo algorithm

- Common Data Bus $\rightarrow$ communication bottleneck
- Instructions complete out-of-order
  - Instructions following conditional branch — **NR**
    - No way to handle branch mispredictions $\Rightarrow$ no branch prediction
  - No well-defined return-point between instructions
    - No precise breakpoint for exceptions and interrupts

No Precise Exceptions in Tomasulo Machine

Return-point

- Follows atomic operation
- Previous operations commit all results to state
- No following operations commit any results to state



Precise exception

- Exception with well-defined return-point
- Service exception following atomic operation
- Restart execution at return point without error

In Tomasulo algorithm

- PC advances after fetching instruction
- Complete in order of execution (out-of-order)
  - Broadcast finished results on CDB
- No return-point
  - PC points past instructions that have committed to state
  - Some instructions have committed state before prior instructions that have not

In-Order Retirement

Set retirement order at decode time

- Retirement order = program order (PC)
  - Instructions commit to system state in retirement order
- Program progress index = retirement order
  - Enables speculative execution — issue beyond undetermined branch
  - Enables precise exceptions

Retirement queue indexed by retirement order

- Instruction → retirement queue when
  - Marked as finished
  - Verified as valid — no errors / no exceptions / no mispredictions

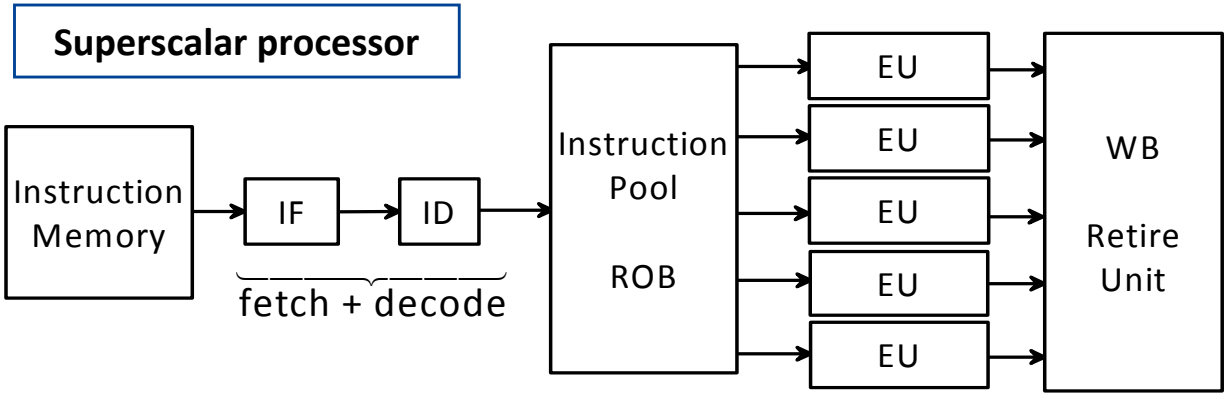
On exception / misprediction

- Identify last valid instruction
- Cancel instructions with later retirement order
- Exception
  - Branch to exception handler
  - Return point = next instruction following last valid

Dynamic Scheduling — Tomasulo + In-Order Retirement

Reorder Buffer (ROB)

Instruction pool



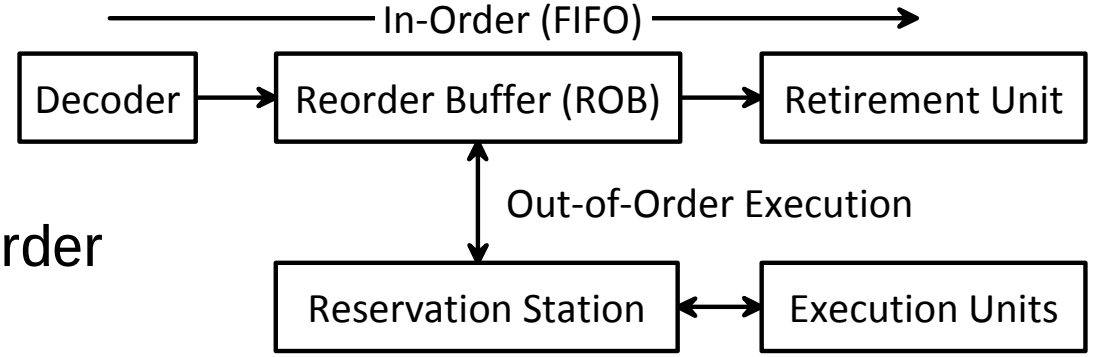
ROB entry

Instruction PC	Operation	Status	VR pointers + stable values

In-Order (FIFO) path

Decode → ROB → Retire

Retirement Unit commits in PC order



Out-of-Order execution path

ROB ↔ Execution Units

RS issues Ready instructions to EU

EU updates VR pointers + status fields in ROB entry

In-Order Completion Issues

Reorder buffer performs data forwarding

```
I_k   ADD R1, R2, R3
I_k+j ADD R4, R5, R1
```

I_k condition at I_{k+j} decode time	Decoded **R1** value for I_{k+j} entry in ROB
Retired	Value of **R1** from architectural register
Finished but not retired	Value of I_k destination entry
Not finished	source **VR** in I_{k+j} entry = destination **VR** in I_k entry

