Superscalar Processors
CS 740
Oct. 3, 1997

Overall Design

- Use PPC 604 as case study
- Speculative Execution
- Register Renaming

Branch Prediction

- Implementation
- Implications for programmers

PPC 604

Superscalar

- Up to 4 instructions per cycle

Speculative & Out-of-Order Execution

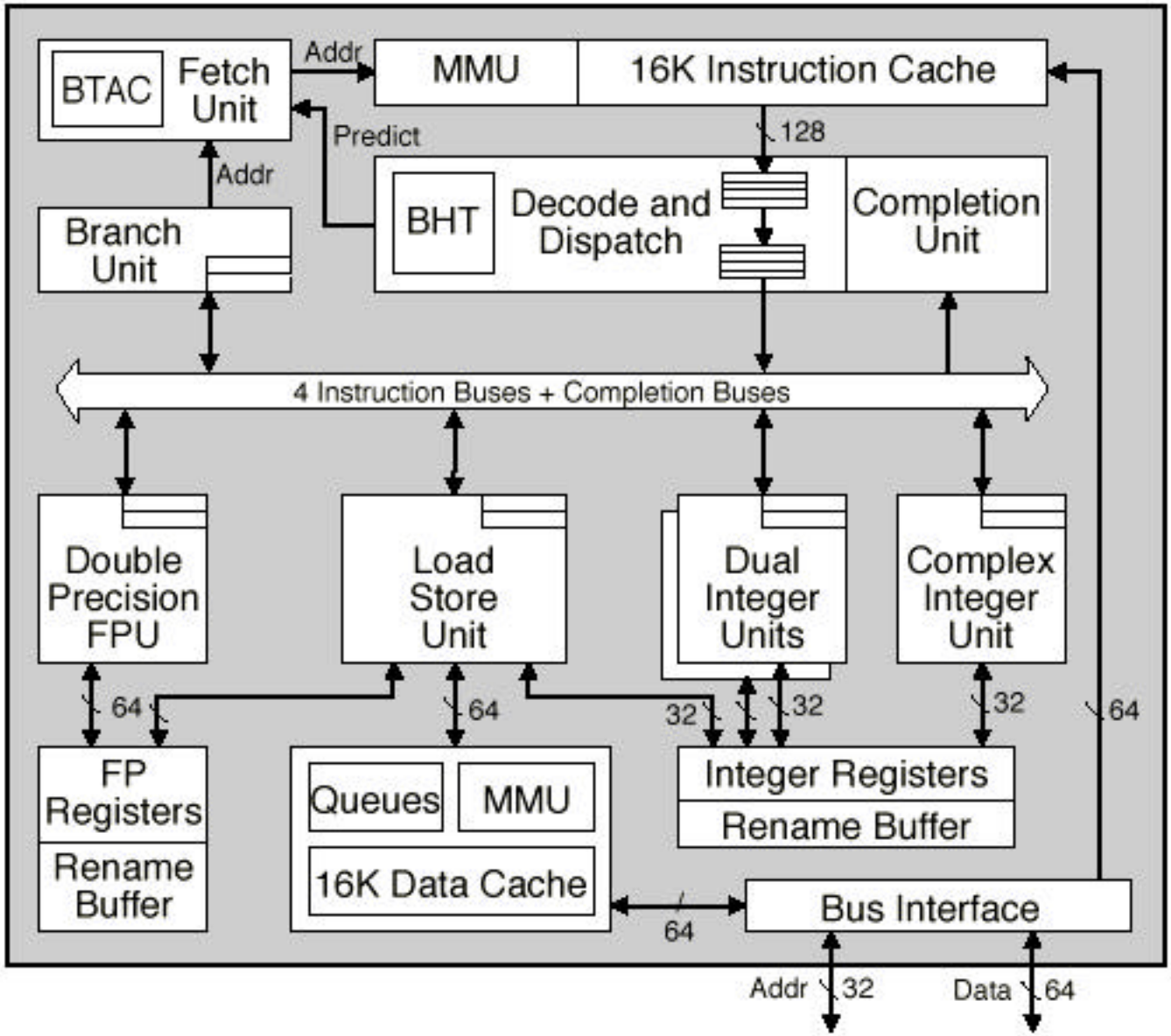
- Begin issuing and executing instructions beyond branch

