

High Performance Processor Implementations CS 347 April 7 & 9, 1998

Intel Processors

- 486, Pentium, Pentium Pro

Superscalar Processor Design

- Use PowerPC 604 as case study
- Speculative Execution, Register Renaming, Branch Prediction

More Superscalar Examples

- MIPS R10000
- DEC Alpha 21264

Intel x86 Processors

Processor	Year	Transistors	MHz	Spec92 (Int/FP)	Spec95 (Int/FP)
8086	'78	29K	4		
Basis of IBM PC & PC-XT					
i286	'83	134K	8		
Basis of IBM PC-AT					
i386	'86	275K	16		
	'88		33	6 / 3	
i486	'89	1.2M	20		
			50	28 / 13	
Pentium	'93	3.1M	66	78 / 64	
			150	181 / 125	4.3 / 3.0
PentiumPro	'95	5.5M	150	245 / 220	6.1 / 4.8
			200	320 / 283	8.2 / 6.0
Pentium II	'97	7.5M	300		11.6 / 6.8
Merced	'98?	14M	?	?	?

Other Processors

Processor	Year	Transistors	MHz	Spec92	Spec95
MIPS R3000 (DecStation 5000/120)	'88		25	16.1 / 21.7	
MIPS R5000 (Wean Hall SGLs)		3.6M	180		4.1 / 4.4
MIPS R10000 (Most Advanced MIPS)	'95	5.9M	200	300 / 600	8.9 / 17.2
Alpha 21164a (Fastest Available)	'96	9.3M	417 500	500 / 750	11 / 17 12.6 / 18.3
Alpha 21264 (Fastest Announced)	'97	15M	500		30 / 60

Architectural Performance

Metric

- SpecX92/Mhz: Normalizes with respect to clock speed
- But ... one measure of good arch. is how fast can run clock

Sampling

Processor	MHz	SpecInt92	IntAP	SpecFP92	FltAP
i386/387	33	6	0.2	3	0.1
i486DX	50	28	0.6	13	0.3
Pentium	150	181	1.2	125	0.8
PentiumPro	200	320	1.6	283	1.4
MIPS R3000A	25	16.1	0.6	21.7	0.9
MIPS R10000	200	300	1.5	600	3.0
Alpha 21164a	417	500	1.2	750	1.8

x86 ISA Characteristics

Multiple Data Sizes and Addressing Methods

- Recent generations optimized for 32-bit mode

Limited Number of Registers

- Stack-oriented procedure call and FP instructions
- Programs reference memory heavily (41%)

Variable Length Instructions

- First few bytes describe operation and operands
- Remaining ones give immediate data & address displacements
- Average is 2.5 bytes

i486 Pipeline

Fetch

- Load 16-bytes of instruction into prefetch buffer

Decode1

- Determine instruction length, instruction type

Decode2

- Compute memory address
- Generate immediate operands

Execute

- Register Read
- ALU operation
- Memory read/write

Write-Back

- Update register file

Pipeline Stage Details

Fetch

- **Moves 16 bytes of instruction stream into code queue**
- **Not required every time**
 - About 5 instructions fetched at once
 - Only useful if don't branch
- **Avoids need for separate instruction cache**

D1

- **Determine total instruction length**
 - Signals code queue aligner where next instruction begins
- **May require two cycles**
 - When multiple operands must be decoded
 - About 6% of “typical” DOS program

Stage Details (Cont.)

D2

- **Extract memory displacements and immediate operands**
- **Compute memory addresses**
 - Add base register, and possibly scaled index register
- **May require two cycles**
 - If index register involved, or both address & immediate operand
 - Approx. 5% of executed instructions

EX

- **Read register operands**
- **Compute ALU function**
- **Read or write memory (data cache)**

WB

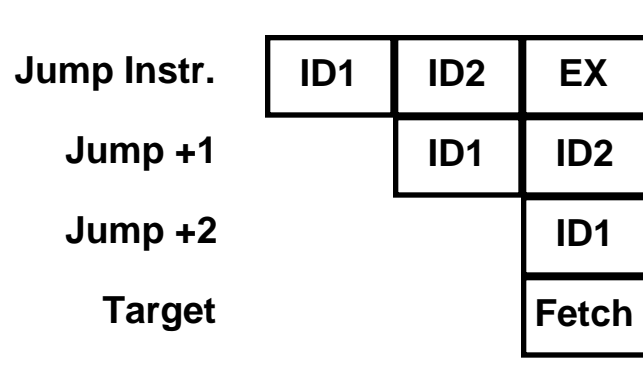
- **Update register result**

Data Hazards

Data Hazards

Generated	Used	Handling
ALU	ALU	EX-EX Forwarding
Load	ALU	EX-EX Forwarding
ALU	Store	EX-EX Forwarding
ALU	Eff. Address	(Stall) + EX-ID2 Forwarding

Control Hazards



Jump Instruction Processing

- Continue pipeline assuming branch not taken
- Resolve branch condition in EX stage
- Also speculatively fetch at target during EX stage

Control Hazards (Cont.)

Branch Not Taken

- Allow pipeline to continue.
- Total of 1 cycle for instruction

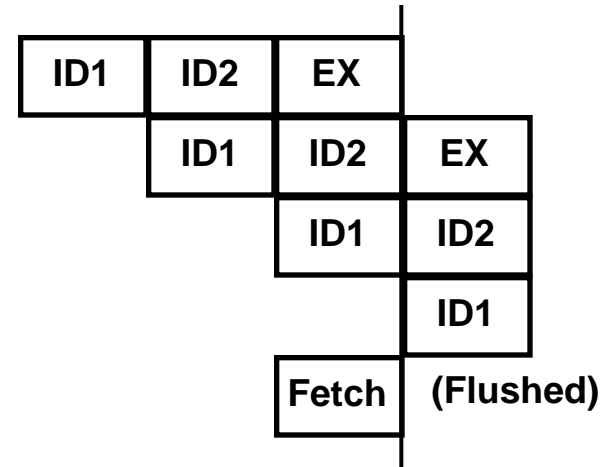
Jump Instr.

Jump +1

Jump +2

Jump +3

Target



Branch taken

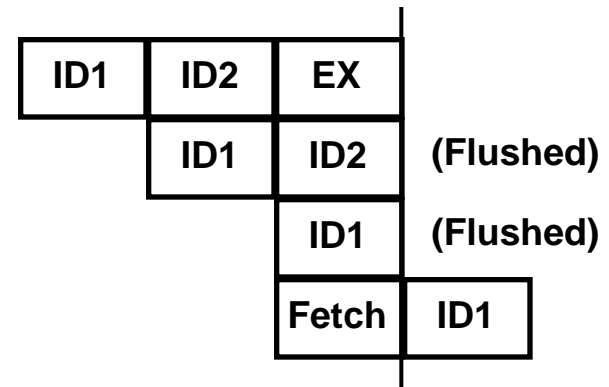
- Flush instructions in pipe
- Begin ID1 at target.
- Total of 3 cycles for instruction

Jump Instr.

Jump +1

Jump +2

Target



Comparison with Our pAlpha Pipeline

Two Decoding Stages

- Harder to decode CISC instructions
- Effective address calculation in D2

Multicycle Decoding Stages

- For more difficult decodings
- Stalls incoming instructions

Combined Mem/EX Stage

- Avoids load stall without load delay slot
 - But introduces stall for address computation

Comparison to 386

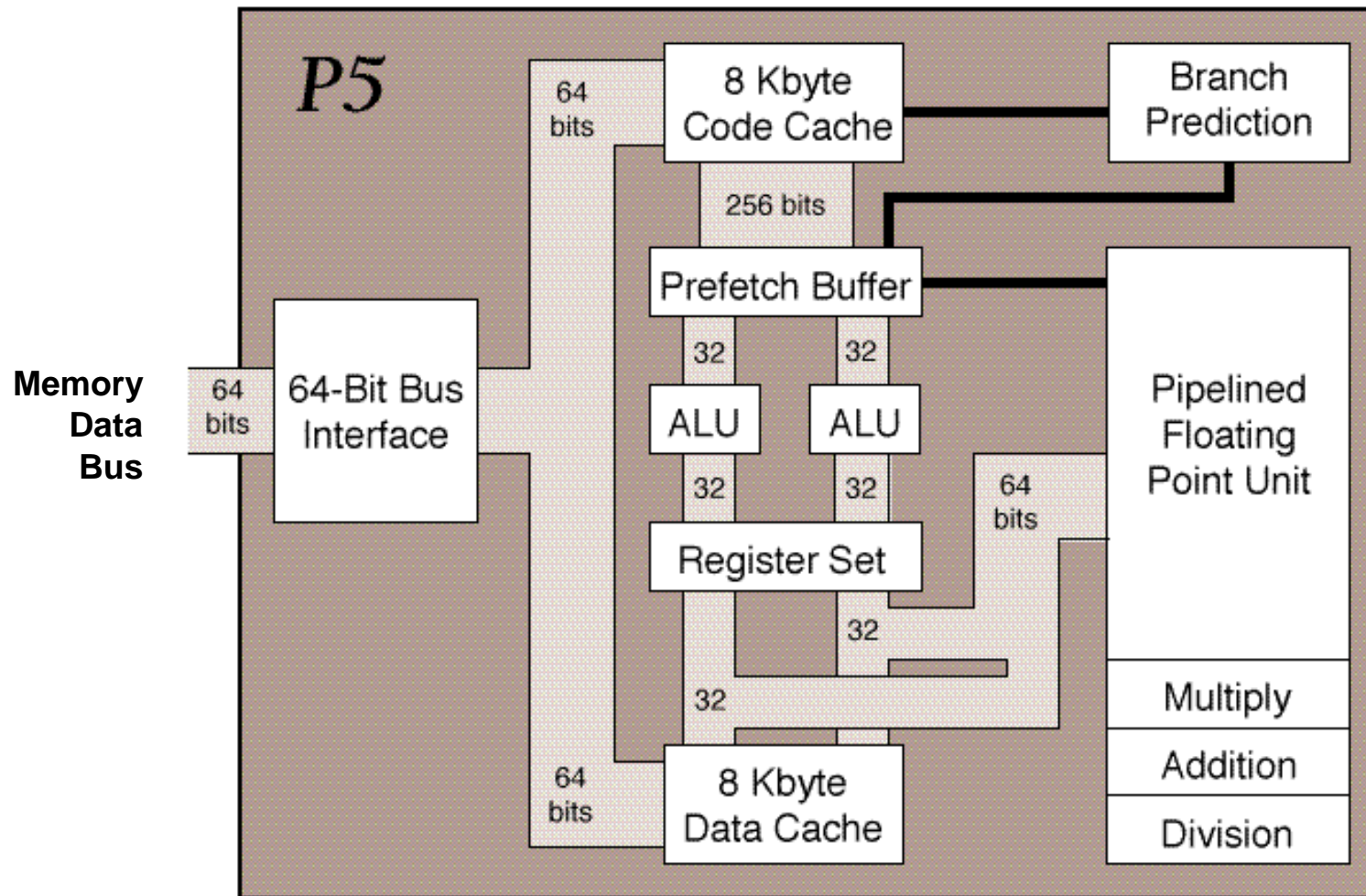
Cycles Per Instruction

Instruction Type	386 Cycles	486 Cycles
Load	4	1
Store	2	1
ALU	2	1
Jump taken	9	3
Jump not taken	3	1
Call	9	3