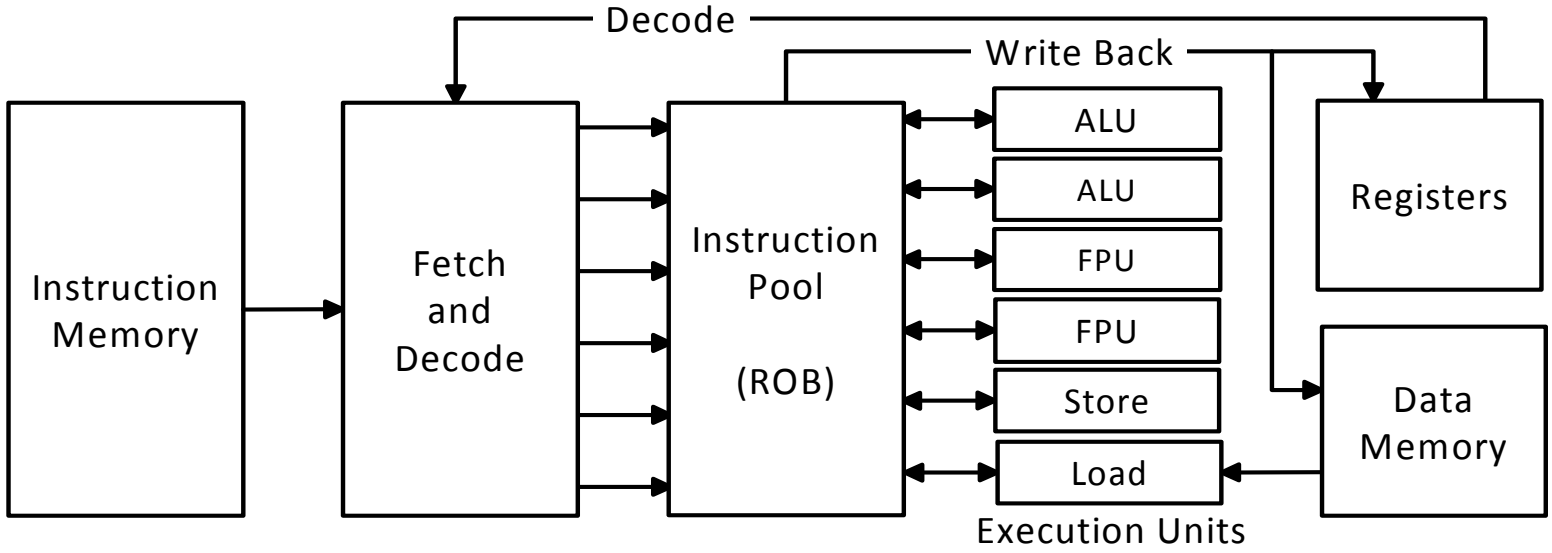
On exception

Cancel invalid instructions in ROB

Finished instructions following exception

New decoded instructions moved into ROB

Intel x86 with Dynamic Scheduling



Intel meta-architecture for Pentium II, III, 4, Multicore, ...

RISC machine with IA-32 CISC adaptation layer

Fetch + Decode

Converts IA-32 instructions to 1 – 6 RISC micro-ops per CC

Dynamic scheduling (out-if-order execution)

ROB holds micro-ops until ready for execution

Scheduler issues micro-ops to parallel execution in ALU, FPU, Load, Store

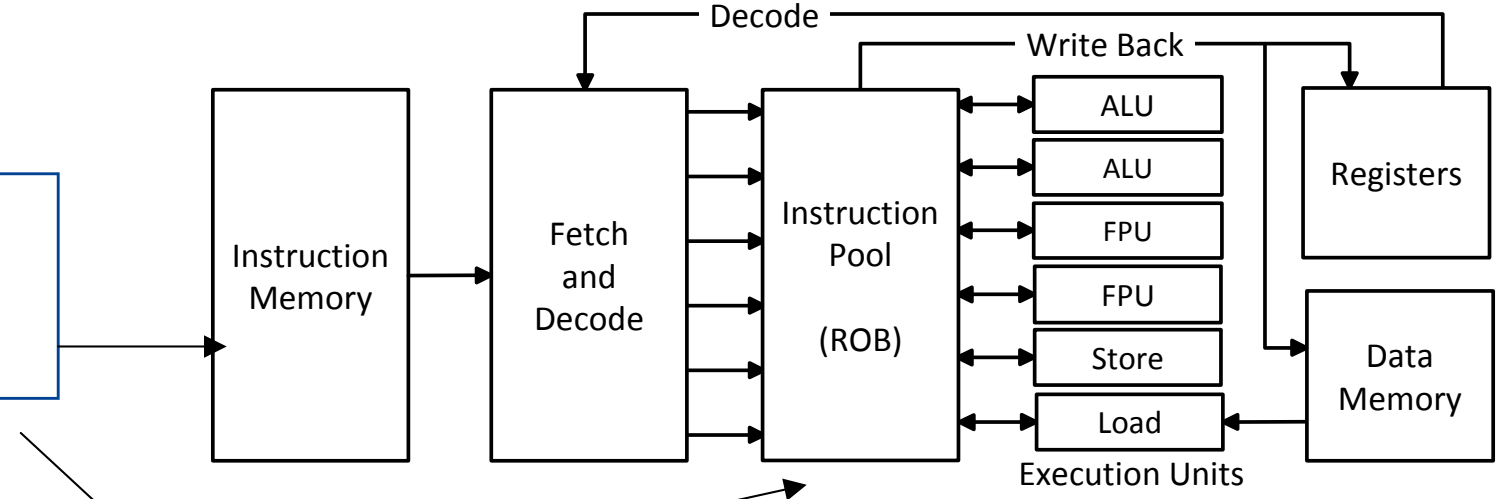
Finished micro-ops update execution results to instruction pool

Retirement (in-order write back)

Finished micro-ops commit in original program order

Program Execution in x86 with Dynamic Scheduling

```
ADD [X],123
SUB [Y],456
SUB [Z],789
```



IA-32 instructions decoded to RISC micro-ops with register renaming in 2 CC

```
LW   VR2,[X]
ADD  VR3,VR2,#123
SW   [X],VR3
LW   VR4,[Y]
SUB  VR5,VR4,#567
SW   [Y],VR5
LW   VR6,[Z]
SUB  VR7,VR6,#789
SW   [Z],VR7
```

