

University of Southern California

Viterbi School of Engineering

EE352

Computer Organization and Architecture

Exploiting Instruction Level Parallelism

References:

- 1) Textbook
- 2) Mark Redekopp's slide series

Shahin Nazarian

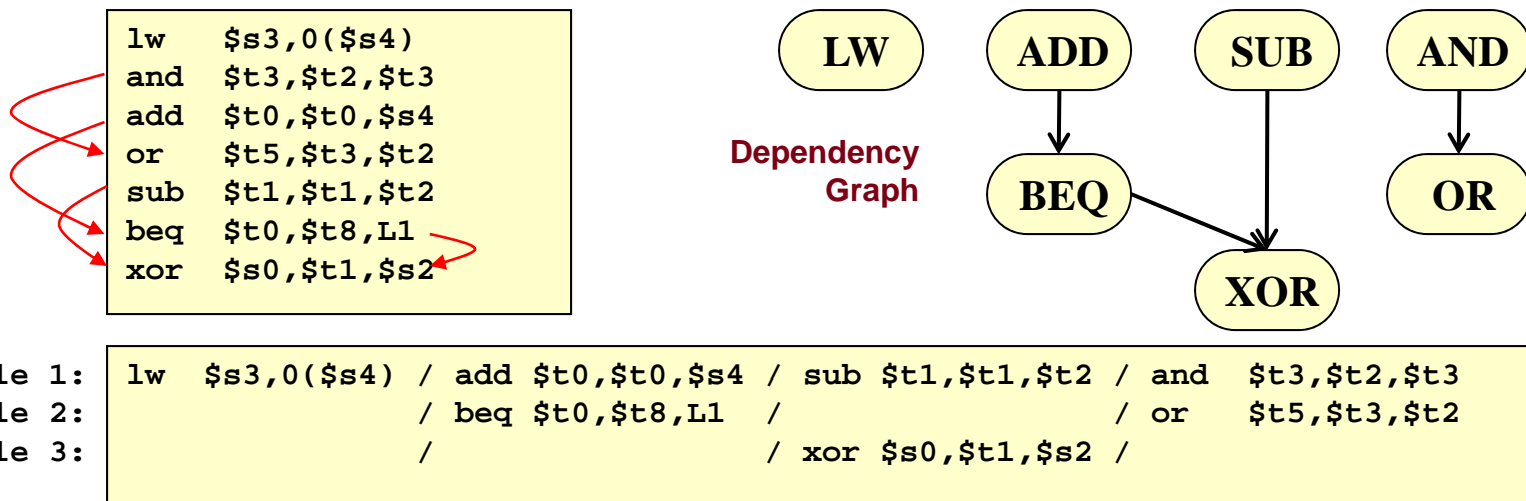
Spring 2010

Exploiting Parallelism

- With increasing transistor budgets of modern processors (i.e. can do more things at the same time) the question becomes how do we find enough *useful* tasks to increase performance, or, put another way, what is the most effective ways of exploiting parallelism!
- Many types of parallelism available
 - Instruction Level Parallelism (ILP): Overlapping instructions within a single process/thread of execution
 - Thread Level Parallelism (TLP): Overlap execution of multiple processes / threads
 - Data Level Parallelism (DLP): Overlap an operation (instruction) that is to be applied to multiple data values (usually in an array)
 - `For(i=0; i < MAX; i++) { A[i] = A[i] + 5; }`

Instruction Level Parallelism (ILP)

- Although a program defines a sequential ordering of instructions, in reality many instructions can be executed in parallel
- ILP refers to the process of finding instructions from a single program/thread of execution that can be executed in parallel
- Data flow (data dependencies) is what truly limits ordering
- Independent instructions (no data dependencies) can be executed at the same time)
- Control hazards also provide some ordering constraints



Basic Blocks

- **Basic Block** (def.) = Sequence of instructions that will always be executed together
 - No conditional branches out
 - No branch targets coming in
 - Also called “straight-line” code
 - Average size: 5-7 instructions
- Instructions in a basic block can be overlapped if there are no data dependencies
- Control dependences really limit our window of possible instructions to overlap
 - W/o extra hardware, we can only overlap execution of instructions within a basic block

```
        lw    $s3, 0($s4)
        and   $t3, $t2, $t3
L1:     add   $t0, $t0, $s4
        or    $t5, $t3, $t2
        sub   $t1, $t1, $t2
        beq   $t0, $t8, L1
        xor   $s0, $t1, $s2
```

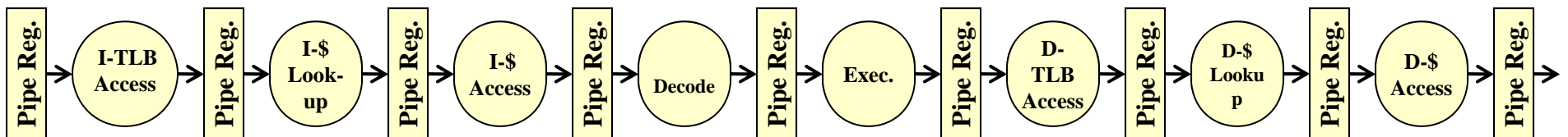
This is a
basic block
(starts w/
target, ends
with branch)

Current Datapath Situation

- 5 Stage Pipeline
 - Best case CPI = 1
 - No structural hazards
 - Control hazards = 1 branch delay slot
 - Data hazards mostly solved by forwarding
 - LW => dependent ALU: 1 clock issue latency
 - But what about cache misses ?!?
- How can we exploit more ILP? (modern processor cores rely on some combination of these three concepts)
 - Superpipelining
 - Superscalar
 - Out-of-Order execution and Speculation

Superpipelining

- Divide our 5 stages into several more stages
- Benefits: Greater PEAK throughput
 - More overlapped instructions
 - Shorter clock cycle time (shorter stages)
- Problems: Sustained throughput < Peak throughput
 - Eventually can't keep dividing stages (small time overhead per stage)
 - Greater branch penalty (especially when branch every 5-7 instrucs.)
 - Possibly more hazards & penalties
 - ALU instruction followed by LW may have to wait n cycles rather than 1
 - Greater hardware complexity (e.g. forwarding paths to > 3 stages)

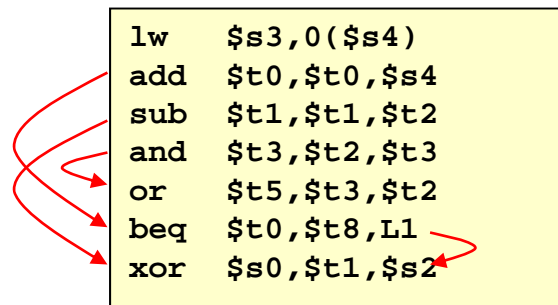


Superscalar (Multiple Issue)

- Multiple “pipelines”
 - **K-way superscalar** = Fetch, decode, and execute max of k instructions at once
- Benefits
 - Theoretical $CPI < 1$ ($IPC > 1$)
- Problems
 - Hazards
 - Dependencies between instructions in different pipelines
 - Branch requires flushing all pipelines
 - Finding enough parallel instructions

Speculation

- All other methods still stall on cache miss or have to wait for a branch outcome to continue executing instructions
- **Speculation** = Don't let hazards stop you from executing instructions
 - Speculate past branches flushing instructions if we're wrong
 - Execute instructions not dependent on a cache miss while waiting
 - Helps us find enough ILP to keep pipelines full



Cycle 1:	lw	\$s3,0(\$s4)	/	add	\$t0,\$t0,\$s4	/	sub	\$t1,\$t1,\$t2	/	and	\$t3,\$t2,\$t3		
Cycle 2:					/	beq	\$t0,\$t8,L1	/	<u>xor</u>	<u>\$s0,\$t1,\$s2</u>	/	or	\$t5,\$t3,\$t2