Reasons for Improvement

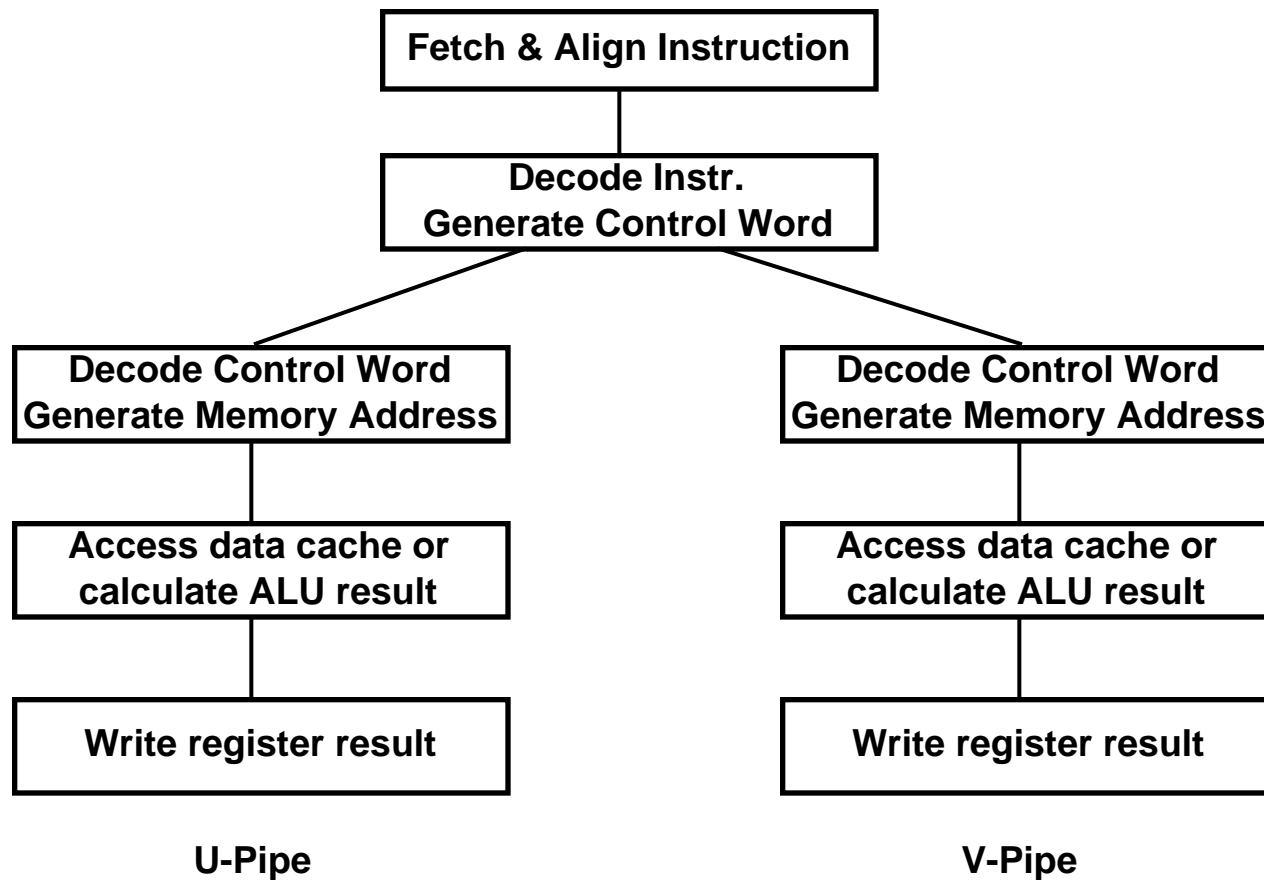
- On chip cache
 - Faster loads & stores
- More pipelining

Pentium Block Diagram



(Microprocessor Report 10/28/92)

Pentium Pipeline



Superscalar Execution

Can Execute Instructions I1 & I2 in Parallel if:

- **Both are “simple” instructions**
 - Don’t require microcode sequencing
 - Some operations require U-pipe resources
 - 90% of SpecInt instructions
- **I1 is not a jump**
- **Destination of I1 not source of I2**
 - But can handle I1 setting CC and I2 being cond. jump
- **Destination of I1 not destination of I2**

If Conditions Don’t Hold

- **Issue I1 to U Pipe**
- **I2 issued on next cycle**
 - Possibly paired with following instruction

Branch Prediction

Branch Target Buffer

- **Stores information about previously executed branches**
 - Indexed by instruction address
 - Specifies branch destination + whether or not taken
- **256 entries**

Branch Processing

- **Look for instruction in BTB**
- **If found, start fetching at destination**
- **Branch condition resolved early in WB**
 - If prediction correct, no branch penalty
 - If prediction incorrect, lose ~3 cycles
 - » Which corresponds to > 3 instructions
- **Update BTB**

Superscalar Terminology

Basic

Superscalar Able to issue > 1 instruction / cycle

Superpipelined Deep, but not superscalar pipeline.
E.g., MIPS R5000 has 8 stages

Branch prediction Logic to guess whether or not branch will be taken,
and possibly branch target

Advanced

Out-of-order Able to issue instructions out of program order

Speculation Execute instructions beyond branch points, possibly
nullifying later

Register renaming Able to dynamically assign physical registers to
instructions

Retire unit Logic to keep track of instructions as they complete.

Superscalar Execution Example

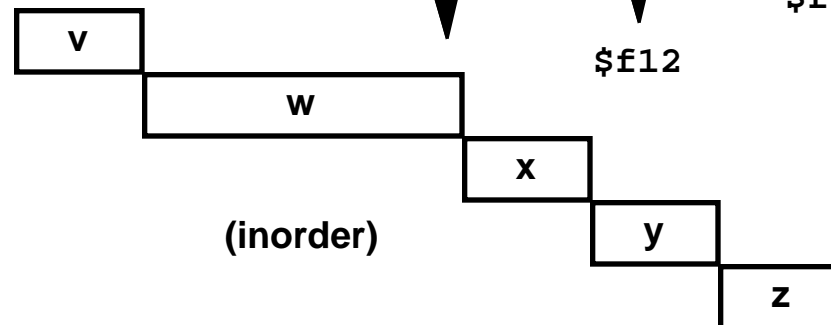
Assumptions

- Single FP adder takes 2 cycles
- Single FP multiplier takes 5 cycles
- Can issue add & multiply together
- Must issue in-order

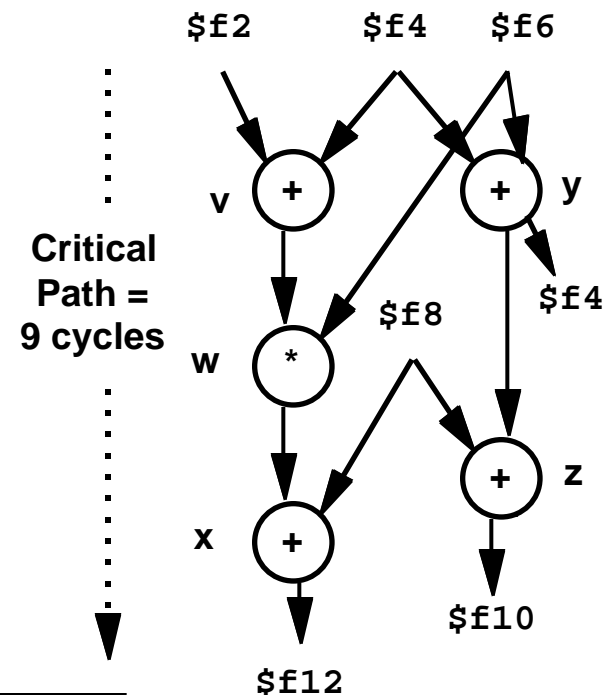
(Single adder, data dependence)
(In order)

```

v:  addt  $f2, $f4, $f10
w:  mult  $f10, $f6, $f10
x:  addt  $f10, $f8, $f12
y:  addt  $f4, $f6, $f4
z:  addt  $f4, $f8, $f10
    
```



Data Flow

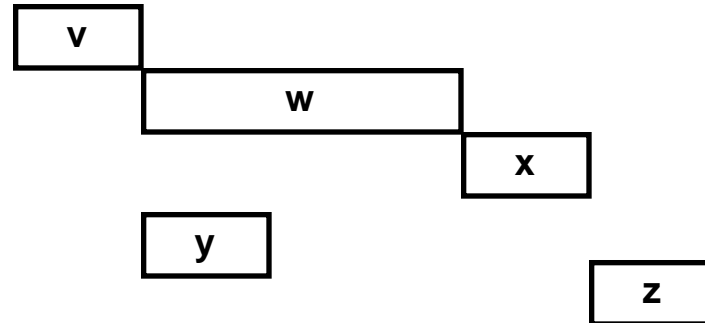


Adding Advanced Features

Out Of Order Issue

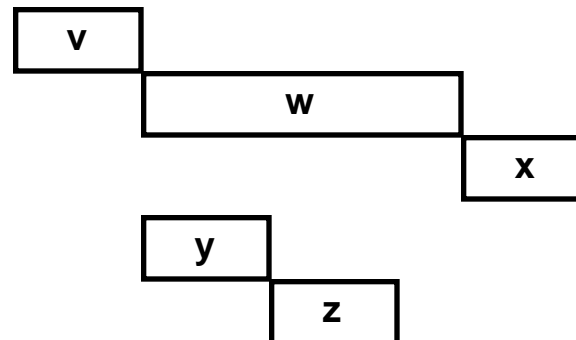
- Can start y as soon as adder available
- Must hold back z until \$f10 not busy & adder available

```
v:  addt  $f2, $f4, $f10
w:  mult  $f10, $f6, $f10
x:  addt  $f10, $f8, $f12
y:  addt  $f4, $f6,  $f4
z:  addt  $f4, $f8, $f10
```