Dynamic scheduling

`LW VR2,[X]`	Load	CC1
`ADD VR3,VR2,#123` `LW VR4,[Y]`	ALU Load	CC2

`SW [X],R3`
`SUB VR5,VR4,#567`
`LW VR6,[Z]`
	Store
	ALU
	Load
		CC3

`SW [Y],VR5` `SUB VR7,VR6,#789`	Store ALU	CC4
`SW [Z],VR7`	Store	CC5

Scoreboard Example x86 with Dynamic Scheduling

Execution condition after each CC (ignoring fetch, decode, retire)

Instruction	Execution Unit	CC1	CC2	CC3	CC4	CC5	CC6
`LW VR2,[X]`	Load	**F**	F	F	F	F	
`ADD VR3,VR2,#123`	ALU	R	**F**	F	F	F	
`SW [X],VR3`	Store	NR	R	**F**	F	F	
`LW VR4,[Y]`	Load	R	**F**	F	F	F	
`SUB VR5,VR4,#456`	ALU	NR	R	**F**	F	F	
`SW [Y],VR5`	Store	NR	NR	R	**F**	F	
`LW VR6,[Z]`	Load	R	R	**F**	F	F	
`SUB VR7,VR6,#789`	ALU	NR	NR	R	**F**	F	
`SW [Z],VR7`	Store	NR	NR	NR	R	**F**	

Scheduling rules

Issued only ready instructions

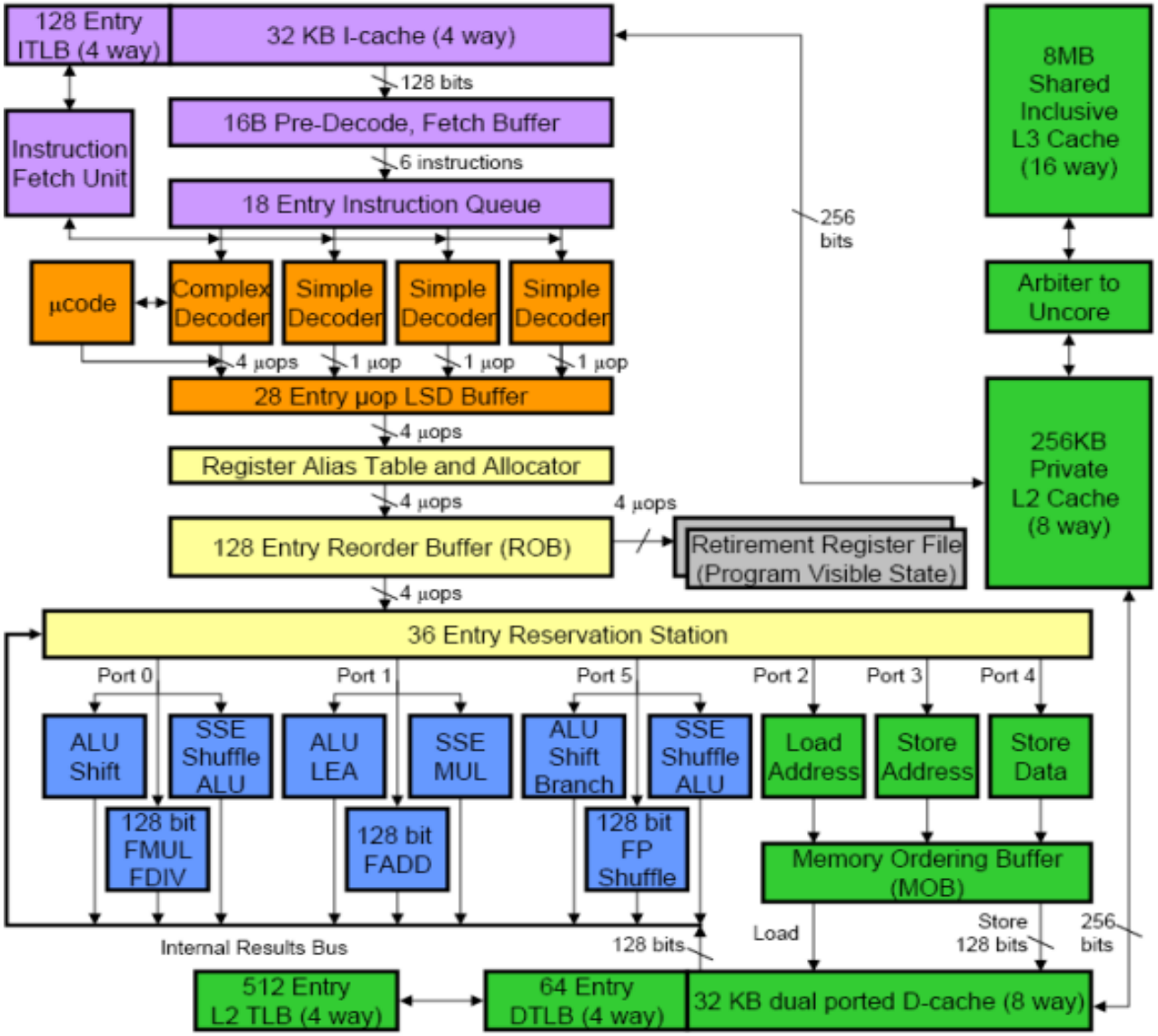
Among ready instructions, maintain program list order

Only 1 load and 1 store per CC

Up to 2 ALU and 2 FPU instructions per CC

NR	Not Ready
R	Ready
F	Finished

Intel Nehalem Micro-Architecture



David Kanter, "Inside Nehalem: Intel's Future Processor and System", http://realworldtech.com/includes/templates/articles.cfm?ArticleID=RWT040208182719