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**

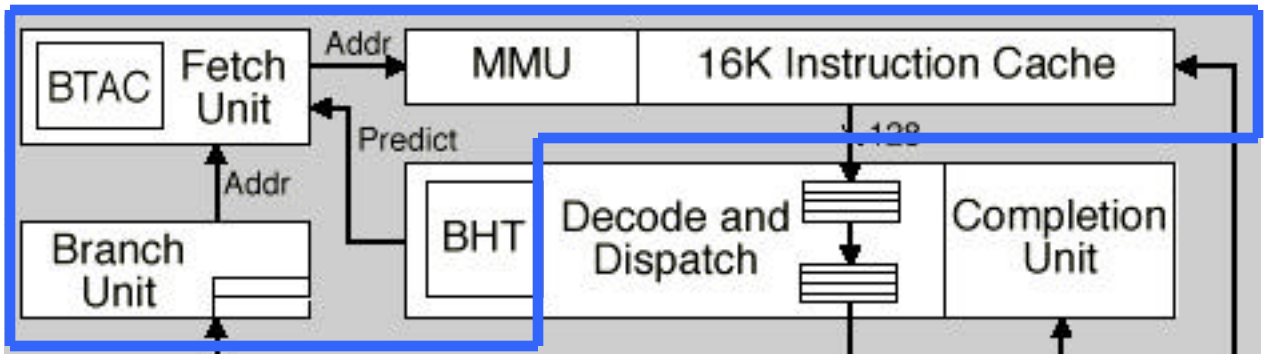
Execute

- **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)

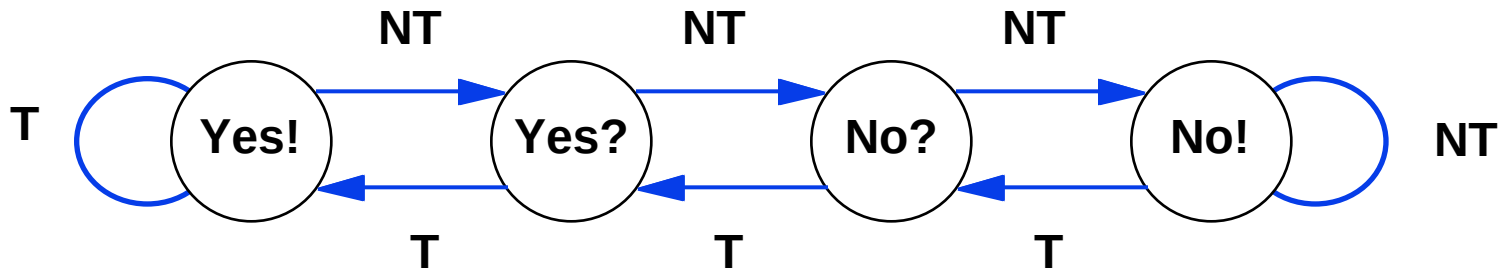
- **Target addresses of recently-executed, predicted-taken branches**
  - 64 entries
  - Indexed by instruction address
- **Accessed in parallel with instruction fetch**
- **If hit, fetch at predicted target starting next cycle**