With Register Renaming

```
v:  addt  $f2, $f4, $f10a
w:  mult  $f10a, $f6, $f10a
x:  addt  $f10a, $f8, $f12
y:  addt  $f4, $f6,  $f4
z:  addt  $f4, $f8, $f10
```



Pentium Pro (P6)

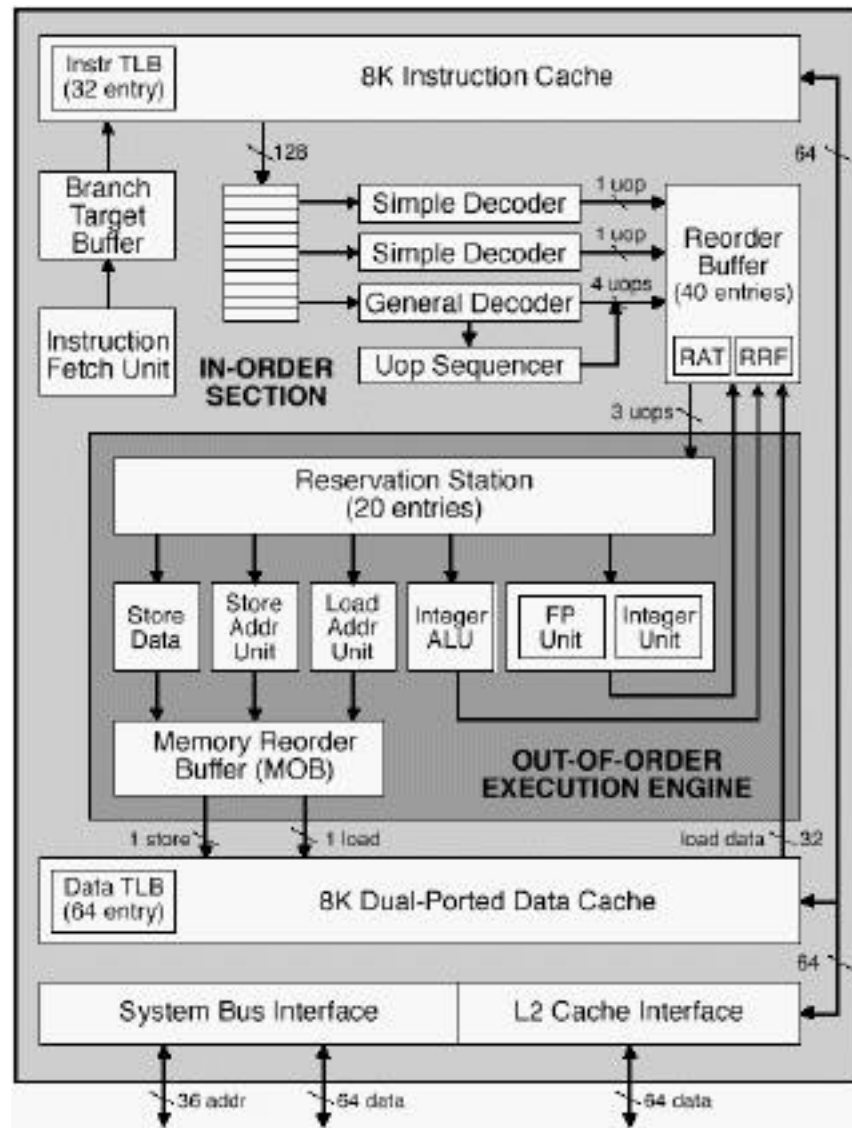
History

- **Announced in Feb. '95**
- **Delivering in high end machines now**

Features

- **Dynamically translates instructions to more regular format**
 - Very wide RISC instructions
- **Executes operations in parallel**
 - Up to 5 at once
- **Very deep pipeline**
 - 12–18 cycle latency

PentiumPro Block Diagram



Microprocessor Report
2/16/95

PentiumPro Operation

Translates instructions dynamically into “Uops”

- 118 bits wide
- Holds operation, two sources, and destination

Executes Uops with “Out of Order” engine

- Uop executed when
 - Operands available
 - Functional unit available
- Execution controlled by “Reservation Stations”
 - Keeps track of data dependencies between uops
 - Allocates resources

Branch Prediction

Critical to Performance

- 11–15 cycle penalty for misprediction

Branch Target Buffer

- 512 entries
- 4 bits of history
- Adaptive algorithm
 - Can recognize repeated patterns, e.g., alternating taken–not taken

Handling BTB misses

- Detect in cycle 6
- Predict taken for negative offset, not taken for positive
 - Loops vs. conditionals

Limitations of x86 Instruction Set

Not enough registers

- too many memory references

Intel is switching to a new instruction set for Merced

- IA-64, joint with HP
- Will dynamically translate existing x86 binaries

PPC 604

Superscalar

- Up to 4 instructions per cycle

Speculative & Out-of-Order Execution

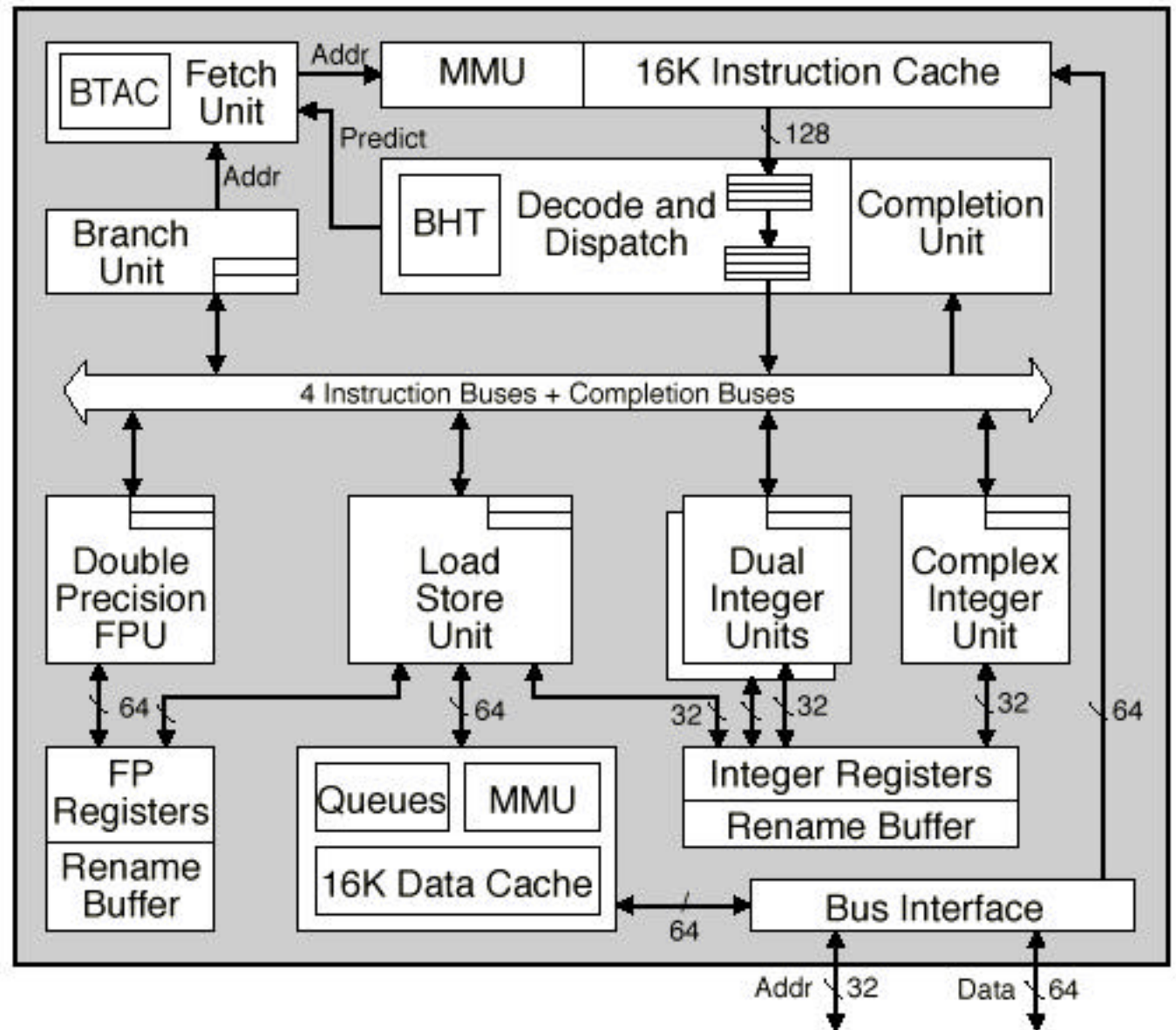
- Begin issuing and executing instructions beyond branch

Other Processors in this Category

- MIPS R10000
- Intel PentiumPro & Pentium II
- Digital Alpha 21264

604 Block Diagram

Microprocessor
Report
April 18, 1994



General Principles

Must be Able to Flush Partially-Executed Instructions

- Branch mispredictions
- Earlier instruction generates exception

Special Treatment of “Architectural State”

- Programmer-visible registers
- Memory locations
- Don’t do actual update until certain instruction should be executed

Emulate “Data Flow” Execution Model

- Instruction can execute whenever operands available

Processing Stages

Fetch

- Get instruction from instruction cache

Dispatch (~= Decode)

- Get available operands
- Assign to hardware execution unit

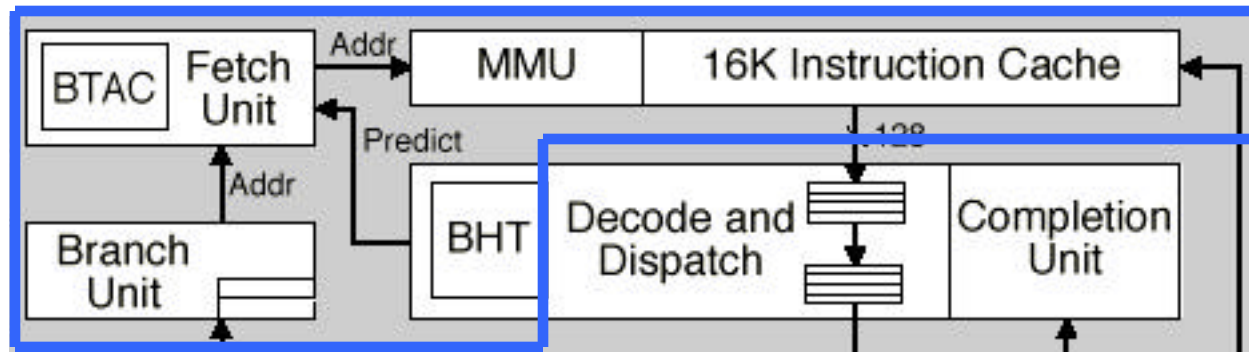
Execute

- Perform computation or memory operation
 - Store's are only buffered

Retire / Commit (~= Writeback)