Deep Superpipeline

Each pipeline stage → 2 smaller stages

New stage does ½ work in ½ time

New stage finishes in ½ time ⇒ clock speed → 2 × clock speed

Deep superpipeline for DLX

	1	2	3	4	5	6	7	8	9	10	11	12
I_1	IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2		
I_2		IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2	
I_3			IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2

$$2 \times \text{clock speed} \Rightarrow \tau_{\text{superpipeline}} = \frac{\tau_{\text{pipeline}}}{2}$$

$$CPI^{\text{ideal}}_{\text{superpipeline}} = \frac{IC + 9}{IC} \xrightarrow[IC \to \text{large}]{} 1$$

$$T^{\text{ideal}}_{\text{superpipeline}} = CPI^{\text{ideal}}_{\text{superpipeline}} \times IC \times \frac{\tau_{\text{pipeline}}}{2} \xrightarrow[IC \to \text{large}]{} IC \times \frac{\tau_{\text{pipeline}}}{2} = \frac{1}{2} T \times^{\text{ideal}}_{\text{pipeline}}$$

Problems with deep superpipeline

Cannot effectively split all instructions

Some operations do not scale — require fixed time

Page fault, branch penalty, cache update, etc

Faster clock ⇒ more stall cycles ⇒ larger CPI^{stall}

Pentium 4 Superpipeline

Pentium III

10 stage pipeline — clock speed ≤ 1.5 GHz

Pentium 4

20 stage pipeline — clock speed ≤ 4.0 GHz

Expect

1.5 GHz processor is 50% faster than same processor at 1.0 GHz

Measurement on SPEC CINT2000

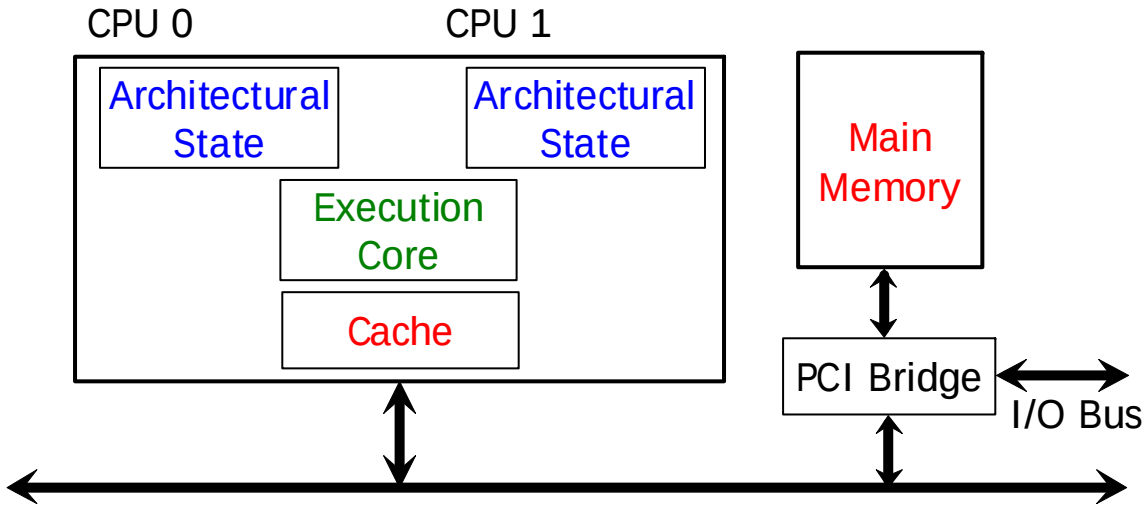
1.5 GHz Pentium-4 is 20% faster than 1.0 GHz Pentium-III

$$S = \frac{CPI^{\text{PIII}} \times IC^{\text{PIII}} \times \frac{1}{1.0\ \text{GHz}}}{CPI^{\text{P4}} \times IC^{\text{P4}} \times \frac{1}{1.5\ \text{GHz}}} = \frac{CPI^{\text{PIII}}}{CPI^{\text{P4}}} \times \frac{1.5}{1.0} = 1.2 \Rightarrow \boxed{CPI^{\text{P4}} = 1.25 \times CPI^{\text{PIII}}}$$

$$CPI^{\text{P4}}_{\text{ideal}} + CPI^{\text{P4}}_{\text{stall}} = 1.25 \times \left(CPI^{\text{PIII}}_{\text{ideal}} + CPI^{\text{PIII}}_{\text{stall}} \right)$$

$$CPI^{\text{P4}}_{\text{ideal}} = CPI^{\text{PIII}}_{\text{ideal}} = 1 \Rightarrow \boxed{CPI^{\text{P4}}_{\text{stall}} = 1.25 \times CPI^{\text{PIII}}_{\text{stall}} + 0.25 \approx 0.5}$$

Hyper-Threading®



Architectural State

Registers, stack pointers and program counter

Execution Core

ALU, FPU, vector processors, memory unit

Two copies of architectural state + one execution core

- OS sees two registers sets — looks like two CPUs
- OS assigns threads to CPU 0 and CPU 1

CPU 0 and CPU 1 issue instructions to shared execution core

- No data hazard in either thread
  - CPU 0 and CPU 1 issue instructions on alternate clock cycles
- Data hazard in one thread
  - Other CPU issues instructions on each clock cycle until stall ends

Both CPUs keep working on most clock cycles

Expected Improvement from Hyper-Threading

Without hyper - threading

$$CPI^{\text{stall}} = \frac{\text{stall cycles}}{\text{stall}} \times P(\text{stall}) = CPS \times P_s$$

$CPS = $ cycles per stall

With hyper - threading

$$CPI^{\text{stall HT}} = \frac{\text{stall cycles}}{\text{stall}} \times P(\text{2 simultaneous stalls}) \approx CPS \times [P_s]^2$$

Speedup

$$S = \frac{CPI \times IC \times \tau}{CPI^{HT} \times IC^{HT} \times \tau^{HT}} = \frac{CPI}{CPI^{HT}} = \frac{1 + CPS \times P_s}{1 + CPS \times [P_s]^2}$$