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**

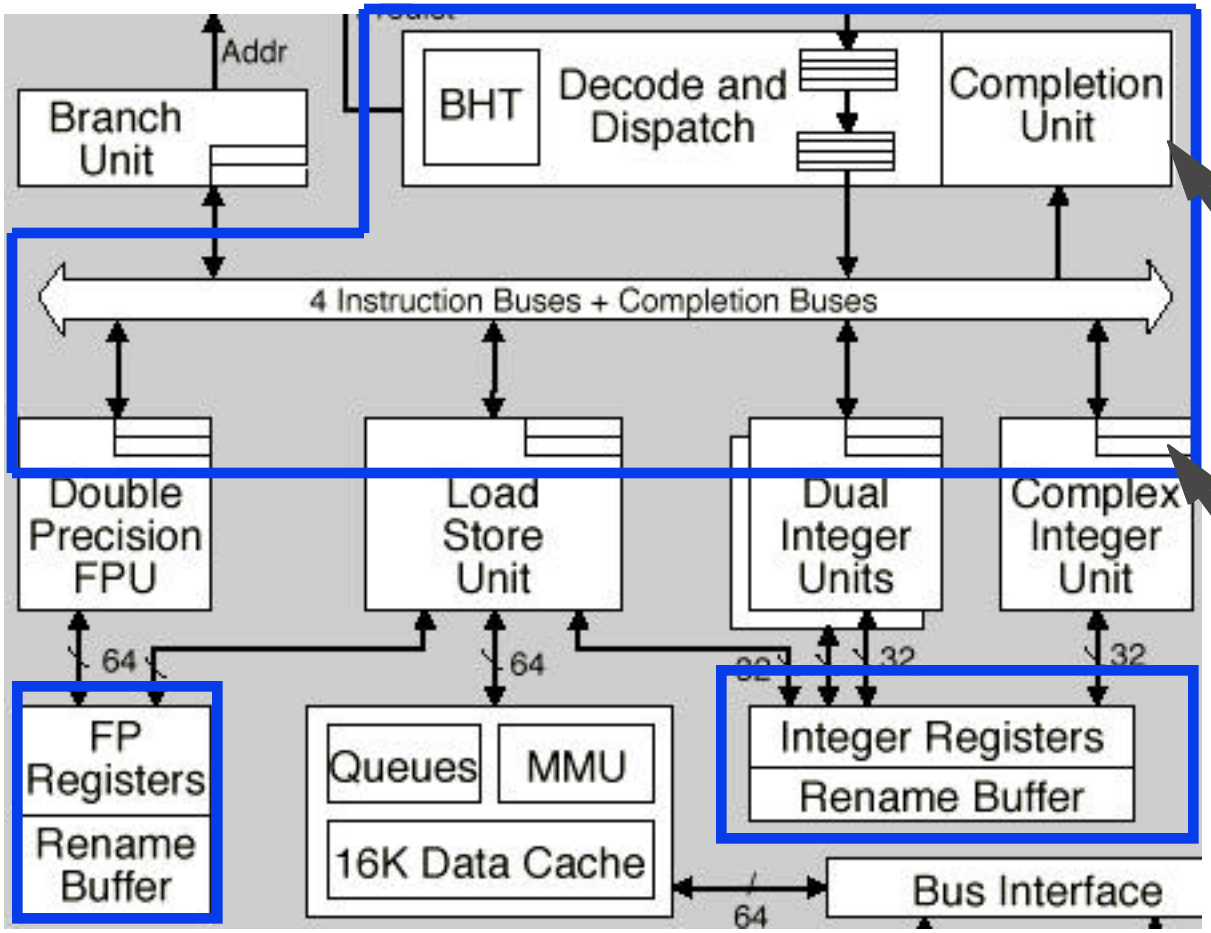
  ≥ 3 cycle penalty if mispredict



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



Retirement Buffer

"Reservation Stations"