- Allow architectural state to be updated
 - Register update
 - Buffered store

Fetching Instructions



- Up to 4 fetched from instruction cache in single cycle

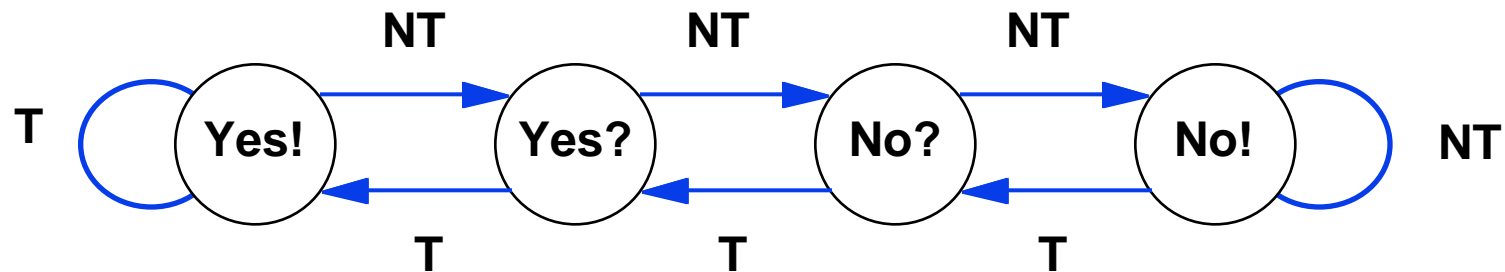
Branch Target Address Cache (BTAC)

- Target addresses of recently-executed, predicted-taken branches
 - 64 entries
 - Indexed by instruction address
- Accessed in parallel with instruction fetch
- If hit, fetch at predicted target starting next cycle

Branch Prediction

Branch History Table (BHT)

- 512 state machines, indexed by low-order bits of instruction address
- Encode information about prior history of branch instructions
 - Small chance of two branch instructions aliasing
- Predict whether or not branch will be taken
 - 3 cycle penalty if mispredict



Interaction with BTAC

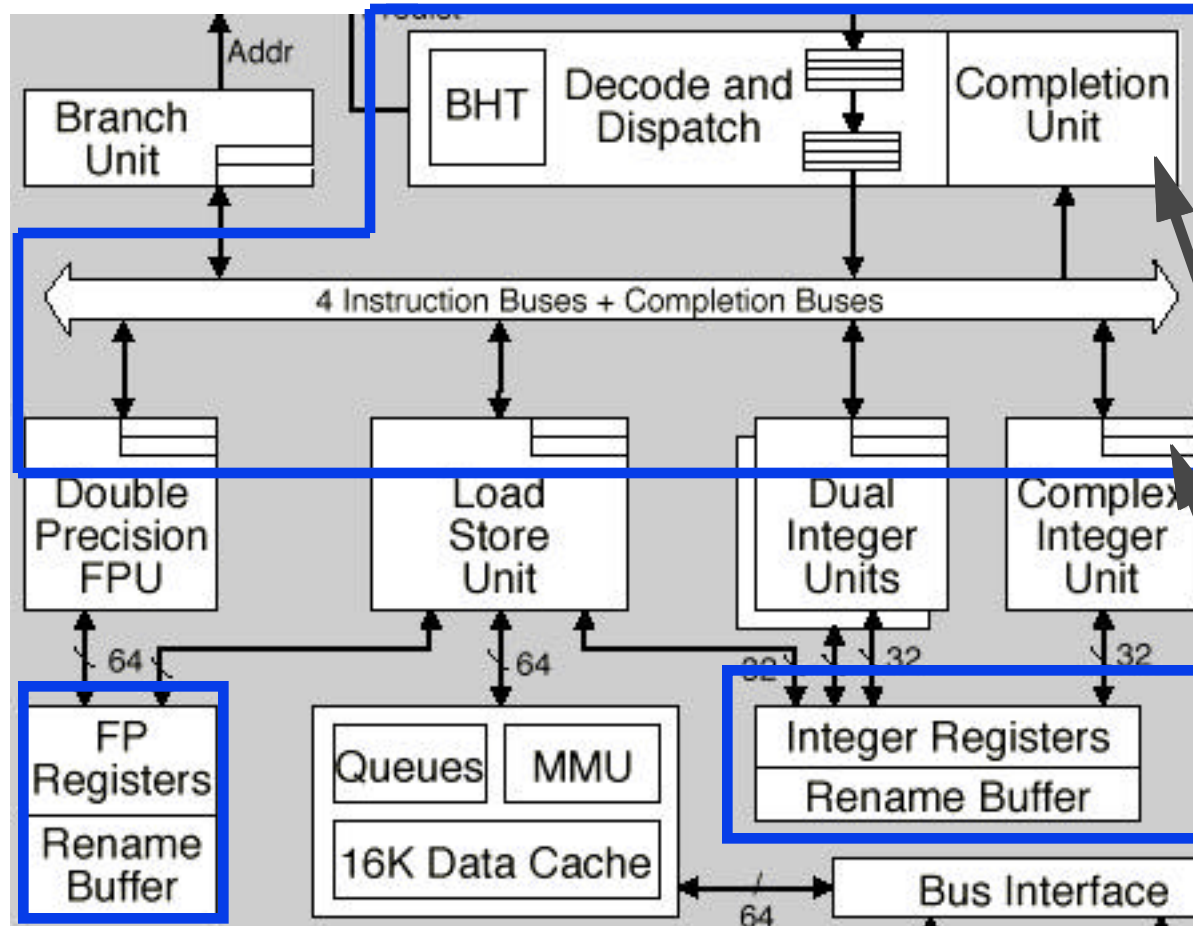
- BHT entries start in state No!
- When make transition from No? to Yes?, allocate entry in BTAC
- Deallocate when make transition from Yes? to No?

Dispatch

- **Up to 4 instructions per cycle**
 - Assign to execution units
 - Put entry in retirement buffer
 - Assign rename registers
- **Ignore data dependencies**

Retirement Buffer

“Reservation Stations”



Dispatching Actions

Generate Entry in Retirement Buffer

- 16-entry buffer tracking instructions currently “in flight”
 - Dispatched but not yet completed
- Circular buffer in program order
- Instruction tagged with branches they depend on
 - Easy to flush if mispredicted

Assign Rename Register as Target

- Additional registers (12 integer, 8 FP) used as targets for in-flight instructions
- Instruction updates this register
- Update of actual architectural register occurs only when instruction retired

Hazard Handling with Renaming

Dispatch Unit Maintains Mapping

- From register ID to actual register
- Could be the actual architectural register
 - Not target of currently-executing instruction
- Could be rename register
 - Perhaps already written by instruction that has not been retired
 - » E.g., still waiting for confirmation of branch prediction
 - Perhaps instruction result not yet computed
 - » Grab later when available

Hazards

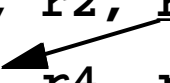
- **RAW:** Mapping identifies operand source
- **WAR:** Write will be to different rename register
- **WAW:** Writes will be to different rename register

Read-after-Write (RAW) Dependences

Also known as a “true” dependence

Example:

```
S1:      addq r1, r2, r3
S2:      addq r3, r4, r4
```



How to optimize?

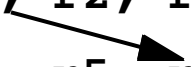
- cannot be optimized away

Write-after-Read (WAR) Dependences

Also known as an “anti” dependence

Example:

```
S1:    addq r1, r2, r3
S2:    addq r4, r5, r1
      ...
      addq r1, r6, r7
```



How to optimize?

- rename dependent register (e.g., r1 in S2 -> r8)

```
S1:    addq r1, r2, r3
S2:    addq r4, r5, r8
      ...
      addq r8, r6, r7
```

Write-after-Write (WAW) Dependencies

Also known as an “output” dependence

Example:

```
S1:      addq r1, r2, r3
          ↓
S2:      addq r4, r5, r3
          ...
          addq r3, r6, r7
```

How to optimize?

- rename dependent register (e.g., r3 in S2 -> r8)

```
S1:      addq r1, r2, r3
S2:      addq r4, r5, r8
          ...
          addq r8, r6, r7
```

Moving Instructions Around

Reservation Stations

- **Buffers associated with execution units**
- **Hold instructions prior to execution**
 - Plus those operands that are available
- **May be waiting for one or more operands**
 - Operand mapped to rename register that is not yet available
- **May be waiting for unit to be available**

Completion Busses

- **Results generated by execution units**
- **Tagged by rename register ID**
- **Monitored by reservation stations**
 - So they can get needed operands
 - Effectively implements bypassing
- **Supply results to completion unit**

Execution Resources

Integer

- **Two units to handle regular integer instructions**
- **One for “complex” operations**
 - Multiply with latency 3--4 and throughput once per 1--2 cycles
 - Unpipelined divide with latency 20

Floating Point

- **Add/multiply with latency 3 and throughput 1**
- **Unpipelined divide with latency 18--31**

Load Store Unit

- **Own address ALU**
- **Buffer of pending store instructions**
 - Don't perform actual store until ready to retire instruction
- **Loads can be performed speculatively**
 - Check to see if target of pending store operation