Take for Pentium - 4

$$CPI^{\text{stall}} \approx 0.5$$

$$CPS = 2 \text{ cycles per stall} \Rightarrow P_s = 0.5 / 2 = 0.25$$

$$S = \frac{1 + 2 \times 0.25}{1 + 2 \times [0.25]^2} = 1.2$$

Measured improvement $= 1.2$

Intel, "Hyper-Threading Technology Architecture and Microarchitecture"

Hardware Utilization in x86 Example

Unit	CC1	CC2	CC3	CC4	CC5
ALU	IDLE	ADD R3,R2,#123	SUB R5,R4,#567	SUB R7,R6,#789	IDLE
ALU	IDLE	IDLE	IDLE	IDLE	IDLE
FPU	IDLE	IDLE	IDLE	IDLE	IDLE
FPU	IDLE	IDLE	IDLE	IDLE	IDLE
Load	LW R2,[X]	LW R4,[Y]	LW R6,[Z]	IDLE	IDLE
Store	IDLE	IDLE	SW [X],R3	SW [Y],R5	SW [Y],R7

6 EUs in x86 example

2 integer ALUs, 2 FPUs, 1 load unit, 1 store unit

Most units idle in most clock cycles

CC1 — 1 EU active

CC2 — 2 EUs active

CC3 — 3 EUs active

CC4 — 2 EUs active

CC5 — 1 EU active

Increased utilization ⇒ more parallel instructions

Overhead in Performance Calculations

Typical program structure

Head	setup instructions	run-once
Main functions	program loop instructions	large number of iterations
Tail	shutdown instructions	run-once

Dynamic Scheduling Superscalar with u Execution Units

`Fetch + Decode stages`	`Execute in EU_1`	`Retire stages`
Overhead cycles	`Execute in EU_2`	Overhead cycles
Fills Instruction Pool	…	Retires Instructions
Independent of IC	`Execute in EU_u`	Independent of IC

Pipelining overlaps fetch / decode / retire with execution in EUs

$$N = N_{IF+ID} + N_{program} + N_{retirement} = N_{IF+ID} + N_{head} + N_{main\ loops} + N_{tail} + N_{retirement}$$

$$= N_{overhead} + N_{main\ loops}$$

where $N_{overhead} = N_{IF+ID} + N_{head} + N_{tail} + N_{retirement}$

CPI Loop Approximation in Superscalar

General approximation

$$CPI = \frac{N_{\text{overhead}} + n_{\text{iterations}} \times N_{\text{one-loop}}}{IC_{\text{overhead}} + n_{\text{iterations}} \times IC_{\text{one-loop}}} = \frac{\frac{N_{\text{overhead}}}{n_{\text{iterations}}} + N_{\text{one-loop}}}{\frac{IC_{\text{overhead}}}{n_{\text{iterations}}} + IC_{\text{one-loop}}}$$

$$CPI \xrightarrow[n_{\text{iterations}} \to \text{large}]{} \frac{N_{\text{one-loop}}}{IC_{\text{one-loop}}} = CPI_{\text{one-loop}}$$

Ideal case

$$N_{\text{one-loop}} = \frac{N_{\text{one-loop}}^{u=1}}{u}$$

$$CPI = \frac{1}{u}\frac{N_{\text{one-loop}}^{u=1}}{IC_{\text{one-loop}}} = \frac{1}{u}CPI_{\text{ideal}}^{u=1} = \frac{1}{u}$$

Loop Performance Example

Loop runs 0x1000/4 = 0x400 = 1024 iterations

		1	2	3	4	
ADDI R1, R0, #1000	EX					
L1: SUBI R1, R1, #4		EX				1
LW R2, 0(R1)			EX			2
ADDI R2, R2, #1				EX		3
SW 0(R1), R2					EX	4
BNEZ R1, L1					EX	5
overhead = fetch + decode + 1 CC + retire		loop = 4 CC				

$$CPI = \frac{overhead + 4 \times 1024}{1 + 5 \times 1024} \rightarrow 0.80$$

Loop Unrolling

Iterations of loop instructions create data dependencies

Successive iterations use same registers Cannot run in parallel	Replicate instructions over shorter loop Register renaming Scheduled for parallel execution

`ADDI R1, R0, #1000`
`L1: SUBI R1, R1, #4`
`LW R2, 0(R1)`
`ADDI R2, R2, #1`
`SW 0(R1), R2`
`BNEZ R1, L1`
	`ADDI R1, R0, #1000`
	`L1: SUBI R1, R1, #10`
	`LW R2, 0(R1)`
	`LW R3, 4(R1)`
	`LW R4, 8(R1)`
	`LW R5, C(R1)`
	`ADDI R2, R2, #1`
	`ADDI R3, R3, #1`
	`ADDI R4, R4, #1`
	`ADDI R5, R5, #1`
	`SW 0(R1), R2`
	`SW 4(R1), R3`
	`SW 8(R1), R4`
	`SW C(R1), R5`
	`BNEZ R1, L1`

Loop Unrolling Performance — 1

EUs: 2 ALUs + 1 Read + 1 Write

		1	2	3	4	5	6	7	
ADDI R1, R0, #1000	F								
SUBI R1, R1, #10	R	F	F	F	F	F	F	F	1
LW R2, 0(R1)	NR	R	F	F	F	F	F	F	2
LW R3, 4(R1)	NR	R	R	F	F	F	F	F	3
LW R4, 8(R1)	NR	R	R	R	F	F	F	F	4
LW R5, C(R1)	NR	R	R	R	R	F	F	F	5
ADDI R2, R2, #1	NR	NR	R	F	F	F	F	F	6
ADDI R3, R3, #1	NR	NR	NR	R	F	F	F	F	7
ADDI R4, R4, #1	NR	NR	NR	NR	R	F	F	F	8
ADDI R5, R5, #1	NR	NR	NR	NR	NR	R	F	F	9
SW 0(R1), R2	NR	NR	NR	R	F	F	F	F	10
SW 4(R1), R3	NR	NR	NR	NR	R	F	F	F	11
SW 8(R1), R4	NR	NR	NR	NR	NR	R	F	F	12
SW C(R1), R5	NR	NR	NR	NR	NR	NR	R	F	13
BNEZ R1, L1	NR	R	R	R	R	R	R	F	14

$$CPI = \frac{\text{overhead} + N_{\text{1-loop}} \times N_{\text{loop iterations}}}{IC_{\text{head}} + IC_{\text{1-loop}} \times N_{\text{loop iterations}} + IC_{\text{tail}}} = \frac{\text{overhead} + 7 \times 256}{1 + 14 \times 256 + 0} = 0.50$$