Multiple Issue Processors

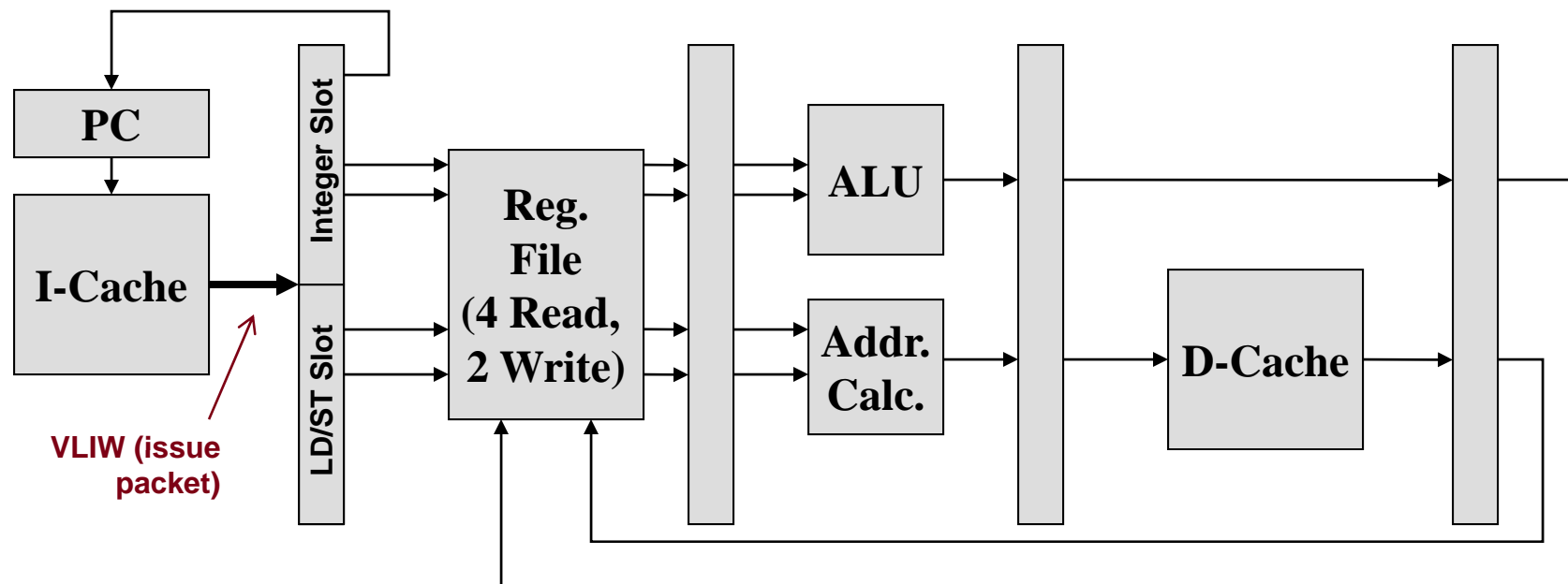
- Duplicate pipeline resources (functional units, etc.)
- Techniques
 - Static Multiple Issue/Static Scheduling (Compiler-based)
 - Dynamic Multiple Issue/ Dynamic Scheduling (HW-based)
 - Both of these can be combined w/ speculation
- Instruction Execution Ordering
 - **In-Order**: If an instruction stalls, all instructions behind it also stall
 - **Out-of-Order**: If an instruction stalls, independent instructions are allowed to continue

Static Multiple Issue

- Compiler is responsible for finding and packaging instructions into issue packets that can execute in parallel
 - Only certain combinations of instructions can be in a packet together
 - Instruction packet example:
 - (1) Integer/Branch instruction slot
 - (1) LD/ST instruction
 - (1) FP operation
- An issue packet is often thought of as an LONG instruction containing multiple instructions (a.k.a. Very Long Instruction Word)
 - Intel's Itanium uses this technique (static multiple issue) but calls it EPIC (Explicitly Parallel Instruction Computer)

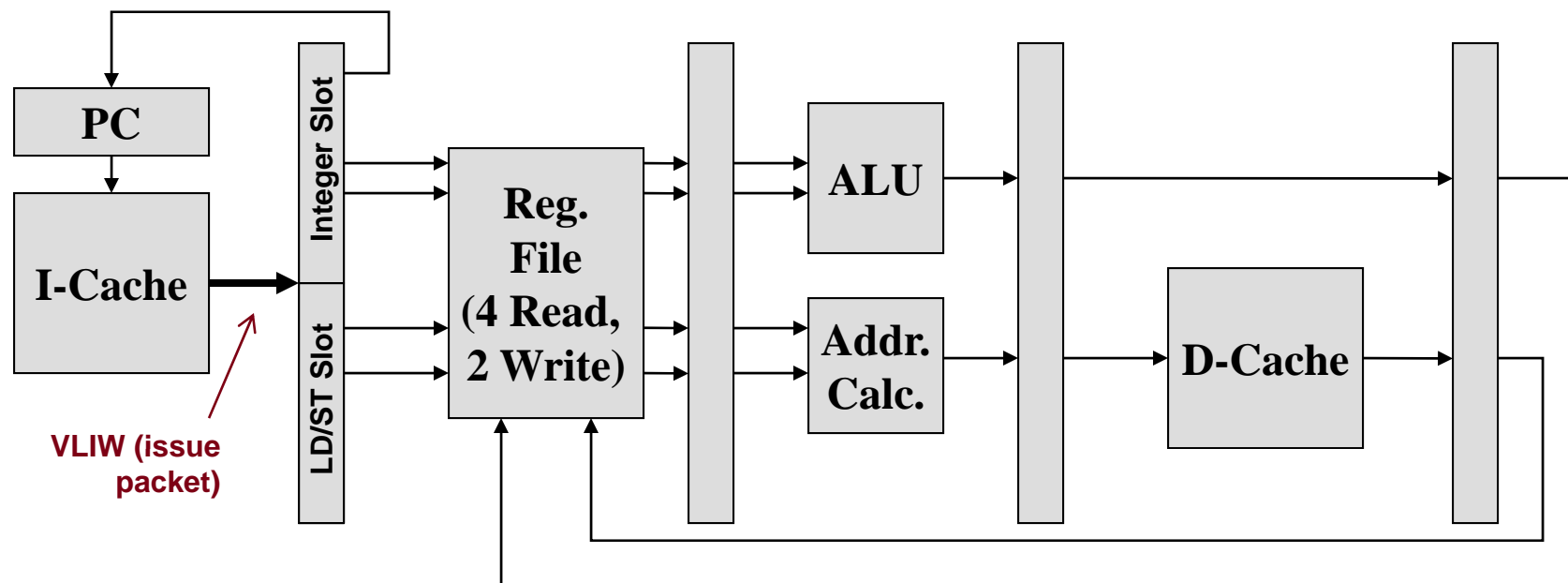
Example 2-way VLIW machine

- One issue slot for integer/branch operations
- One issue slot for LD/ST instructions
- I-Cache reads out an entire issue packet (VLIW)
- Register file needs to double # of read/write ports
- Address Calculation Unit (Adder) is added



2-way VLIW Scheduling

- No forwarding w/in an issue packet (between instructions in a packet)
- Full forwarding paths for instructions already in the pipeline even across slots/pipes (i.e. from 'add' in MEM stage to 'lw' in EX stage)
- Latency of LW is still 1 stall cycle for dependent instructions
- Assume early branch detection (in DECODE stage)



Sample Scheduling

- Schedule the following loop body on our 2-way static issue machine

```
for(i=MAX; i != 0; i--)
    A[i] = A[i] + 5;
```

```
$s0 = pointer to A[i]
$s1 = i = # of iterations

L1: lw    $t1,0($s0)
     addi $t1,$t1,5
     sw    $t1,0($s0)
     addi $s0,$s0,4
     addi $s1,$s1,-1
     bne  $s1,$zero,L1
```