Retiring Instructions

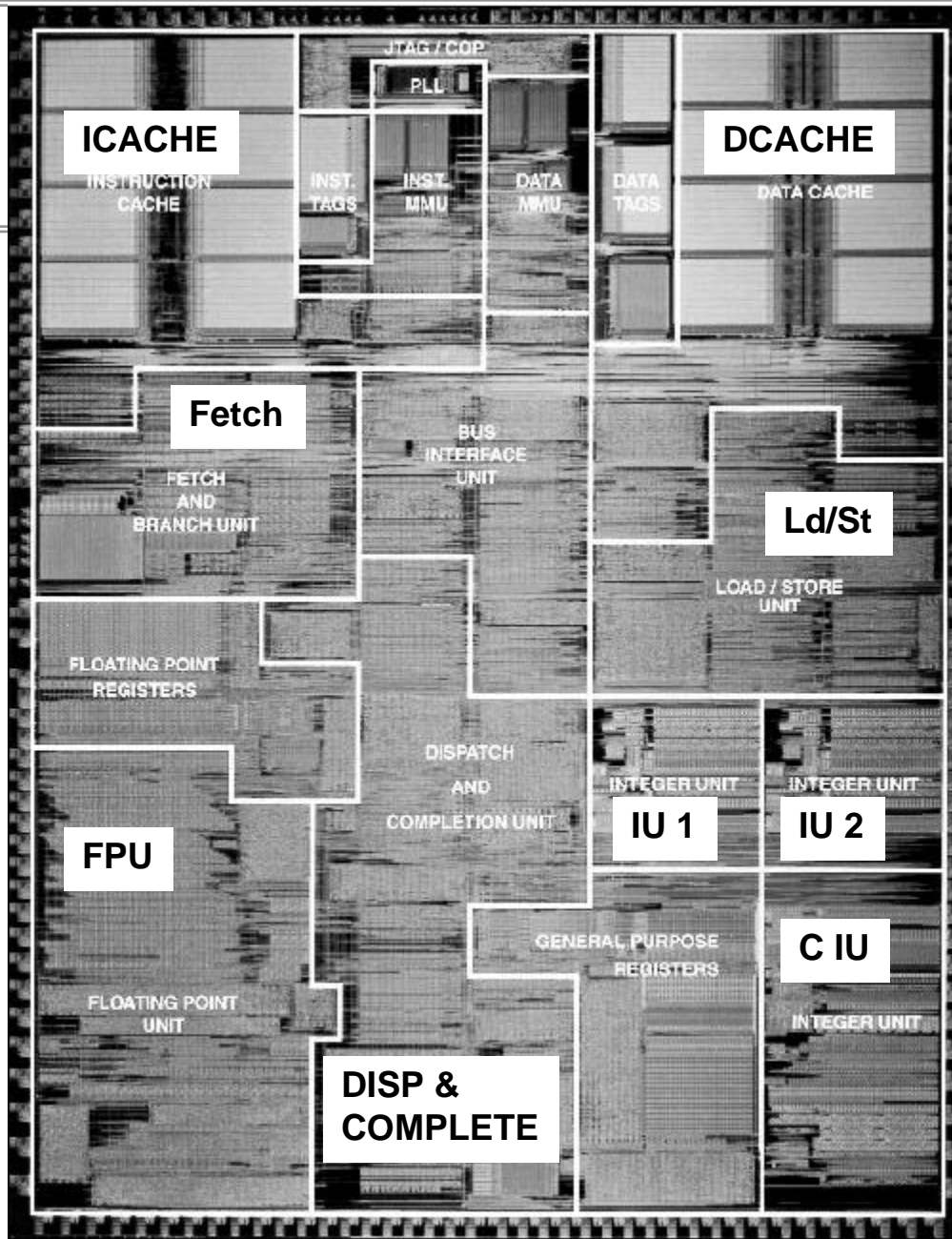
Retire in Program Order

- When instruction is at head of buffer
- Up to 4 per cycle
- Enable change of architectural state
 - Transfer from rename register to architectural
 - » Free rename register for use by another instruction
 - Allow pending store operation to take place

Flush if Should not be Executed

- Tagged by branch that was mispredicted
- Follows instruction that raised exception
- As if instructions had never been fetched

604 Chip



- **Originally 200 mm²**
 - 0.65μm process
 - 100 MHz
- **Now 148 mm²**
 - 0.35μm process
 - bigger caches
 - 300 MHz
- **Performance requires real estate**
 - 11% for dispatch & completion units
 - 6 % for register files
 - » Lots of ports

Figure 3. The PowerPC 604 incorporates 3.6 million transistors on a 12.4 × 15.8 mm die using 0.65-micron, five-layer-metal CMOS.

Execution Example

Assumptions

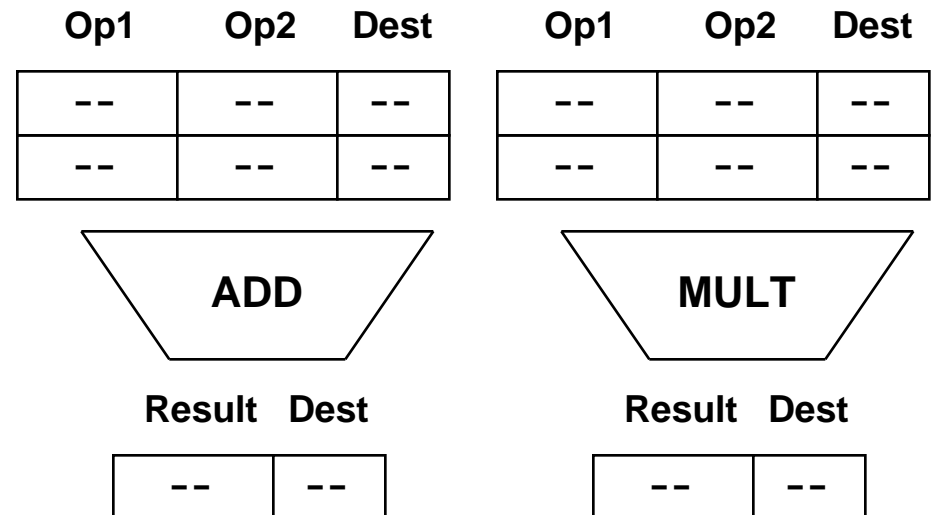
- Two-way issue with renaming
 - Rename registers %f0, %f2, etc.
- 1 cycle add.d latency, 2 cycle mult.d

	Value	Rename
\$f2	10.0	\$f2
\$f4	20.0	\$f4
\$f6	40.0	\$f6
\$f8	80.0	\$f8
\$f10	160.0	\$f10
\$f12	320.0	\$f12

	Value	Renames	Valid
%f0	--	--	F
%f2	--	--	F
%f4	--	--	F
%f6	--	--	F

```

v:  addt  $f2, $f4, $f10
w:  mult  $f10, $f6, $f10
x:  addt  $f10, $f8, $f12
y:  addt  $f4, $f6,  $f4
z:  addt  $f4, $f8, $f10
    
```



Execution Example Cycle 1

Actions

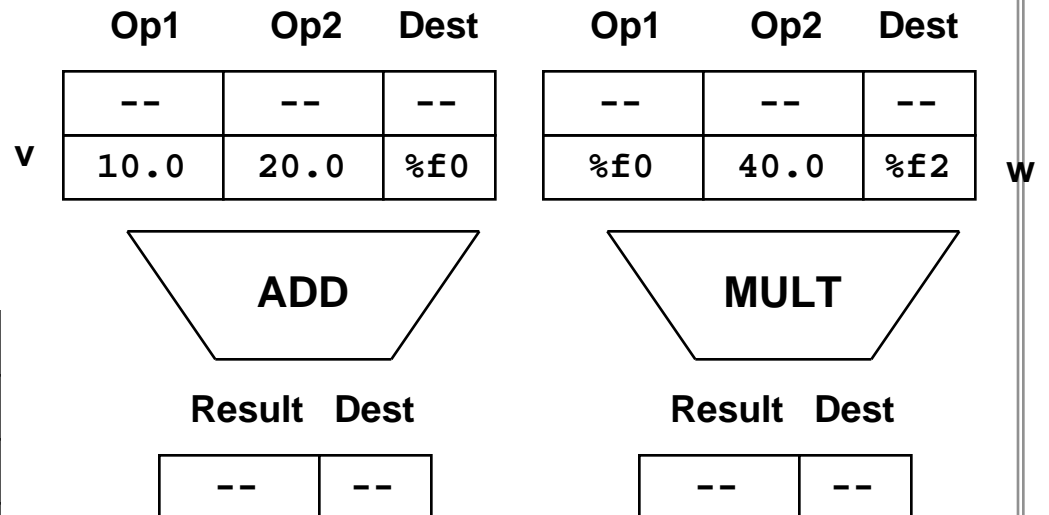
- Instructions v & w issued
 - v target set to %f0
 - w target set to %f2

	Value	Rename
\$f2	10.0	\$f2
\$f4	20.0	\$f4
\$f6	40.0	\$f6
\$f8	80.0	\$f8
\$f10	160.0	%f2
\$f12	320.0	\$f12

	Value	Renames	Valid
%f0	--	\$f10	F
%f2	--	\$f10	F
%f4	--	--	F
%f6	--	--	F

```

v:  addt  $f2, $f4, $f10
w:  mult  $f10, $f6, $f10
x:  addt  $f10, $f8, $f12
y:  addt  $f4, $f6,  $f4
z:  addt  $f4, $f8, $f10
    
```



Execution Example Cycle 2

Actions

- Instructions x & y issued
 - x & y targets set to %f4 and %f6
- Instruction v executed

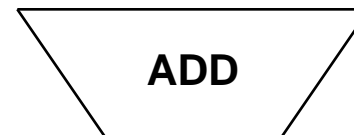
```

v:  addt  $f2, $f4, $f10
w:  mult  $f10, $f6, $f10
x:  addt  $f10, $f8, $f12
y:  addt  $f4, $f6,  $f4
z:  addt  $f4, $f8, $f10
    
```

	Value	Rename
\$f2	10.0	\$f2
\$f4	20.0	%f6
\$f6	40.0	\$f6
\$f8	80.0	\$f8
\$f10	160.0	%f2
\$f12	320.0	%f4

	Value	Renames	Valid
%f0	30.0	\$f10	T
%f2	--	\$f10	F
%f4	--	\$f12	F
%f6	--	\$f4	F

	Op1	Op2	Dest		Op1	Op2	Dest	
y	20.0	40.0	%f6		--	--	--	
x	%f2	80.0	%f4		30.0	40.0	%f2	w



Result Dest

30.0	%f0	v
------	-----	---



Result Dest

--	--	
----	----	--

Cycle 3

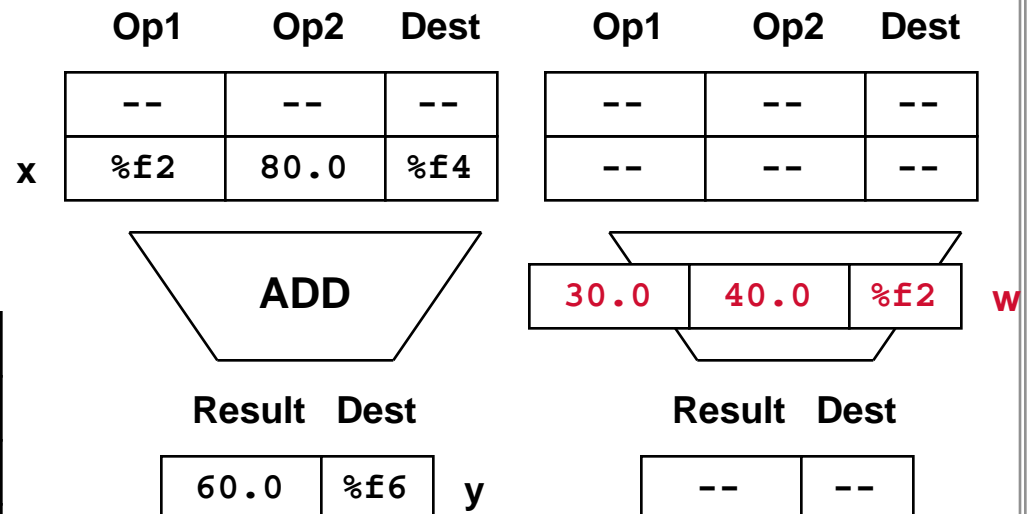
- **Instruction v retired**
 - But doesn't change \$f10
- **Instruction w begins execution**
 - Moves through 2 stage pipeline
- **Instruction y executed**
- **Instruction z stalled**
 - Not enough reservation stations

```

v:  addt  $f2, $f4, $f10
w:  mult  $f10, $f6, $f10
x:  addt  $f10, $f8, $f12
y:  addt  $f4, $f6,  $f4
z:  addt  $f4, $f8, $f10
  