Loop Unrolling Performance — 2

EUs: 4 ALUs + 4 Read + 4 Write

		1	2	3	4	
ADDI R1, R0, #1000	F					
SUBI R1, R1, #10	R	F	F	F	F	1
LW R2, 0(R1)	NR	R	F	F	F	2
LW R3, 4(R1)	NR	R	F	F	F	3
LW R4, 8(R1)	NR	R	F	F	F	4
LW R5, C(R1)	NR	R	F	F	R	5
ADDI R2, R2, #1	NR	NR	R	F	F	6
ADDI R3, R3, #1	NR	NR	R	F	F	7
ADDI R4, R4, #1	NR	NR	R	F	R	8
ADDI R5, R5, #1	NR	NR	R	F	NR	9
SW 0(R1), R2	NR	NR	NR	R	F	10
SW 4(R1), R3	NR	NR	NR	R	F	11
SW 8(R1), R4	NR	NR	NR	R	F	12
SW C(R1), R5	NR	NR	NR	R	F	13
BNEZ R1, L1	NR	R	R	R	F	14

$$CPI = \frac{\text{overhead} + N_{\text{1-loop}} \times N_{\text{loop iterations}}}{IC_{\text{head}} + IC_{\text{1-loop}} \times N_{\text{loop iterations}} + IC_{\text{tail}}} = \frac{\text{overhead} + 4 \times 256}{1 + 14 \times 256 + 0} = 0.29$$

Loop Unrolling with x86 VR Renaming — 1

Simple x86 code fragment

Sets counter , loads word to register, updates register from table, stores word in register, loops while counter > 0

```
     MOV ECX, 400h          ; reg[ECX] ← 400h = 1K = 1024
L1:  LODSD                  ; reg[EAX] ← mem[reg[ESI]]
                            ; reg[ESI] ← reg[ESI] + 4
     ADD EAX,[EBX + ESI]    ; reg[EAX] ← reg[EAX] + table
                            ; entry at mem[reg[EBX]+reg[ESI]]
     STOSD                  ; mem[EDI] ← reg[EAX]
                            ; reg[EDI] ← reg[EDI] + 4
     LOOP L1                ; reg[ECX] ← reg[ECX] - 1
                            ; branch to L1 if reg[ECX] > 0
```

Loop Unrolling with x86 VR Renaming — 2

x86 Machine Instruction	VR Allocation	Recoded Instruction
ECX ← 400h = 1K = 1024	VR1 ← ECX	VR1 ← 400h
L1: EAX ← (ESI) ESI ← ESI + 4	VR2 ← EAX VR3 ← ESI	VR2 ← (reg[ESI]) VR3 ← reg[ESI] + 4
EAX ← EAX + (EBX+ESI)	VR4 ← EAX	VR4 ← VR2 + (reg[EBX]+VR3)
(EDI) ← EAX EDI ← EDI + 4	VR5 ← EDI	(reg[EDI]) ← VR4 VR5 ← reg[EDI] + 4
		ESI ← VR3 // commit VR3 EDI ← VR5 // commit VR5
ECX ← ECX – 1 branch L1 if ECX > 0	VR9 ← ECX	VR9 ← VR1 – 1 branch L1 if VR9 > 0