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

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**

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**

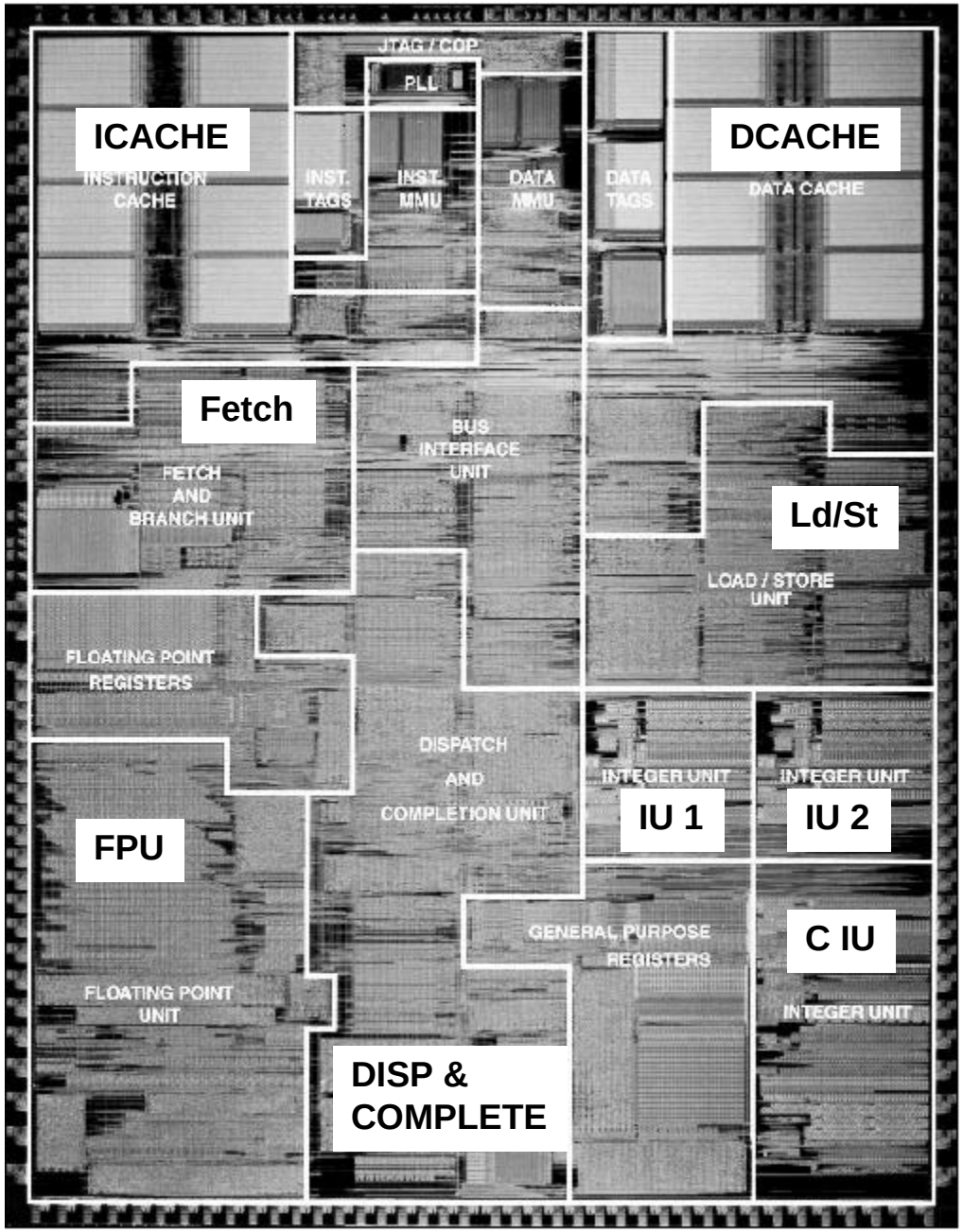


Figure 3. The PowerPC 604 incorporates 3.6 million transistors on a 12.4 × 15.8 mm die using 0.65-micron, five-layer-metal CMOS.