Int./Branch Slot	LD/ST Slot
	lw \$t1,0(\$s0)
addi \$s1,\$s1,-1	
addi \$t1,\$t1,5	
addi \$s0,\$s0,4	sw \$t1,0(\$s0)
bne \$s1,\$zero,L1	

w/o modifying original code but with code movement
IPC = 6 instrucs. / 5 cycles = 1.2

Int./Branch Slot	LD/ST Slot
addi \$s1,\$s1,-1	lw \$t1,0(\$s0)
addi \$s0,\$s0,4	
addi \$t1,\$t1,5	
bne \$s1,\$zero,L1	sw \$t1,-4(\$s0)

w/ modifications and code movement
IPC = 6 instrucs. / 4 cycle = 1.5

Loop Unrolling

- Often not enough ILP w/in a single iteration (body) of a loop
- However, different iterations of the loop are often independent and can thus be run in parallel
- This parallelism can be exposed in static issue machines via loop unrolling
 - Assume the loop executes a multiple of n (i.e. $k*n$) times
 - Copy the body of the loop n times and iterate only k times
 - Instructions from different body iterations can be run in parallel

```
// Assume MAX is a multiple of 4
for(i=MAX; i != 0; i--)
    A[i] = A[i] + 5;
```

```
// Loop unrolled 4 times
for(i=MAX; i != 0; i=i-4){
    A[i] = A[i] + 5;
    A[i+1] = A[i+1] + 5;
    A[i+2] = A[i+2] + 5;
    A[i+3] = A[i+3] + 5;
}
```

Loop Unrolling

```
for(i=MAX; i != 0; i--)  
    A[i] = A[i] + 5;
```

```
$s0 = pointer to A[i]  
$s1 = i = # of iterations  
  
L1: lw    $t1,0($s0)  
     addi $t1,$t1,5  
     sw    $t1,0($s0)  
     addi $s0,$s0,4  
     addi $s1,$s1,-1  
     bne  $s1,$zero,L1
```

Original Code

A side effect of unrolling is the reduction of overhead instructions (less branches and counter/ptr. updates)

```
// Loop unrolled 4 times  
for(i=MAX; i != 0; i=i-4){  
    A[i] = A[i] + 5;  
    A[i+1] = A[i+1] + 5;  
    A[i+2] = A[i+2] + 5;  
    A[i+3] = A[i+3] + 5;  
}
```

Unrolled Code

```
$s0 = pointer to A[i]  
$s1 = i = # of iterations  
  
L1: lw    $t1,0($s0)  
     addi $t1,$t1,5  
     sw    $t1,0($s0)  
     lw    $t1,4($s0)  
     addi $t1,$t1,5  
     sw    $t1,4($s0)  
     lw    $t1,8($s0)  
     addi $t1,$t1,5  
     sw    $t1,8($s0)  
     lw    $t1,12($s0)  
     addi $t1,$t1,5  
     sw    $t1,12($s0)  
     addi $s0,$s0,16  
     addi $s1,$s1,-4  
     bne  $s1,$zero,L1
```

Code Movement & Hazards

- To effectively schedule the code, the compiler will often move code up or down but must take care not to change the intended program behavior
- Must deal with WAW and WAR hazards in addition to RAW hazards when moving code
 - **WAW and WAR hazards are not TRUE hazards** (no data communication between instrucs.) but simply conflicts because we use the same register...we call them **antidependences!**
 - How can we solve? Register renaming

```
L1: add $t0,$t1,$t2
     add $s0,$t0,$t1
     lw  $t0,0($t1)
```

LW instruction is
REALLY independent
Could LW instruction be
moved between the 2
add's?
Not as is, WAR hazard

```
L1: add $t0,$t1,$t2
     add $s0,$t0,$t1
     lw  $t0,0($t1)
     sub $s1,$t0,$t3
```

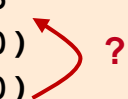
Could LW instruction
be run in parallel with
or before first add?
Not as is, WAW hazard
(race condition)

Register Renaming

- Unrolling is not enough because even though each iteration is independent there are conflicts in the use of registers (\$t1 in this case)
 - Can't move another 'lw' instruction up until 'sw' is complete due to a WAR hazard even though there is not a true data dependence
- Since there is no true dependence we can solve the problem by register renaming
- **Register Renaming: using different registers to solve WAR / WAW hazards**

```
$s0 = pointer to A[i]
$s1 = i = # of iterations

L1: lw    $t1,0($s0)
     addi $t1,$t1,5
     sw    $t1,0($s0)
     lw    $t1,4($s0)
     addi $t1,$t1,5
     sw    $t1,4($s0)
     lw    $t1,8($s0)
     addi $t1,$t1,5
     sw    $t1,8($s0)
     lw    $t1,12($s0)
     addi $t1,$t1,5
     sw    $t1,12($s0)
     addi $s0,$s0,16
     addi $s1,$s1,-4
     bne  $s1,$zero,L1
```



Scheduling w/ Unrolling & Renaming

- Schedule the following loop body on our 2-way static issue machine

```

$s0 = pointer to A[i]
$s1 = i = # of iterations

L1: lw    $t1,0($s0)
     addi $t1,$t1,5
     sw    $t1,0($s0)
     lw    $t2,4($s0)
     addi $t2,$t2,5
     sw    $t2,4($s0)
     lw    $t3,8($s0)
     addi $t3,$t3,5
     sw    $t3,8($s0)
     lw    $t4,12($s0)
     addi $t4,$t4,5
     sw    $t4,12($s0)
     addi $s0,$s0,16
     addi $s1,$s1,-4
     bne  $s1,$zero,L1
    
```

Int./Branch Slot	LD/ST Slot
	lw \$t1,0(\$s0)
addi \$s1,\$s1,-4	lw \$t2,4(\$s0)
addi \$t1,\$t1,5	lw \$t3,8(\$s0)
addi \$t2,\$t2,5	lw \$t4,12(\$s0)
addi \$t3,\$t3,5	sw \$t1,0(\$s0)
addi \$t4,\$t4,5	sw \$t2,4(\$s0)
addi \$s0,\$s0,16	sw \$t3,8(\$s0)
bne \$s1,\$zero,L1	sw \$t4,-4(\$s0)


w/ Loop Unrolling and Register Renaming

(Notice how the compiler would have to modify the code to effectively reschedule)

IPC = 15 instrucs. / 8 cycle = 1.875

Loop Unrolling & Register Renaming Summary

- Loop unrolling increases code size (memory needed to store instructions)
- Register renaming burns more registers and thus may require HW designers to add more registers
- Must have some amount of independence between loop bodies
 - Dependence between iterations is known as loop carried dependence



```
// Dependence between iterations
A[0] = 7;
for(i=1; i < MAX; i++)
    A[i] = A[i-1] + 5;
```

Data Dependency Hazards Summary

- **RAW is the Only real data dependence**
 - Must be respected in terms of code movement and ordering
 - Forwarding reduces latency of dependent instructions
- **WAW and WAR hazards = Antidependencies**
 - Solved by using register renaming
- **RAR = No issues / dependencies**

New Data Hazard Issue

- All data hazards examined so far deal with register value dependencies and are easy to identify because each register has only one name
- Data dependencies can occur via memory and are harder for the compiler to find at compile time

```
int x, y, *ptr;  
ptr = &x;  
x = 5;  
y = *ptr + 1;
```

```
// x @ address in $s0  
// y @ address in $s1  
// ptr address in $s2  
  
move $s2,$s0      ; ptr=&x  
addi $t0,$zero,5  
sw   $t0,0($s0)   ; x=5  
lw   $t1,0($s2)   ; *ptr  
addi $t2,$t1,1    ; *ptr+1  
sw   $t2,0($s1)
```

Int./Branch Slot	LD/ST Slot
move \$s2,\$s0	
addi \$t0,\$0,5	lw \$t1,0(\$s2)
	sw \$t0,0(\$s0)
addi \$t2,\$t1,1	
	sw \$t2,0(\$s1)

‘sw’ depends on previous ‘addi’ Can we move the ‘lw’ up to enhance performance?
No...Need to wait for ‘sw’!!