Loop Unrolling with x86 VR Renaming — 3

Loop unrolling

Define two parallel pointers for read pointer

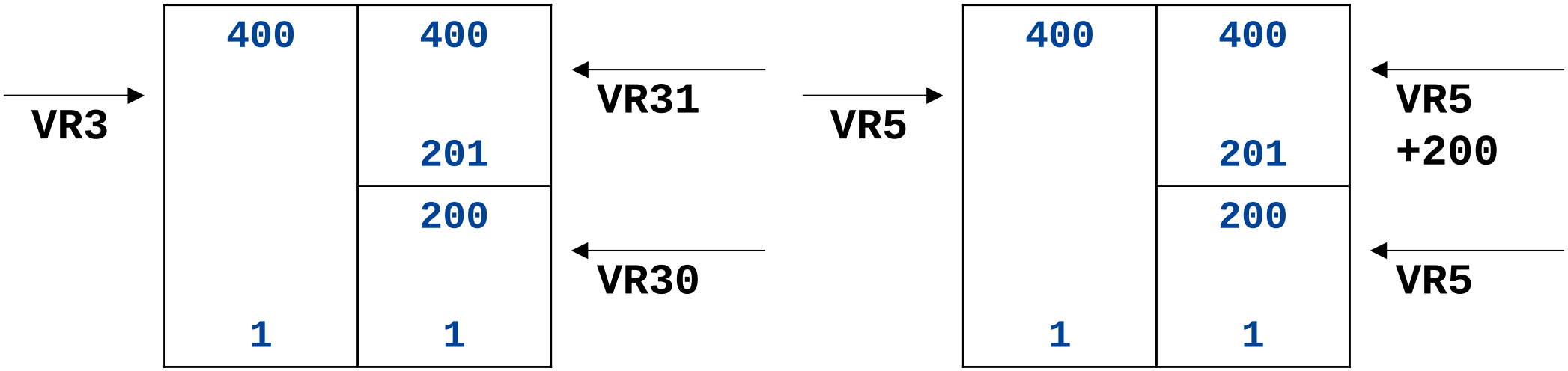
VR3 ← ESI ⇒ VR30 ← ESI VR31 ← ESI + 200

Perform two ALU operations per loop

Enhance EU utilization

Half number of loop iterations

Program runs twice as fast



Loop Unrolling with x86 VR Renaming — 4

VR1 ← 400h	VR1 ← 200h
L1: VR2 ← (reg[ESI])	L1: VR20 ← (reg[ESI])
	VR21 ← (reg[ESI]+200)
VR3 ← reg[ESI] + 4	VR30 ← reg[ESI] + 4
	VR31 ← reg[ESI] + 204
VR4 ← VR2 + (reg[EBX]+VR3)	VR40 ← VR20 + (reg[EBX]+VR30)
	VR41 ← VR21 + (reg[EBX]+VR31)
(reg[EDI]) ← VR4	(reg[EDI]) ← VR40
	(reg[EDI]+200) ← VR41
VR5 ← reg[EDI] + 4	VR5 ← reg[EDI] + 4
ESI ← VR3 // commit VR3 EDI ← VR5 // commit VR5	ESI ← VR30 // commit VR30 EDI ← VR5 // commit VR5
VR9 ← VR1 – 1 branch L1 if VR9 > 0	VR9 ← VR1 – 1 branch L1 if VR9 > 0

Loop Unrolling with Automatic VR Renaming in x86 — 3

Loop before unrolling

4 CC per loop iteration

1024 / 4 = 256 iterations

EUs: 4 ALUs + 4 Read + 4 Write

	1	2	3	4
L1: VR2 ← (reg[ESI])	F	F	F	F
VR3 ← reg[ESI] + 4	F	F	F	F
VR4 ← VR2 + (reg[EBX]+VR3)	R	F	F	F
(reg[EDI]) ← VR4	NR	R	F	F
VR5 ← reg[EDI] + 4	NR	R	F	F
ESI ← VR3 // commit VR3 EDI ← VR5 // commit VR5	NR	NR	R	F
VR9 ← VR1 – 1 branch L1 if VR9 > 0	NR	NR	R	F

Loop Unrolling with Automatic VR Renaming in x86 — 4

Loop before unrolling

4 CC per loop iteration

512 / 4 = 128 iterations ⇒ half the run time

	1	2	3	4
L1: VR20 ← (reg[ESI])	F	F	F	F
VR21 ← (reg[ESI]+200)	F	F	F	F
VR30 ← reg[ESI] + 4	F	F	F	F
VR31 ← reg[ESI] + 204	F	F	F	F
VR40 ← VR20 + (reg[EBX]+VR30)	R	F	F	F
VR41 ← VR21 + (reg[EBX]+VR31)	R	F	F	F
(reg[EDI]) ← VR40	NR	R	F	F
(reg[EDI]+200) ← VR41	NR	R	F	F
VR5 ← reg[EDI] + 4	NR	R	F	F
ESI ← VR30 // commit VR3 EDI ← VR50 // commit VR5	NR	NR	R	F
VR9 ← VR1 – 1 branch L1 if VR9 > 0	NR	NR	R	F

Software Pipelining

Simple sequential loop

```
R2 ← mem[index]
R3 ← ALU(R2, literal)
mem[index] ← R3
index ← index - offset
```

Pipelined version

Header

```
Initialize old_R2, old_R3, index
```

Loop

```
mem[index + offset] ← old_R3
new_R2 ← mem[index - offset]
new_R3 ← ALU(old_R2, literal)
new_index ← ALU(index, literal)
```

All loop instructions run in parallel

During loop iteration N

- All source reads were written during iteration N-1
- All destinations are read only during iteration N+1

Parallel read / write

Reads performed at start of clock cycle

Writes performed at end of clock cycle

Reads/Writes in same clock cycle are independent

Software Pipelining Example — 1

Sequential	Pipelined
`ADDI R1, R0, #1000 ;;`	`head: ADDI R1, R0, #1000 ;;`
`L1: SUBI R1, R1, #4 ;;`	`SUBI R1, R1, #4 ;;`
`LW R2, 0(R1) ;;`	`LW R2, 0(R1) ;;`

`ADDI R3, R2, #1 ;;`	`ADDI R3, R2, #1`
	`LW R2, -4(R1)`
	`SUBI R1, R1, #4 ;;`

`SW 0(R1), R3 ;;`	
`BNEZ R1, L1 ;;`	// first loop iteration uses source values set in head // iteration N+1 uses source values set in iteration N

Software pipelining requires explicit parallelization support in CPU hardware
Compiler marks groups of instructions to run in parallel
symbol `;;` marks end of parallel instruction group
	`L1: SUBI R1, R1, #4`
	`SW 4(R1), R3`
	`ADDI R3, R2, #1`
	`LW R2, -4(R1)`
	`BNEZ R1, L1 ;;`

	`tail: SW 0(R1), R3`

Software Pipelining Example — 2

Pipelined		R1	R2	R3	Read	Write
head:	ADDI R1, R0, #1000	1000	—	—	—	—
	SUBI R1, R1, #4	996	—	—	—	—
	LW R2, 0(R1)		(996)	—	996	—
	ADDI R3, R2, #1			(996)+1		—
	LW R2, -4(R1)		(992)		992	—
	SUBI R1, R1, #4	992				
L1:	SUBI R1, R1, #4	988				
	SW 4(R1), R3					996
	ADDI R3, R2, #1			(992)+1		
	LW R2, -4(R1)		(988)			
	BNEZ R1, L1					
L1:	SUBI R1, R1, #4	984				
	SW 4(R1), R3					992
	ADDI R3, R2, #1			(988)+1		
	LW R2, -4(R1)		(984)			
	BNEZ R1, L1					