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

Execution Example

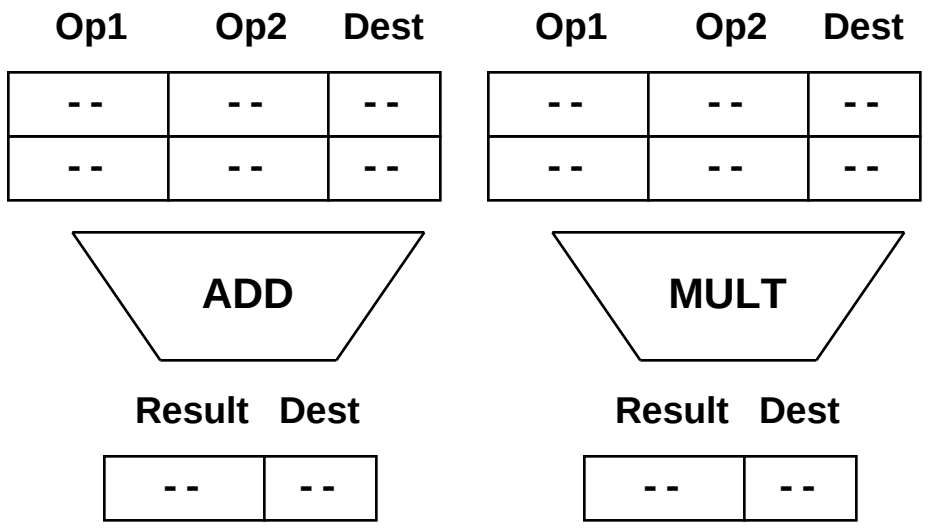
Assumptions

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

```
v:   add.d $f10, $f2,  $f4
w:   mul.d $f10, $f10,  $f6
x:   add.d $f12, $f10, $f8
y:   add.d $f4,  $f4,  $f6
z:   add.d $f10, $f4,  $f8
```

	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



Execution Example Cycle 1

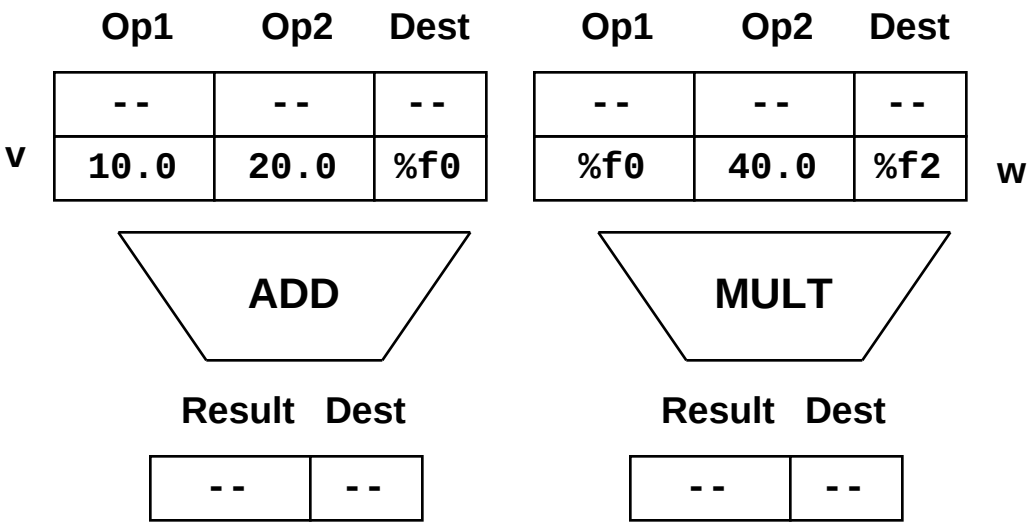
Actions

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

```
v:   add.d $f10, $f2,  $f4
w:   mul.d $f10, $f10,  $f6
x:   add.d $f12, $f10, $f8
y:   add.d $f4,  $f4,  $f6
z:   add.d $f10, $f4,  $f8
```