Memory Disambiguation

- Memory RAW dependencies are also made harder because of different ways of addressing the same memory location
 - 'sw \$t1, 4(\$s0)' could write the same location as 'lw \$t2, -12(\$s1)' reads if $\$s1 = \$s0 + 16$
- **Memory disambiguation** refers to the process of determining if a sequence of stores and loads reference the same address (ordering often needs to be maintained)
- We can only reorder 'lw' and 'sw' instructions if we can disambiguate their addresses to determine any RAW, WAR, WAW hazards
 - LW -> LW is always fine (RAR)
 - SW -> LW (RAW), LW -> SW (WAR) or SW -> SW (WAW) hazards that need to be disambiguated

[Optional] Speculation in VLIW

- **Handling of Speculation**
 - Compiler can move instructions across branches or perform “speculative” loads
 - To handle misspeculation, often includes “fix-up” routines (code to undo the misspeculation)

```
//if(a==0) A=B; else A=A+4;  
  
    lw   $t1,0($t3)  
    bne  $t1,$0,L1  
    lw   $t1,0($t2)  
    b    L2  
L1:  add  $t1,$t1,4  
    sw   $t1,0($t3)
```

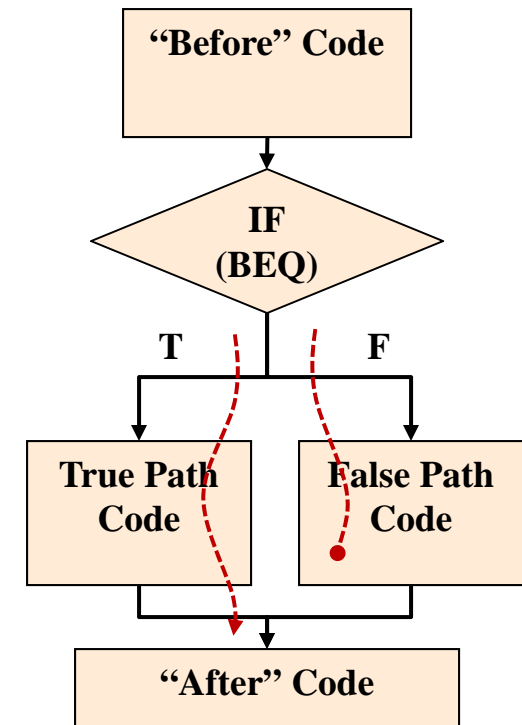
Original Code

```
//if(a==0) A=B; else A=A+4;  
  
    lw   $t1,0($t3)  
    lw   $t8,0($t2)  
    beq  $t1,$0,L1  
L1:  add  $t8,$t1,4  
    sw   $t8,0($t3)
```

‘lw’ speculatively moved but code changed to keep correctness. Increases basic block above ‘beq’.

Predication

- Predicated (def.) = Found upon, based upon
- Used by static issue (VLIW) machines
- 'IF' statement causes a program to branch into two paths and then merge again
- Rather than using branches we can reduce overhead and often increase performance by executing both paths and throwing away the results of one path
- Make instructions predicated on some condition
 - If condition is true instruction completes and updates state (registers) of the processor
 - If condition is false, instruction does not affect processor state



Flowchart perspective of the if...else statement
Predication takes both paths and cancel results of 'false' path

Predication (Cont.)

- Make instructions predicated on some condition
 - If condition is true instruction completes and commits changes to processor state (write results to registers)
 - If condition is false, instruction does not affect processor state
- Add 1-bit predicate registers (p[1]-p[n])
- Removes instruction overhead of branches
 - Can get rid of conditional branch and unconditional branch
 - Converts control hazards to RAW hazards (**predicate bit is essentially a register that is written by one instruction and consumed by a following instruction**)

```
if(x)
    y = 1;
else
    y = 2;
```

Original C Code

```
// Assume x = $t0
// Assume y = $t1

        beq  $t0,$zero,L1
        addi $t1,$zero,1
        b    L2
L1: addi $t1,$zero,2
L2: ...
```

w/o Predication

```
// Assume x = $t0
// Assume y = $t1

        cmp.ne p1,$t0=$zero
(p1)    addi  $t1,$zero,1
(~p1)   addi  $t1,$zero,2
        ...
```

w/ Predication

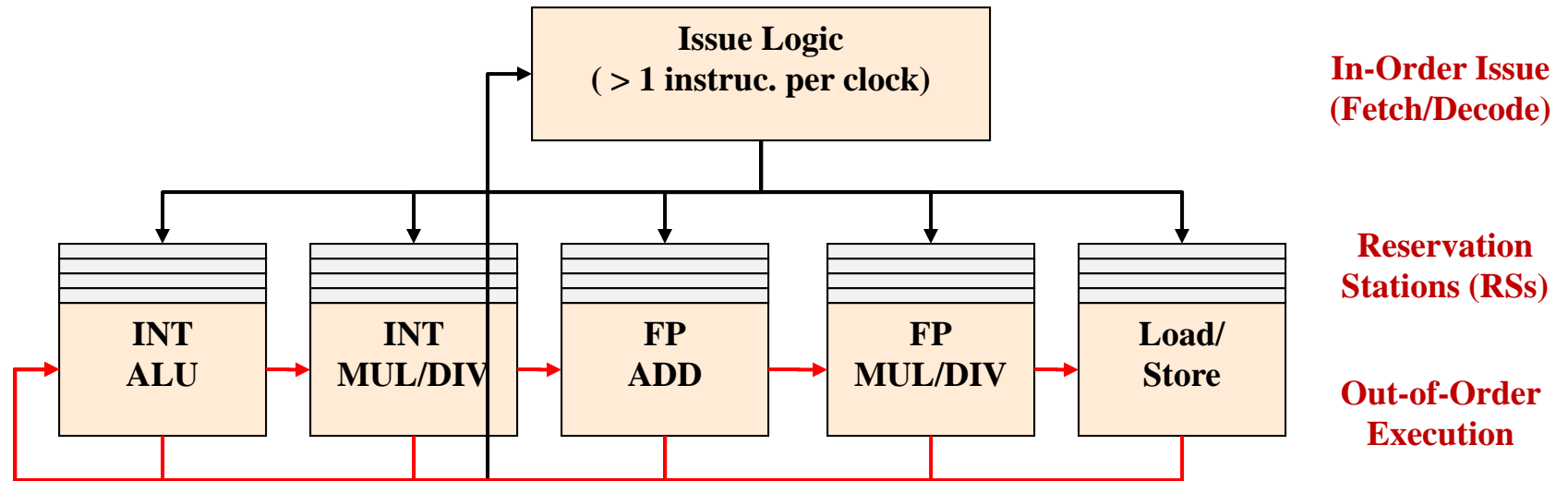
Itanium 2 Case Study

- Max 6 instruction issues/clock
- 6 Integer Units, 4 Memory units, 3 Branch, 2 FP
 - Although full utilization is rare
- Registers
 - (128) 64-bit GPRs
 - (128) FPRs
 - (64) 1-bit predicate registers
- On-chip L3 cache (12 MB or cache memory)

Dynamic Multiple Issue

- Burden of scheduling code for parallelism is placed on the HW and is performed as the program runs (not necessarily at compile time)
 - Compiler can help by moving code, but HW guarantees correct operation no matter what
- Goal is for HW to determine data dependencies and let independent instructions execute even if previous instructions (dependent on something) are stalled
 - We call this a form of Out-of-Order Execution
- Primarily used in conjunction with speculation methods but we'll start by examining non-speculative methods (i.e. don't execute until all previous branches are resolved)

Dynamic Multiple Issue



Issue / Dispatch

- Fetch, decode, and issue instructions to execution (functional) units in-order
- During issue, an instruction will try to read its source operands from the register file
- To handle RAW dependencies, issue unit maintains a **register status table** indicating what instruction (reservation station) already in the pipeline is going to write this register
 - If this entry is blank, then no one is currently producing a result for this register and it is safe to read it
 - If the entry is not blank then it contains the name/# of a reservation station which is the latest instruction that will be producing the value for that register. The instruction trying to issue will take with it that reservation station's name and get the data directly from that unit when it executes