Software Pipelining Example — 3

Pipelined		Action
head:	ADDI R1, R0, #1000	R1 ← 1000
	SUBI R1, R1, #4	R1 ← 1000-4 = 996
	LW R2, 0(R1)	R2 ← (996)
	ADDI R3, R2, #1	R3 ← (996)+1
	LW R2, -4(R1)	R2 ← (996-4) = (992)
	SUBI R1, R1, #4	R1 ← 996-4 = 992
L1:	SUBI R1, R1, #4	R1 ← 992-4 = 988
	SW 4(R1), R3	(992+4) = (996) ← (996)+1
	ADDI R3, R2, #1	R3 ← (992)+1
	LW R2, -4(R1)	R2 ← (992-4) = (988)
	BNEZ R1, L1	BNEZ 992, L1
L1:	SUBI R1, R1, #4	R1 ← 988-4 = 984
	SW 4(R1), R3	(988+4) = (992) ← (992)+1
	ADDI R3, R2, #1	R3 ← (988)+1
	LW R2, -4(R1)	R2 ← (992-4) = (988)
	BNEZ R1, L1	BNEZ 988, L1

Software Pipelining Example — 4

Pipelined		Action
head:	ADDI R1, R0, #12	R1 ← 12
	SUBI R1, R1, #4	R1 ← 12-4 = 8
	LW R2, 0(R1)	R2 ← (8)
	ADDI R3, R2, #1	R3 ← (8)+1
	LW R2, -4(R1)	R2 ← (8-4) = (4)
	SUBI R1, R1, #4	R1 ← 8-4 = 4
L1:	SUBI R1, R1, #4	R1 ← 4-4 = 0
	SW 4(R1), R3	(4+4) = (8) ← (8)+1
	ADDI R3, R2, #1	R3 ← (4)+1
	LW R2, -4(R1)	R2 ← (0-4) = (0)
	BNEZ R1, L1	BNEZ 4, L1
L1:	SUBI R1, R1, #4	R1 ← 0-4 = -4
	SW 4(R1), R3	(0+4) = (4) ← (4)+1
	ADDI R3, R2, #1	R3 ← (0)+1
	LW R2, -4(R1)	R2 ← (0-4) = (-4)
	BNEZ R1, L1	BNEZ 0, L1 // last iteration
tail:	SW 0(R1), R3	(0+0) = (0) ← (0)+1

Loop Unrolling + Software Pipelining

For unrolled program on slide 61

Requires 5 ALUs + 4 Reads + 4 Writes + Branch Unit for efficient execution

```
head: ADDI R1, R0, #1000 ;;
      SUBI R1, R1, #10 ;;
      LW R2, 0(R1)
      LW R3, 4(R1)
      LW R4, 8(R1)
      LW R5, C(R1) ;;
      ADDI R6, R2, #1
      ADDI R7, R3, #1
      ADDI R8, R4, #1
      ADDI R9, R5, #1
      LW R2, -10(R1)
      LW R3, -0C(R1)
      LW R4, -08(R1)
      LW R5, -04(R1)
      SUBI R1, R1, #10 ;;
```

```
L1:   SUBI R1, R1, #10
      SW 10(R1), R6
      SW 14(R1), R7
      SW 18(R1), R8
      SW 1C(R1), R9
      ADDI R6, R2, #1
      ADDI R7, R3, #1
      ADDI R8, R4, #1
      ADDI R9, R5, #1
      LW R2, -10(R1)
      LW R3, -0C(R1)
      LW R4, -08(R1)
      LW R5, -04(R1)
      BNEZ R1, L1 ;;
```

```
tail: SW 0(R1), R6
      SW 4(R1), R7
      SW 8(R1), R8
      SW C(R1), R9
```

Parallel reads/writes

Independent operations

Example:

ADDI R6, R2, #1

reads **OLD R2**

at **START** of CC

LW R2, 0(R1)

writes **NEW R2**

at **END** of CC

Software Pipelining Performance

			1	
head	15 instructions in 5 CC			
SUBI R1, R1, #10		R	F	1
SW 10(R1), R6		R	F	2
SW 14(R1), R7		R	F	3
SW 18(R1), R8		R	F	4
SW 1C(R1), R9		R	F	5
ADDI R6, R2, #1		R	F	6
ADDI R7, R3, #1		R	F	7
ADDI R8, R4, #1		R	F	8
ADDI R9, R5, #1		R	F	9
LW R2, -10(R1)		R	F	10
LW R3, -0C(R1)		R	F	11
LW R4, -08(R1)		R	F	12
LW R5, -04(R1)		R	F	13
BNEZ R1, L1 ;;		R	F	14
…				
tail	4 sequential instructions			

$$CPI = \frac{\text{overhead} + N_{\text{1-loop}} \times N_{\text{loop iterations}}}{IC_{\text{head}} + IC_{\text{tail}} + IC_{\text{1-loop}} \times N_{\text{loop iterations}}} = \frac{\text{overhead} + 1 \times 256}{7 + 8 + 14 \times 256} \simeq 0.07$$

Improving x86 Hardware Utilization — New Directions

Unit	CC1	CC2	CC3	CC4	CC5
ALU	IDLE	ADD R2,R2,#123	SUB R3,R3,#567	SUB R4,R4,#789	IDLE
ALU	IDLE	IDLE	IDLE	IDLE	IDLE
FPU	IDLE	IDLE	IDLE	IDLE	IDLE
FPU	IDLE	IDLE	IDLE	IDLE	IDLE
Load	LW R2,[X]	LW R3,[Y]	LW R4,[Z]	IDLE	IDLE
Store	IDLE	IDLE	SW [X],R2	SW [Y],R3	SW [Y],R4

Hardware utilization

Most execution units idle in most clock cycles

Higher ILP ⇒ higher utilization of execution units

Higher utilization ⇒ more independent instructions to choose from

Speculation

Issue some instructions beyond undetermined conditional branch

Keep CPU busy — accurate prediction performs work before needed