	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



Execution Example Cycle 2

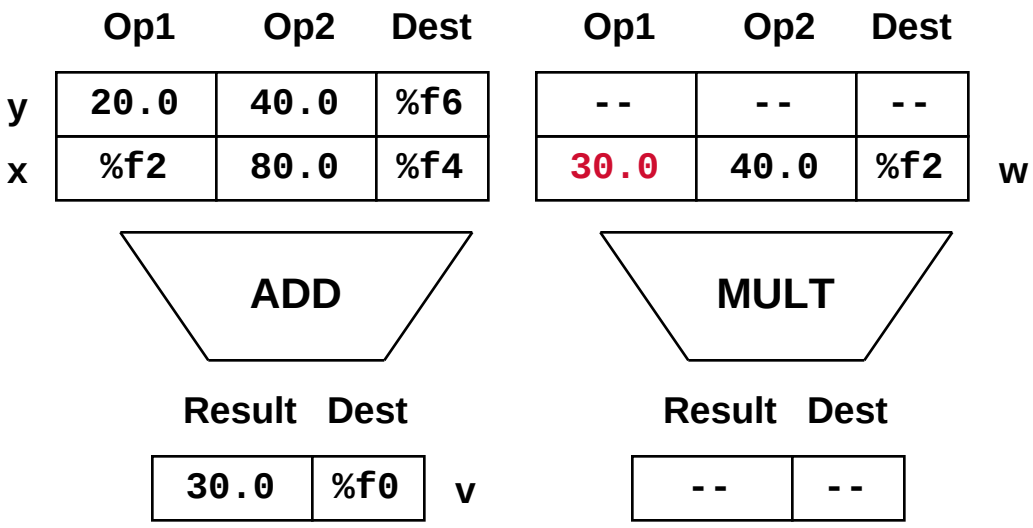
Actions

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

```
v:   add.d $f10, $f2,  $f4
w:   mul.d $f10, $f10,  $f6
x:   add.d $f12, $f10, $f8
y:   add.d $f4,  $f4,  $f6
z:   add.d $f10, $f4,  $f8
```

	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



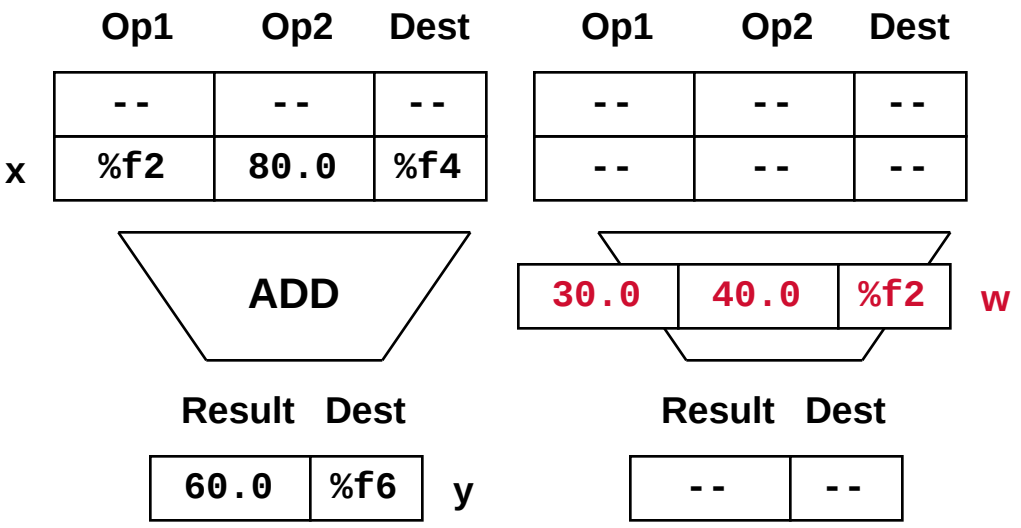
Cycle 3

- **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:   add.d $f10, $f2,  $f4
w:   mul.d $f10, $f10,  $f6
x:   add.d $f12, $f10, $f8
y:   add.d $f4,  $f4,  $f6
z:   add.d $f10, $f4,  $f8
```

	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:   add.d $f10, $f2,  $f4
w:   mul.d $f10, $f10,  $f6
x:   add.d $f12, $f10, $f8
y:   add.d $f4,  $f4,  $f6
z:   add.d $f10, $f4,  $f8
```