Out-of-Order Execution

- Functional units (INT, FP, LD/ST) have buffers (reservation stations) to store instructions waiting to use the units
 - Allow any waiting instruction to execute as soon as it has all of its data operands AND the functional unit is free
 - Instructions dependent on other unexecuted instructions simply wait
 - This means instructions can execute out of program order
- Dynamic scheduling: Process of executing instructions as soon as they have their operands and no hazards exist

Reservation Stations

- Provide the method for register renaming in that each reservation station has a name that other instructions can reference (a set of “virtual registers” transparent to the programmer)
- Contain entries for:
 - **Busy:** Is the functional unit currently computing this entry
 - **Op:** Which operation should the unit perform
 - **Vj,Vk =** Values of up to 2 source operands
 - **Qj,Qk =** If values were not available (RAW hazard), these contain the reservation station name/# who will produce the needed value
 - **A:** For load/store instructions, this contains the offset initially, then the effective address when it is computed
- **Example:** add \$8,\$9,\$10 followed by sub \$4,\$5,\$8

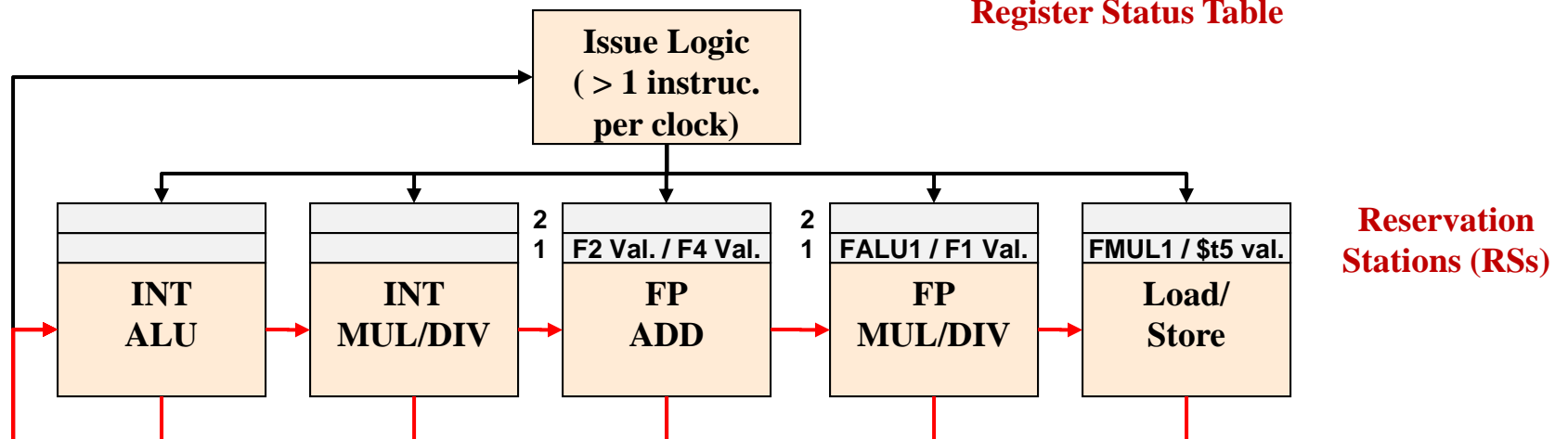
Station	Busy	Op	Vj	Vk	Qj	Qk	A
2	No	Sub	\$5 Val.	-	-	INTALU1	-
1	Yes	Add	\$9 Val.	\$10 Val.	-	-	-

Example 1

```
add.d $f0,$f2,$f4
mul.d $f0,$f0,$f1
s.d   $f0,0($t5)
```

After Issue Of	\$f0	\$f1	...
originally	-	-	-
add.d	FALU1	-	-
mul.d	FMUL1	-	-
s.d	FMUL1	-	-

Register Status Table



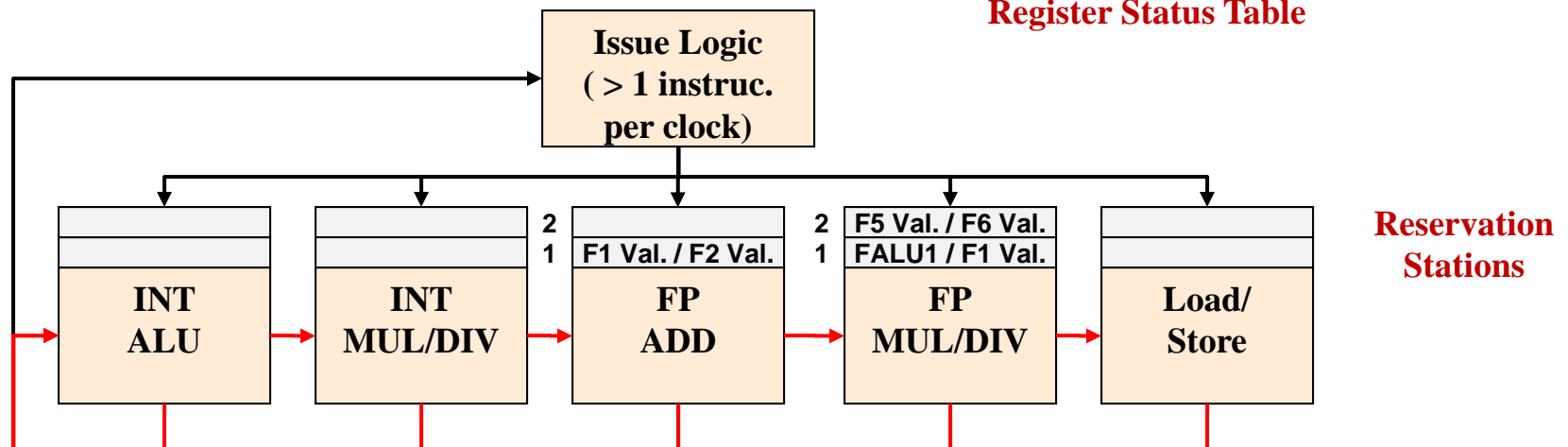
- RAW hazards handled by issuing dependent instruction with name of RS who will produce the result (RS name passed w/ instruction = current entry in status table)
- WAW hazard solved because status table only writes \$f0 with latest instruction

Example 2

```
add.d $f0,$f1,$f2
mul.d $f3,$f0,$f1
div.d $f0,$f5,$f6
```

After Issue Of	\$f0	\$f1	...
originally	-	-	-
add.d	FALU1	-	-
mul.d	FALU1	-	-
div.d	FMUL2	-	-

Register Status Table



- WAR hazard handled because \$f0 in MUL.D has been renamed to “FALU1”
- WAW hazard solved because status table only writes \$f0 with latest instruction. If ADD.D and MUL.D were stalled (waiting for \$f1 to be produced), DIV.D could execute, complete, and write value to \$f0 without affecting correctness of the program