```

	Value	Rename
\$f2	10.0	\$f2
\$f4	20.0	%f6
\$f6	40.0	\$f6
\$f8	80.0	\$f8
\$f10	160.0	%f2
\$f12	320.0	%f4

	Value	Renames	Valid
%f0	--	--	F
%f2	--	\$f10	F
%f4	--	\$f12	F
%f6	60.0	\$f4	T



Execution Example Cycle 4

- Instruction w finishes execution
- Instruction y cannot be retired yet
- Instruction z issued
 - Assigned to %f0

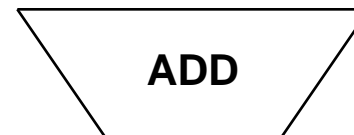
```

v:  addt  $f2, $f4, $f10
w:  mult  $f10, $f6, $f10
x:  addt  $f10, $f8, $f12
y:  addt  $f4, $f6,  $f4
z:  addt  $f4, $f8, $f10
    
```

	Value	Rename
\$f2	10.0	\$f2
\$f4	20.0	%f6
\$f6	40.0	\$f6
\$f8	80.0	\$f8
\$f10	160.0	%f0
\$f12	320.0	%f4

	Value	Renames	Valid
%f0	--	\$f10	F
%f2	120.0	\$f10	T
%f4	--	\$f12	F
%f6	60	\$f4	T

	Op1	Op2	Dest
z	60.0	80.0	%f0
x	120.0	80.0	%f4



Result Dest

--	--
----	----

	Op1	Op2	Dest
	--	--	--
	--	--	--



Result Dest

120.0	%f2	w
-------	-----	---

Execution Example Cycle 5

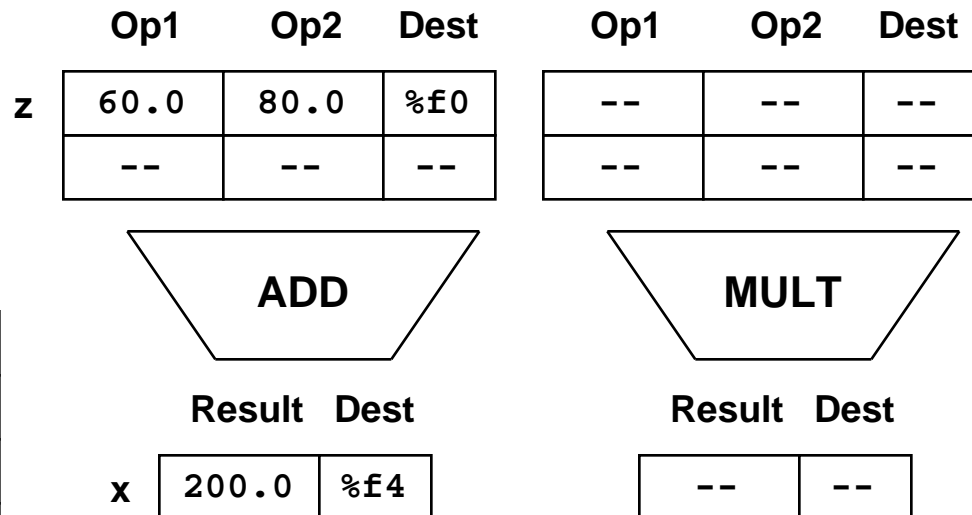
- Instruction w retired
 - But does not change \$f10
- Instruction y cannot be retired yet
- Instruction x executed

```

v:  addt  $f2, $f4, $f10
w:  mult  $f10, $f6, $f10
x:  addt  $f10, $f8, $f12
y:  addt  $f4, $f6,  $f4
z:  addt  $f4, $f8, $f10
    
```

	Value	Rename
\$f2	10.0	\$f2
\$f4	20.0	%f6
\$f6	40.0	\$f6
\$f8	80.0	\$f8
\$f10	160.0	%f0
\$f12	320.0	%f4

	Value	Renames	Valid
%f0	--	\$f10	F
%f2	--	--	F
%f4	200.0	\$f12	T
%f6	60	\$f4	T



Execution Example Cycle 6

- Instruction x & y retired
 - Update \$f12 and \$f4
- Instruction z executed

	Value	Rename
\$f2	10.0	\$f2
\$f4	60.0	\$f4
\$f6	40.0	\$f6
\$f8	80.0	\$f8
\$f10	160.0	%f0
\$f12	200.0	\$f12

	Value	Renames	Valid
%f0	140.0	\$f10	T
%f2	--	--	F
%f4	--	--	F
%f6	--	--	F

```

v:  addt  $f2, $f4, $f10
w:  mult  $f10, $f6, $f10
x:  addt  $f10, $f8, $f12
y:  addt  $f4, $f6,  $f4
z:  addt  $f4, $f8, $f10
    
```

Op1	Op2	Dest	Op1	Op2	Dest
--	--	--	--	--	--
--	--	--	--	--	--



Result Dest

z	140.0	%f0
---	-------	-----



Result Dest

--	--
----	----

Execution Example Cycle 7

- Instruction z retired

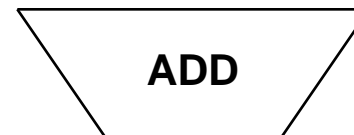
	Value	Rename
\$f2	10.0	\$f2
\$f4	60.0	\$f4
\$f6	40.0	\$f6
\$f8	80.0	\$f8
\$f10	140.0	\$f10
\$f12	320.0	\$f12

	Value	Renames	Valid
%f0	--	--	F
%f2	--	--	F
%f4	--	--	F
%f6	--	--	F

```

v:  addt  $f2, $f4, $f10
w:  mult  $f10, $f6, $f10
x:  addt  $f10, $f8, $f12
y:  addt  $f4, $f6,  $f4
z:  addt  $f4, $f8, $f10
    
```

Op1	Op2	Dest	Op1	Op2	Dest
--	--	--	--	--	--
--	--	--	--	--	--



Result Dest

--	--
----	----



Result Dest

--	--
----	----

Living with Expensive Branches

Mispredicted Branch Carries a High Cost

- **Must flush many in-flight instructions**
- **Start fetching at correct target**
- **Will get worse with deeper and wider pipelines**

Impact on Programmer / Compiler

- **Avoid conditionals when possible**
 - Bit manipulation tricks
- **Use special conditional-move instructions**
 - Recent additions to many instruction sets
- **Make branches predictable**
 - Very low overhead when predicted correctly

Branch Prediction Example

```
static void loop1() {
    int i;
    data_t abs_sum = (data_t) 0;
    data_t prod = (data_t) 1;
    for (i = 0; i < CNT; i++) {
        data_t x = dat[i];
        data_t ax;
        ax = ABS(x);
        abs_sum += ax;
        prod *= x;
    }
    answer = abs_sum+prod;
}
```

```
#define ABS(x) x < 0 ? -x : x
```

MIPS Code

0x6c4:	8c620000	lw	r2,0(r3)
0x6c8:	24840001	addiu	r4,r4,1
0x6cc:	04410002	bgez	r2,0x6d8
0x6d0:	00a20018	mult	r5,r2
0x6d4:	00021023	subu	r2,r0,r2
0x6d8:	00002812	mflo	r5
0x6dc:	00c23021	addu	r6,r6,r2
0x6e0:	28820400	slti	r2,r4,1024
0x6e4:	1440fff7	bne	r2,r0,0x6c4
0x6e8:	24630004	addiu	r3,r3,4

- Compute sum of absolute values
- Compute product of original values

Some Interesting Patterns

PPPPPPPPP

+1 ...

- **Should give perfect prediction**

RRRRRRRRR

-1 -1 +1 +1 +1 +1 -1 +1 -1 -1 +1 +1 -1 -1 +1 +1 +1 +1 +1 -1 -1 -1 +1 -1 ...

- **Will mispredict 1/2 of the time**

N*N[PNPN]

-1 -1 -1 -1 -1 -1 -1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 ...

- **Should alternate between states No! and No?**

N*P[PNPN]

-1 -1 -1 -1 -1 -1 -1 +1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 ...

- **Should alternate between states No? and Yes?**

N*N[PPNN]

-1 -1 -1 -1 -1 -1 -1 -1 +1 +1 -1 -1 +1 +1 -1 -1 +1 +1 -1 -1 +1 +1 -1 -1 ...

N*P[PPNN]

-1 -1 -1 -1 -1 -1 -1 +1 +1 +1 -1 -1 +1 +1 -1 -1 +1 +1 -1 -1 +1 +1 -1 -1 ...