	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

ADD

Result	Dest
--	--

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

MULT

Result	Dest	
120.0	%f2	w

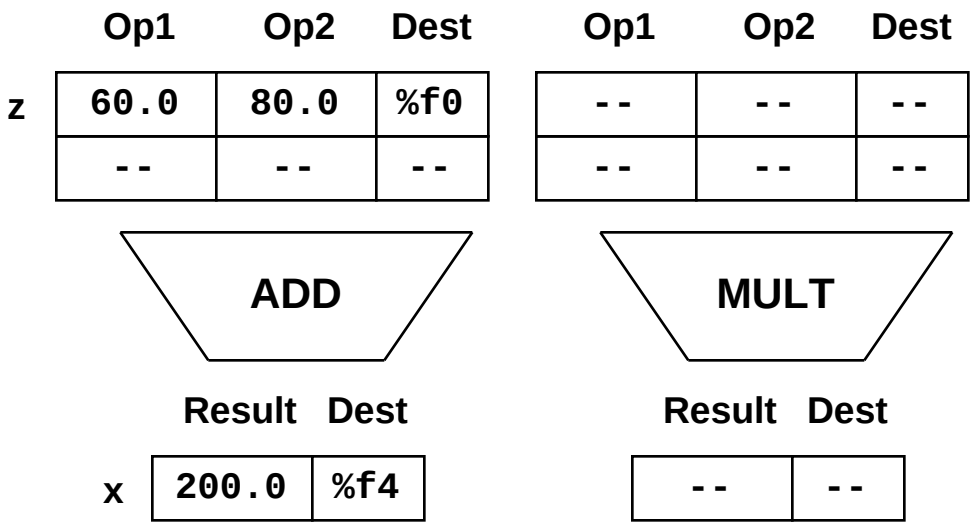
Execution Example Cycle 5

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

```
v:   add.d $f10, $f2,  $f4
w:   mul.d $f10, $f10,  $f6
x:   add.d $f12, $f10, $f8
y:   add.d $f4,  $f4,  $f6
z:   add.d $f10, $f4,  $f8
```

	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



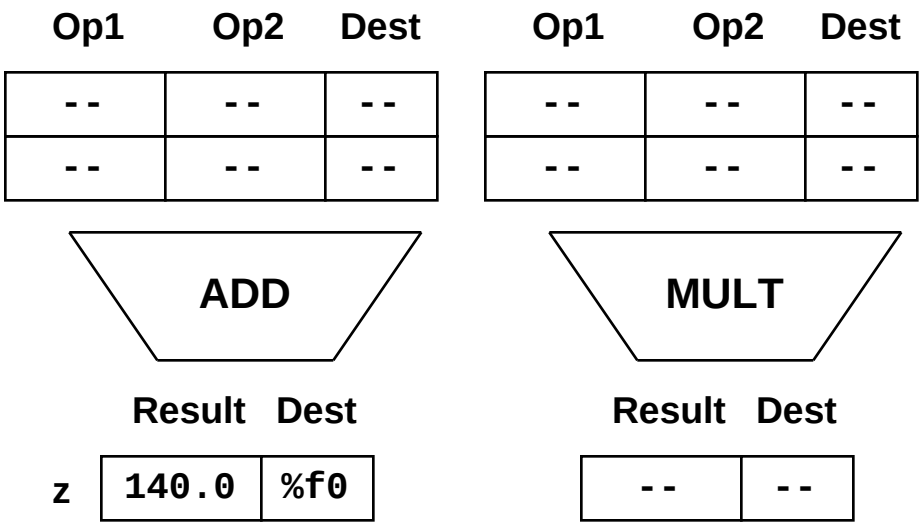
Execution Example Cycle 6

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

```
v:   add.d $f10, $f2,  $f4
w:   mul.d $f10, $f10,  $f6
x:   add.d $f12, $f10, $f8
y:   add.d $f4,  $f4,  $f6
z:   add.d $f10, $f4,  $f8
```