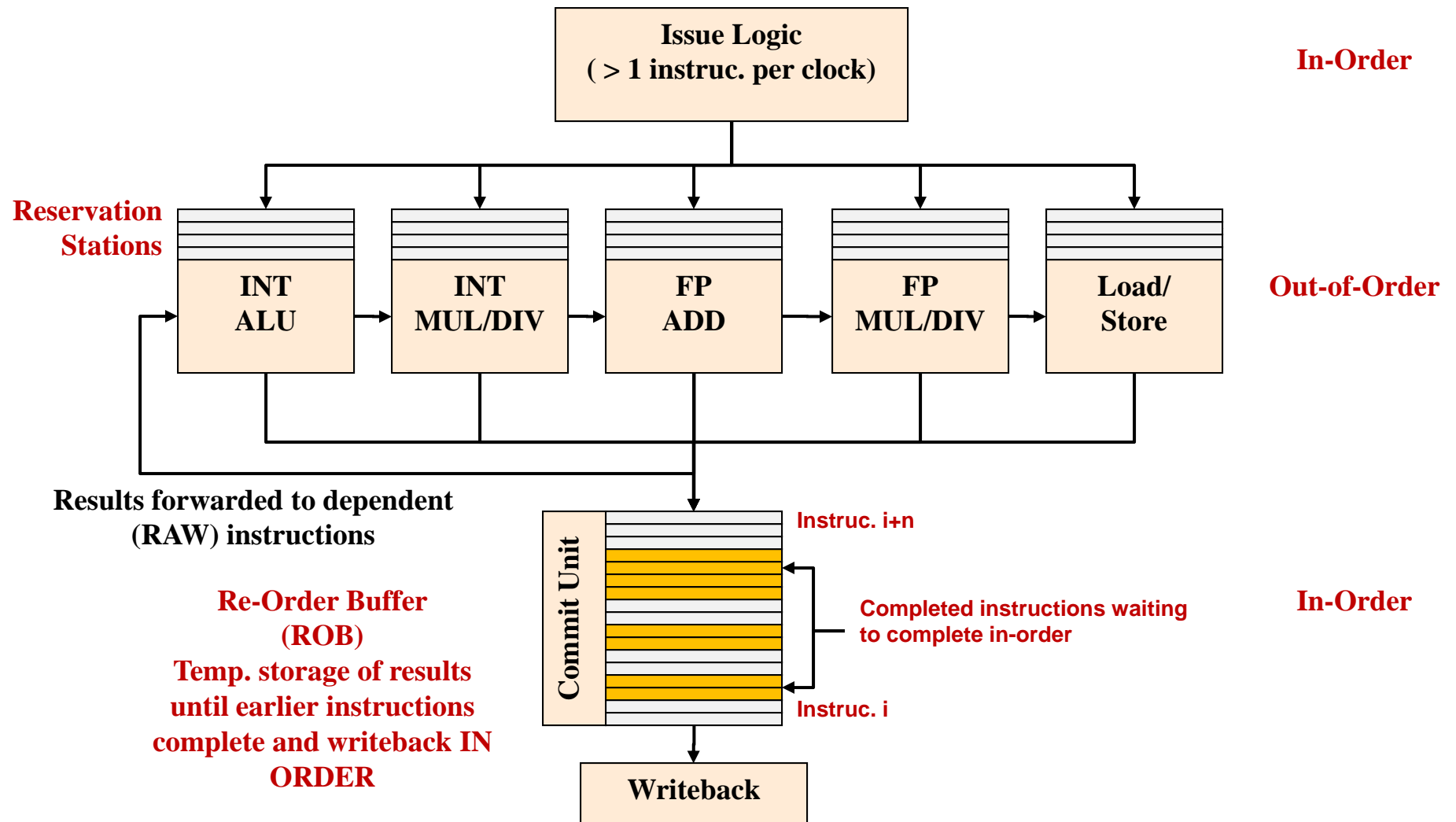
Dynamic Scheduling Summary

- Automatically solves data dependencies w/o compiler help (static scheduling)
- Provides tolerance for long-latency hazards such as cache misses
 - All independent instructions are allowed to execute “around” the stall
 - All dependent instructions stall in their reservation stations
 - Eventually may stall the issue unit (entire processor) if all RSs are used up w/ stalled instructions

Speculation w/ Dynamic Scheduling

- Basic block size of 5-7 instructions between branches limits ability to issue/execute instructions
 - For safety, assume no instructions issued until all previous branch outcomes known
- Speculation allows us to predict a branch outcome and continue *issuing* and *executing* down that path
- To be able to roll-back if we mispredict, we add a structure known as the **commit unit** (or **re-order buffer / ROB**)

Out-Of-Order Diagram

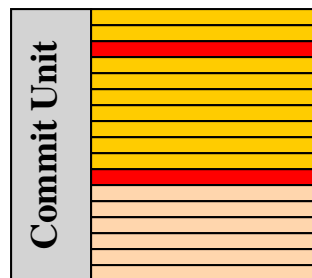
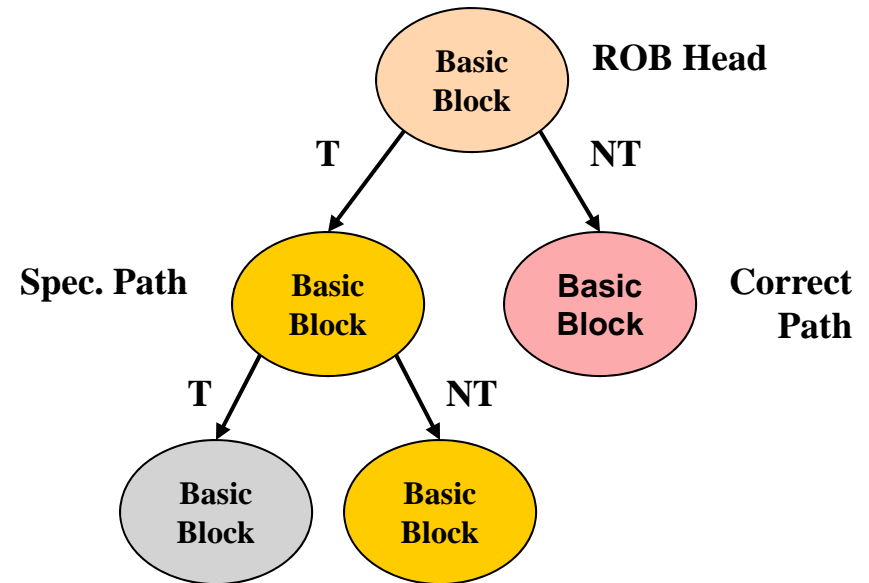


Commit Unit (ROB)

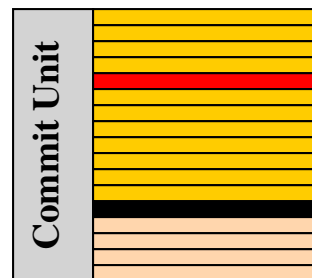
- ROB tail entry is allocated for each instruction on issue (ensuring instruction entries are maintained in program order)
- When an instruction completes, its results are forwarded to others but also stored in the ROB (rather than writing back to reg. file) until it reaches the head of the ROB
 - Issuing instructions now have to fetch values out of the ROB for destination registers of instructions in the ROB
- Commit unit only commits the instruction(s) at the head of the queue and only when they are fully completed
 - Ensures that everything committed was correct
 - If we misspeculate or mispredict a branch then throw away everyone behind it in the ROB and start fresh using the correct outcome

Speculation Example

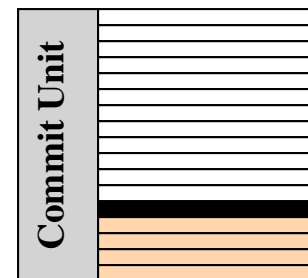
- Misprediction requires flushing instructions behind it wasting all that effort
 - Need good prediction capabilities



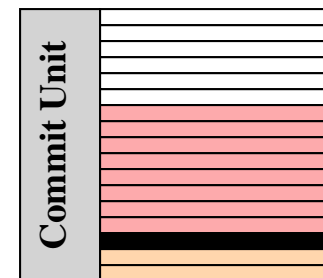
Time 1: ROB
Red Entries = Predicted Branches