Loop Performance (FP)

	R3000		PPC 604		Pentium	
Pattern	Cycles	Penalty	Cycles	Penalty	Cycles	Penalty
PPPPPPPPP	13.6	0	9.2	0	21.1	0
RRRRRRRRR	13.6	0	12.6	3.4	22.9	1.8
N*N[PNPN]	13.6	0	15.8	6.6	23.3	2.2
N*P[PNPN]	13.3	-0.3	15.9	6.7	24.3	3.2
N*N[PPNN]	13.3	-0.3	12.5	3.3	23.9	2.8
N*P[PPNN]	13.6	0	12.5	3.3	24.7	3.6

Observations

- **604 has prediction rates 0%, 50%, and 100%**
 - Expected 50% from N*N[PNPN]
 - Expected 25% from N*N[PPNN]
 - Loop so tight that speculate through single branch twice?
- **Pentium appears to be more variable, ranging 0 to 100%**

Special Patterns Can be Worse than Random

- Only 50% of all people are “above average”

Loop 1 Surprises

Pentium II

	R10000		Pentium II	
Pattern	Cycles	Penalty	Cycles	Penalty
PPPPPPPPP	3.5	0	11.9	0
RRRRRRRRR	3.5	0	19	7.1
N*N[PNPN]	3.5	0	12.5	0.6
N*P[PNPN]	3.5	0	13	1.1
N*N[PPNN]	3.5	0	12.4	0.5
N*P[PPNN]	3.5	0	12.2	0.3

- Random shows clear penalty
- But others do well
 - More clever prediction algorithm

R10000

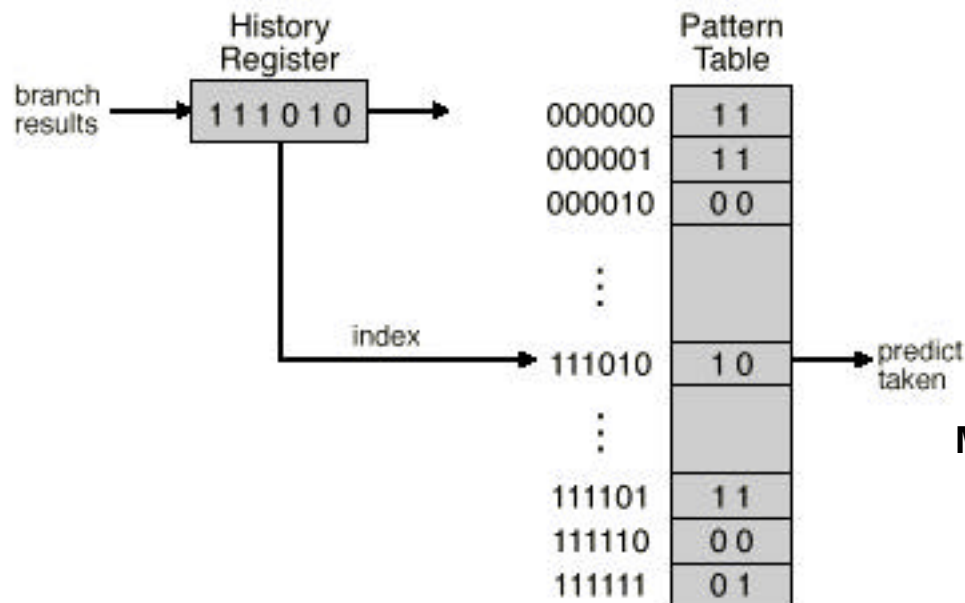
- Has special “conditional move” instructions
- Compiler translates $a = Cond ? Texpr : Fexpr$ into


```

a = Fexpr
temp = Texpr
CMOV(a, temp, Cond)

```
- Only valid if *Texpr* & *Fexpr* can't cause error

P6 Branch Prediction

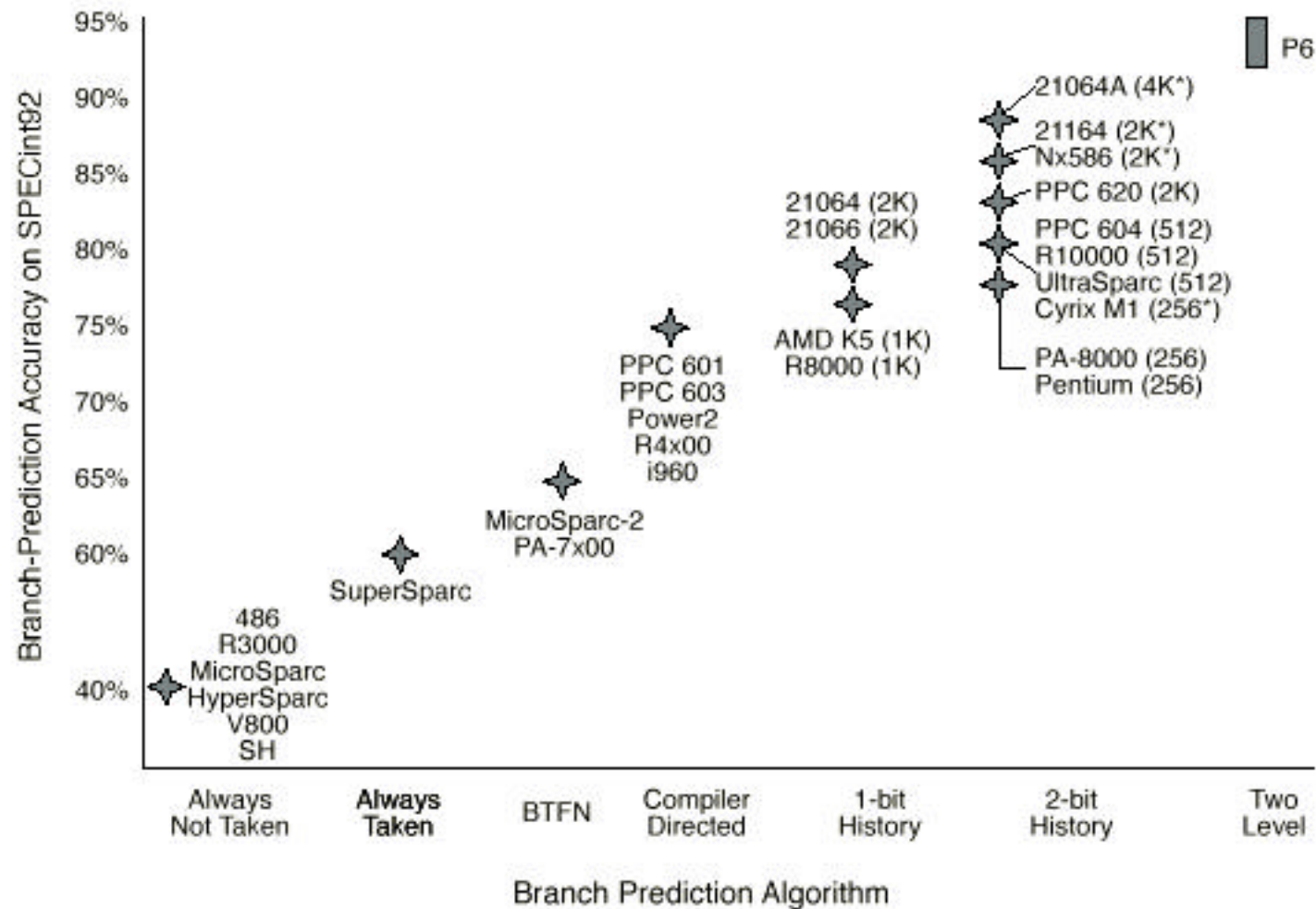


Microprocessor Report
March 27, 1995

Two-Level Scheme

- Yeh & Patt, ISCA '93
- Keep shift register showing past k outcomes for branch
- Use to index 2^k entry table
- Each entry provides 2-bit, saturating counter predictor
- Very effective for any deterministic branching pattern

Branch Prediction Comparisons



Microprocessor Report March 27, 1995

Effect of Loop Unrolling

	PPC 604e	1X	PPC 604e	2X
Pattern	Cycles	Penalty	Cycles	Penalty
PPPPPPPPP	9.2	0	7.7	0
RRRRRRRRR	12.6	3.4	11.3	3.6
N*N[PNPN]	15.8	6.6	7.6	0
N*P[PNPN]	15.9	6.7	7.7	0
N*N[PPNN]	12.5	3.3	11.3	3.6
N*P[PPNN]	12.5	3.3	13.1	5.4

Observations

- [PNPN] yields PPPP ... for one branch, NNNN ... for the other
- [PPNN] yields PNPN ... for both branches
 - 50% accuracy if start in state No?
 - 25% accuracy if start in state No!

Another stressor in the life of a benchmarker

- Must look carefully at what compiler is doing

MIPS R10000

(See attached handouts.)

More info available at:

- `http://www.sgi.com/MIPS/products/r10k`

DEC Alpha 21264

Fastest Announced Processor

- Spec95: 30 Int 60 FP
- 500 MHz, 15M transistors, 60 Watts

Fastest Existing Processor is Alpha 21164

- Spec95: 12.6 Int 18.3 FP
- 500 MHz, 9.2M transistors, 25 Watts

Uses Every Trick in the Book

- 4–6 way superscalar
- Out of order execution with renaming
- Up to 80 instructions in process simultaneously
- Lots of cache & memory bandwidth

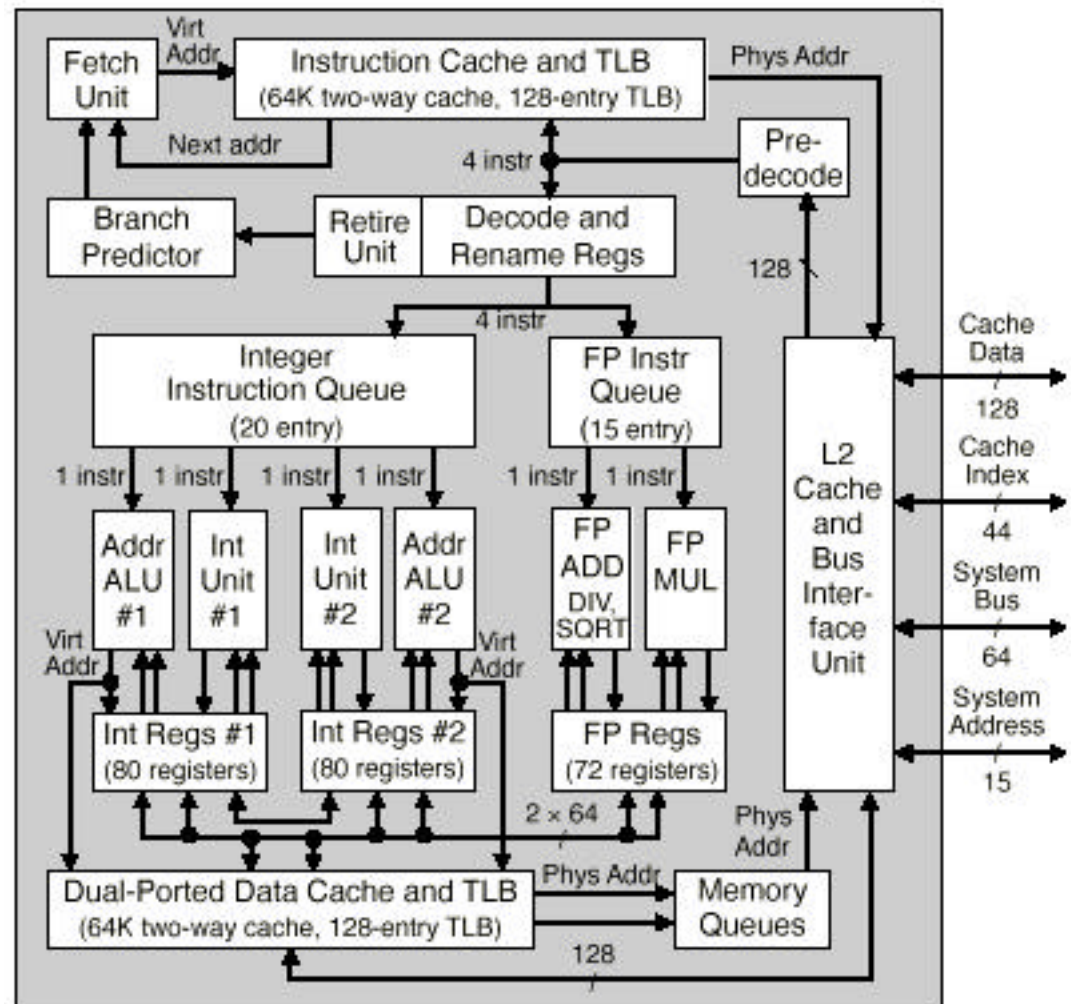
21264 Block Diagram

4 Integer ALUs

- Each can perform simple instructions
- 2 handle address calculations

Register Files

- 32 arch / 80 physical Int
- 32 arch / 72 physical FP
- Int registers duplicated
 - Extra cycle delay from write in one to read in other
 - Each has 6 read ports, 4 write ports
 - Attempt to issue consumer to producer side