	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



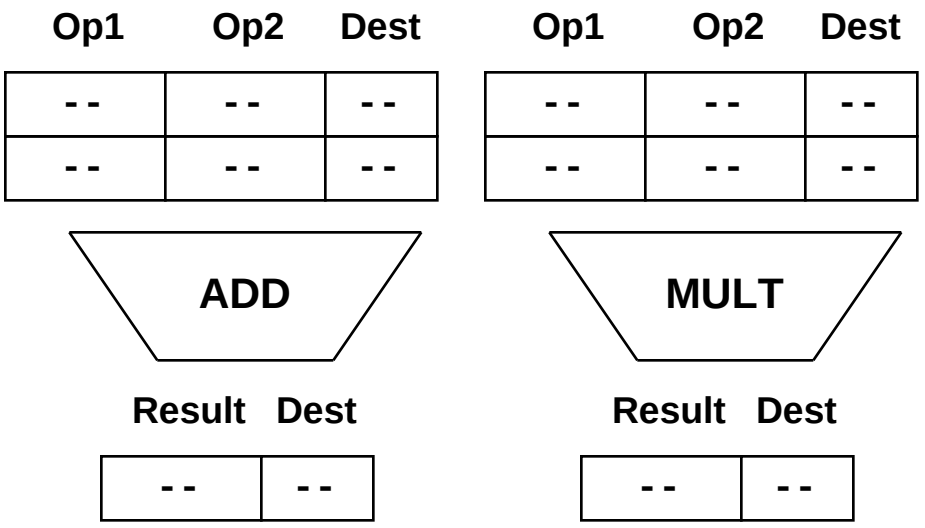
Execution Example Cycle 7

- **Instruction z retired**

```
v:   add.d $f10, $f2,  $f4
w:   mul.d $f10, $f10,  $f6
x:   add.d $f12, $f10, $f8
y:   add.d $f4,  $f4,  $f6
z:   add.d $f10, $f4,  $f8
```

	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



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

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

```
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;
}
```

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

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
```

Some Interesting Patterns

PPPPPPPPP

+1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +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 1 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%**
  - Not clear why
- **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

	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

Pentium II

- **Random shows clear penalty**
- **But others do well**
  - Perhaps 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**

Avoiding Branch Instructions

```
#define ABS(x) abs(x)
```

```
ax = ABS(dat[i]);
```

MIPS Code

```
lw      r2,0(r4)    # dat[i]
nop
sra     r3,r2,31    # t = x >> 31
xor     r2,r2,r3    # v = x ^ t
subu    r2,r2,r3    # ax = v - t
```

Idea

`t = x >> 31`

- 000…0 for x ≥ 0
- 111…1 for x < 0

  `== -1`

`v = x ^ t`

- x for x ≥ 0
- ~x for x < 0

`ax = v - t`

- x for x ≥ 0
- `-x` for x < 0

Performance

- Insensitive to pattern
- Faster in all cases

Other Branch-Less Computations

Min & Max

```
msk = (x-y) >> 31
```

- **Could give error when x << 0 and y >> 0, or vice-versa**

```
min = msk & x + ~msk & y
```

Floating Point Absolute Value

- **Unix `fabs` defined in math.h**
- **Hardware supported operation**
  - Clears single bit in IEEE FP format
- **Always faster than conditional version, and no branch penalty**

PPC 604e	Cond		fabs	
Pattern	Cycles	Penalty	Cycles	Penalty
PPPPPPPPP	9.2	0	7.1	0
RRRRRRRRR	12.6	3.4	7.1	0
N*N[PNPN]	15.8	6.6	7.2	0
N*P[PNPN]	15.9	6.7	7.1	0
N*N[PPNN]	12.5	3.3	7.1	0
N*P[PPNN]	12.5	3.3	7.1	0

A 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**
  - E.g., multiplies in loop1_1

Think Parallel

- **Try to get lots of things going at once**

Not a Truly Parallel Machine

- **Bounded resources**
- **Access from limited code window**

Parallel Accumulation

```
/* Double loop */
double loop2_sum() {
    int i;
    double sum0 = 0.0;
    double sum1 = 0.0;
    for (i = 0; i < CNT; i+= 2) {
      sum0 += dat[i];
      sum1 += dat[i+1];
    }
    return sum0 + sum1;
}
```

Single Accumulator

- **Performance limited by adder latency**
  - = 3 cycles

Unrolled Loop

- **Perform multiple accumulations of subsets**
  - Reduce data dependencies
- **Exploit high throughput of FP adder**