Time 2a: ROB
Black Entry = Mispredicted branch



Time 2b:
Flush ROB/Pipeline of instructions behind it



Time 3: ROB
Pipeline begins to fill w/ correct path

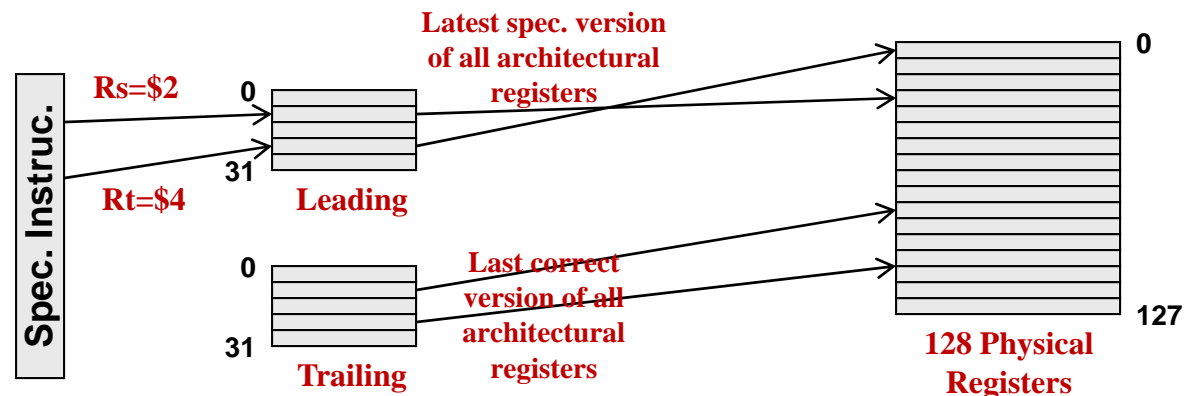
Physical & Architectural Reg's.

- In static scheduling, the compiler accomplished register renaming by changing the instruction to use other programmer-visible GPRs
- In dynamic scheduling, renaming was accomplished using reservation stations or ROB entries as “virtual” registers
- Another HW approach combines the two
 - Architectural registers = The 32 GPRs or 32 FPRs that the programmer can explicitly use as operands
 - Physical registers = A greater number of actual registers than architectural registers that is used as a “pool” for renaming
 - Use of a large (80-128) pool of physical registers with some additional logic makes HW implementation easier/smaller than reservation stations and ROB entries

Physical Register Mappings

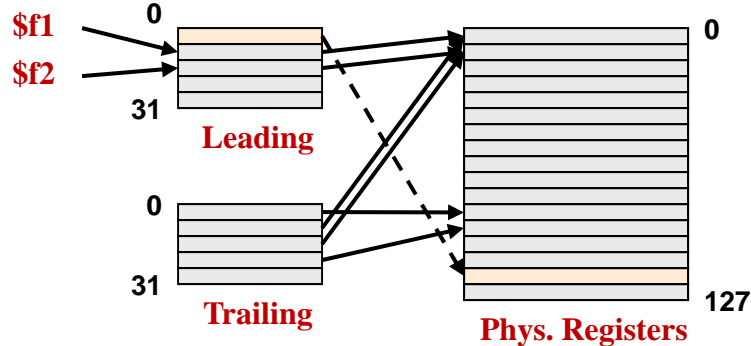
To do renaming we have a large pool of physical registers and two mapping tables

- Tables map architectural registers used in an instruction to a physical register
- **Leading Table:** Similar to “register status table” contains the physical register # used by the last (speculative) instruction issued who will produce the corresponding architectural register
 - Destination operands will be allocated new physical registers & update the leading table
 - Source operands will use entries from leading table to know where to get data
- **Trailing Table:** Contains a mapping of which physical register holds the latest “committed” value of the corresponding architectural register
 - When an instruction commits, trailing table entry for destination operand is updated to point at the physical register it used

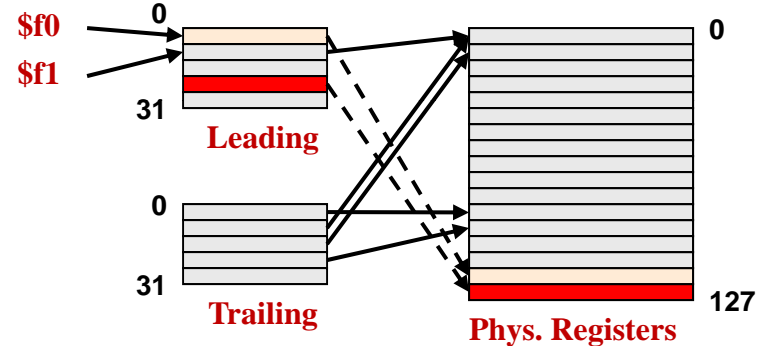


Physical Register Example

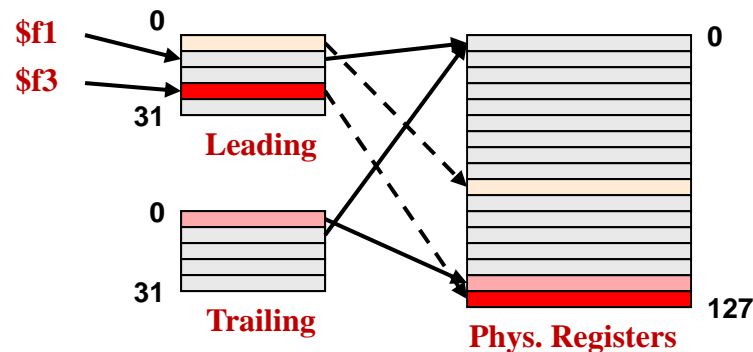
```
add.d $f0,$f1,$f2
mul.d $f3,$f0,$f1
div.d $f0,$f3,$f1
```



ADD: \$f1 and \$f2 are non-speculative (trailing = leading). \$f0 is allocated and updates leading table but not trailing



MUL: \$f1 is non-speculative. \$f0 is “renamed” using leading table, \$f3 is allocated and updates leading table but not trailing



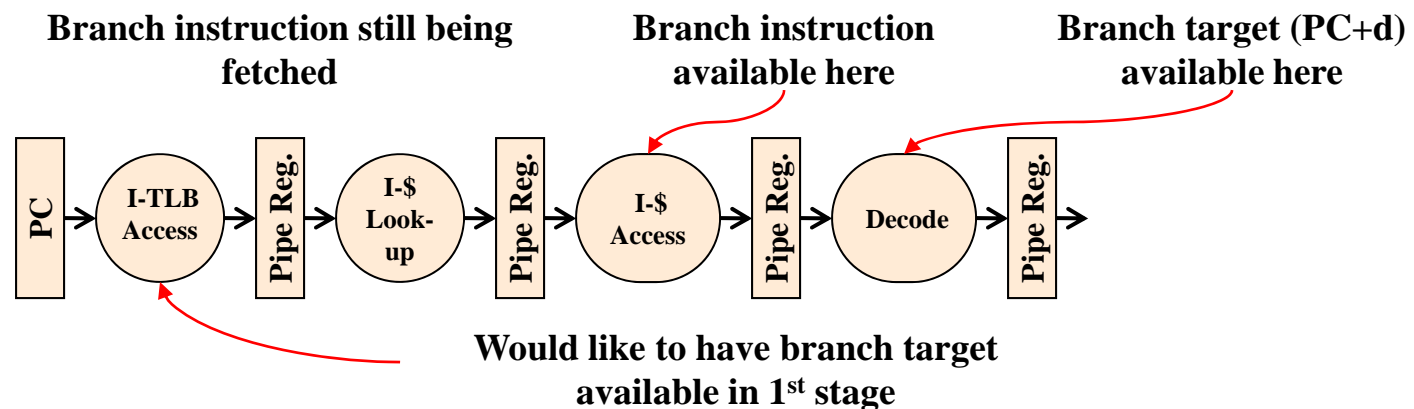
- 1.) Assume add finishes, trailing table is updated to point at latest value of \$f0
- 2.) DIV: \$f3 is speculative while \$f1 is not. \$f0 is allocated a new physical register and thus updates the leading table.