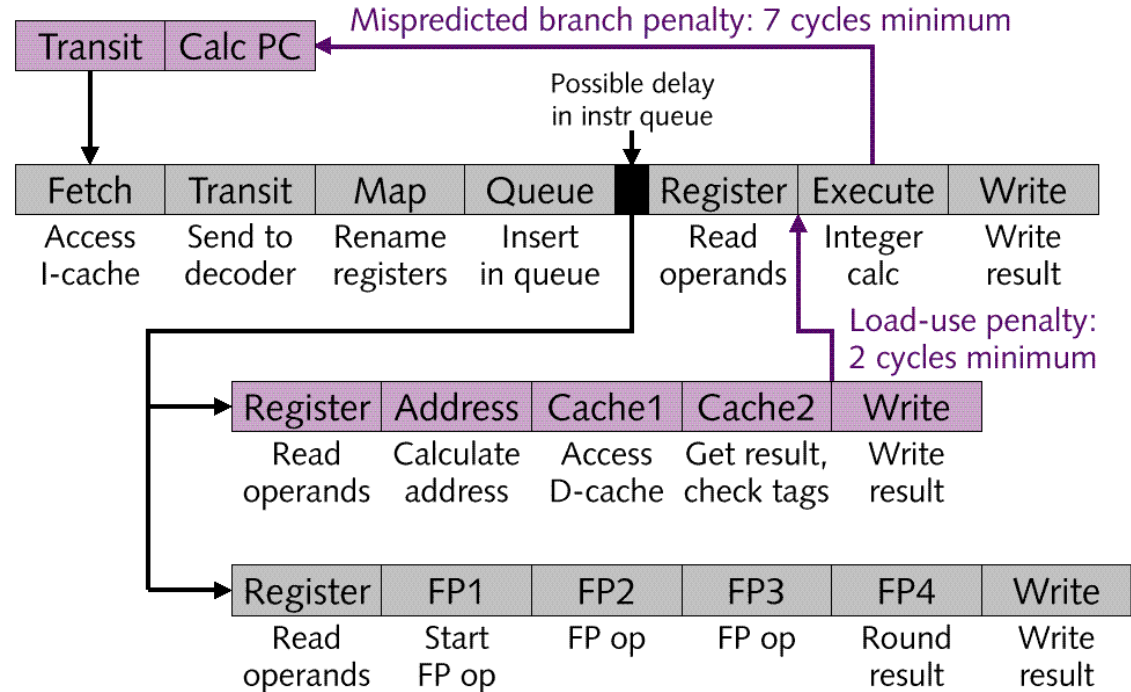


Microprocessor Report 10/28/96

21264 Pipeline

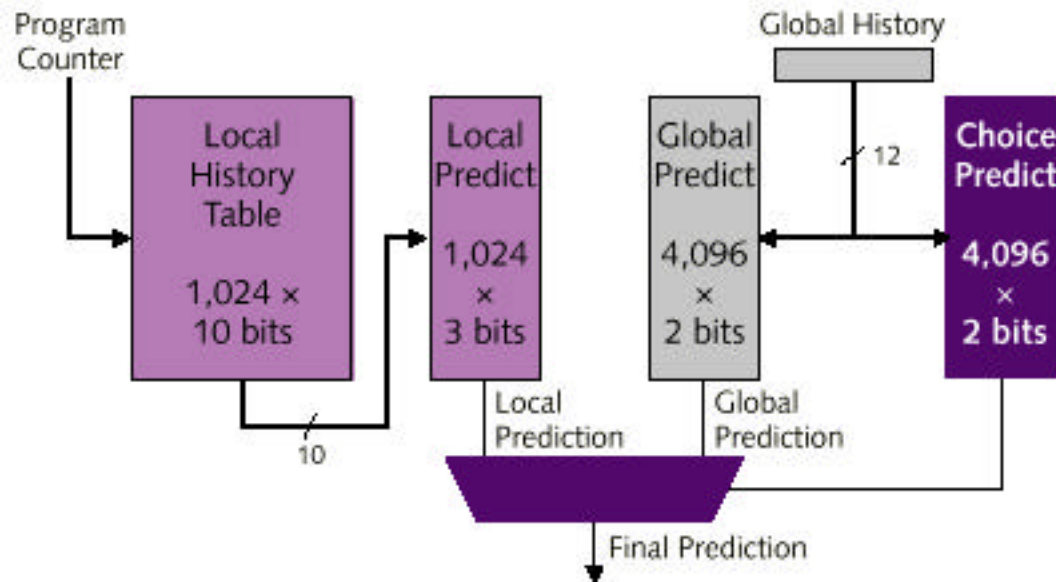
Very Deep Pipeline

- Can't do much in 2ns clock cycle!
- 7 cycles for simple instruction
- 9 cycles for load or store
- 7 cycle penalty for mispredicted branch
 - Elaborate branch predication logic
 - Claim 95% accuracy



Microprocessor Report 10/28/96

21264 Branch Prediction Logic



- **Purpose:** Predict whether or not branch taken
- **35Kb** of prediction information
- **2%** of total die size
- **Claim 0.7--1.0%** misprediction

Processor Comparisons

PROCESSORS FOR WORKSTATIONS AND SERVERS											
	Max Clock Speed	Cache Size	Supply Voltage	Max Power	Transistor Count	IC Process	Die Size	Est Mfg Cost*	SPEC95b int/fp	List Price	Availability
Digital 21164	500 MHz	8K/8K/96K	2.0 V	25 W	9.3 million	0.35 μ 4M	209 mm ²	\$150	12.6/18.3	\$1,450	now
Digital 21264	>500 MHz	64K/64K	2.0 V	60 W	15 million	0.35 μ 6M	300 mm ²	\$300	30/60	N.D.	4Q97
Fuj. TurboSparc	170 MHz	16K/16K	3.3 V	9 W	3.0 million	0.35 μ 4M	132 mm ²	\$50	3.5/3.0	\$499	now
HP PA-7300LC	160 MHz	64K/64K	3.3 V	15 W	9.2 million	0.5 μ 4M	259 mm ²	\$95	5.5/7.3	not sold	now
HP PA-8000	180 MHz	none	3.3 V	>40 W	3.9 million	0.5 μ 4M	345 mm ²	\$290	10.8/18.3	not sold	now
IBM P2SC	135 MHz	32K/128K	2.5 V	30 W	15 million	0.29 μ 4M	335 mm ²	\$375	5.5/14.5	not sold	now
MIPS R5000	180 MHz	32K/32K	3.3 V	10 W	3.6 million	0.35 μ 3M	84 mm ²	\$25	4.0/3.7	\$365	now
MIPS R7000	300 MHz	288K ⁽¹⁾	3.3 V	13 W	N.D.	0.25 μ 4M	80 mm ²	\$35	10/10	N.D.	2H97
MIPS R10000	200 MHz	32K/32K	3.3 V	30 W	5.9 million	0.35 μ 4M	298 mm ²	\$160	8.9/17.2	\$3,000	now
Sun UltraSparc-2	250 MHz	16K/16K	2.5 V	20 W	3.8 million	0.29 μ 5M	149 mm ²	\$90	8.5/15	\$1,995	limited
PROCESSORS FOR PCS AND WORKSTATIONS											
	Max Clock Speed	Cache Size	Supply Voltage	Max Power	Transistor Count	IC Process	Die Size	Est Mfg Cost*	SPEC95b int/fp	List Price	Availability
Exponential x704	533 MHz	2K/2K/32K	3.6 V	85 W	2.7 million	0.5 μ 5M ⁽²⁾	150 mm ²	\$90	12/10	\$1,000*	2Q97
PowerPC 603e	240 MHz	16K/16K	2.5 V	6 W	2.6 million	0.35 μ 4M	79 mm ²	\$30	5.5/4*	\$408	now
PowerPC 604e	225 MHz	32K/32K	2.5 V	24 W	5.1 million	0.35 μ 4M	148 mm ²	\$60	8/7*	\$533	now
Intel Pentium	200 MHz	8K/8K	3.3 V	17 W	3.3 million	0.35 μ 4M ⁽³⁾	90 mm ²	\$40	5.5/2.9	\$509	now
Intel P55C	200 MHz	16K/16K	2.8 V	16 W	4.5 million	0.28 μ 4M	140 mm ²	\$50	6/3*	N.D.	1Q97
Intel PPro	200 MHz	8K/8K	3.3 V	35 W†	5.5 million	0.35 μ 4M ⁽³⁾	196 mm ²	\$145†	8.2/6.0†	\$525†	now
Intel Klamath	266 MHz*	16K/16K	2.8 V*	N.D.	7.5 million	0.28 μ 4M	203 mm ²	\$80	11/7*	N.D.	2Q97*

Microprocessor Report 12/30/96

Challenges Ahead

Diminishing Returns on Cost vs. Performance

- Superscalar processors require instruction level parallelism
- Many programs limited by sequential dependencies

Finding New Sources of Parallelism

- e.g., thread-level parallelism

Getting Design Correct Difficult

- Verification team larger than design team
- Devise tests for interactions between concurrent instructions
 - May be 80 executing at once

New Era for Performance Optimization

Data Resources are Free and Fast

- Plenty of computational units
- Most programs have poor utilization

Unexpected Changes in Control Flow Expensive

- Kill everything downstream when mispredict
- Even if will execute in near future where branches reconverge

Think Parallel

- Try to get lots of things going at once

Not a Truly Parallel Machine

- Bounded resources
- Access from limited code window