Branch Prediction

- Since basic blocks are often small, wide issue (static or dynamic) processors may encounter a branch every 1-2 cycles
- We not only need to know the outcome of the branch but the target of the branch
 - **Branch target:** Branch target buffer (cache)
 - **Branch outcome:** Static (compile-time) or dynamic (run-time / HW assisted) prediction techniques
- To keep the pipeline full and make speculation efficient and not wasteful, the processor needs accurate predictions

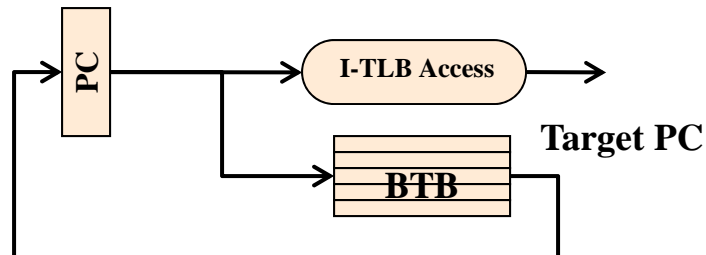
Branch Target Availability

- Branches perform $PC = PC + \text{displacement}$ where displacement is stored as part of the instruction
- Usually can't get the target until after the instruction is completely fetched (displacement is part of instruction)
 - May be 2-3 cycles in a deeply pipelined processor
ex. [I-TLB, I-Cache Lookup, I-Cache Access, Decode]
 - If a 4-way superscalar and 3 cycle branch penalty, we throw away 12 instructions on a misprediction
- Key observation: Branches always branch to same place (target is constant for each branch)



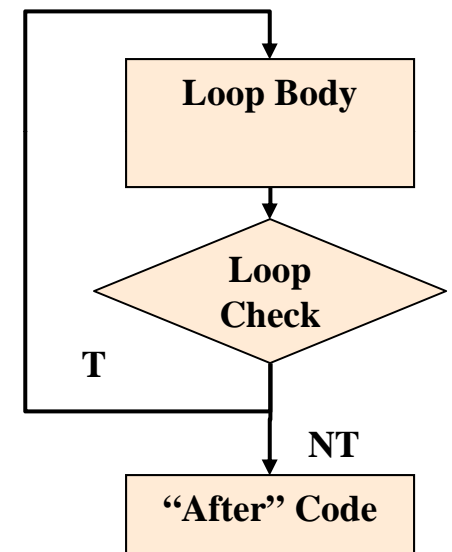
Branch Target Buffer

- Idea: Keep a cache (branch target buffer / BTB) of branch targets that can be accessed using the PC in the 1st stage
 - Cache holds target addresses and is accessed using the PC (address of instruction)
 - First time a branch is executed, cache will miss, and we'll take the branch penalty but save its target address in the cache
 - Subsequent accesses will hit (until evicted) in the BTB and we can use that target if we predict the branch is taken
- Note: BTB is a “fully-associative” cache (search all entries for PC match)...thus it can't be very large



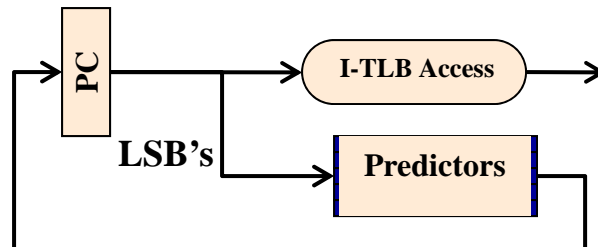
Static Branch Outcome Prediction

- Can be used by static issue machines
- Many branch instructions are biased to a certain outcome (NT or T)
 - Loop termination checks are often taken
 - Example: `for (i=0; i < 100; i++)` results in 99 taken and 1 not taken branch exec.
- Instruction set supports a “hint” being attached to the instruction which the compiler will set
- The HW will then use that hint and fetch instructions down that path



Dynamic Branch Outcome Prediction

- Keep some “history” of branch outcomes and use that to predict the future
- Keep a table indexed by LSBs of PC with the current prediction
- Questions:
 - What history should we use to predict a branch
 - How much history should we use/keep to predict a branch



Local vs. Global History

- What history should we look at?
 - Should we look at just the previous executions of only the particular branch we're currently predicting or at surrounding branches as well
- Local History: The previous outcomes of that branch only
 - Usually good for loop conditions
- Global History: The previous outcomes of the last m branches in time (other branches included)

```
Do {  
    if(x == 2) { ... }  
    if(y == 2) { ... }  
    if(x != y) { ... } // Better:  
                        // Local or Global  
}  
  
while (i > 0); // Better:  
              // Local or Global
```

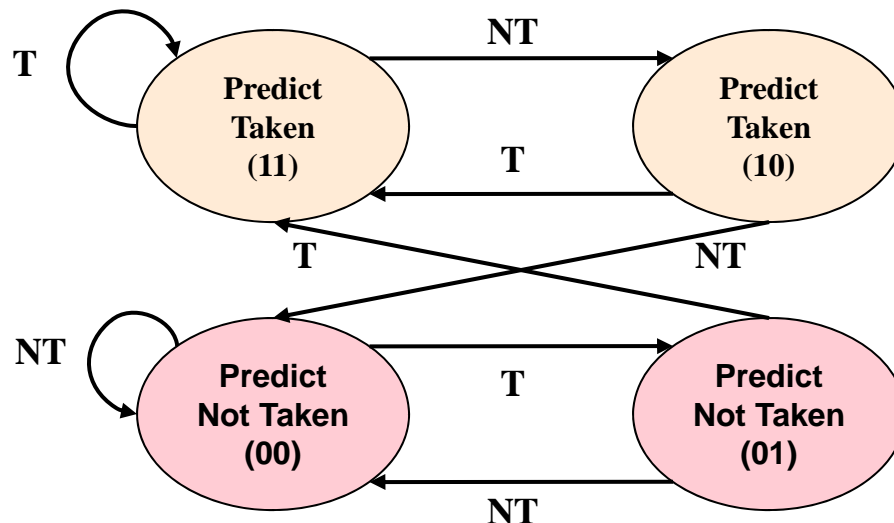
Dynamic Local Predictors

- How much history do we need to keep?
- 1-bit predictor per branch = Last outcome of the branch used to predict next outcome
 - Problem: When wrong once will often be wrong twice
 - Highlighted BNE will be
T, T, ... T, NT, T, T, ... T, NT, T, T, ...
 - NT only every 50 iterations
 - 1-bit predictor will say T when 'bne' is NT, then update to 'NT' and be wrong again the next time when 'bne' is T again

```
                li    $a0,10
LOOP1:          li    $a1,50
LOOP2:          ...
                addi   $a1,$a1,-1
                bne    $a1,$0,LOOP2
                addi   $a0,$a0,-1
                bne    $a0,$0,LOOP1
```


2-bit Predictor

- Solves the problem of 2 mispredictions at the end of a loop
- Keep current prediction (e.g. T) until mispredicted twice in a row (e.g. NT, NT)
 - Require 2 bits for 4 cases of last 2 outcomes
- More than 2-bits does not yield much better accuracy



Assume we start in Predict T state, how many mispredictions will each sequence cause?

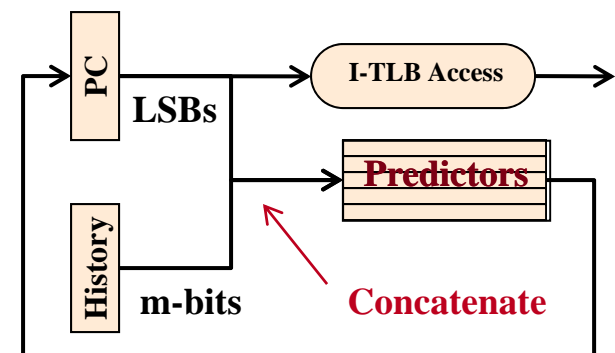
					<u>Mispredicts</u>
1.)	T	T	NT	T	1
2.)	NT	T	NT	NT	3
3.)	NT	NT	T	T	4
4.)	NT	NT	NT	NT	2
5.)	NT	T	NT	T	2

Global (Correlating) Predictor

- Use the outcomes of the last m branches that were executed to select a prediction

```
beq    ...,L1 (T = 1)
      ...
beq    ...,L2 (NT = 0)
      ...
bne    ...,L3 // use history
              // of NT,T to
              // predict
```

- Given last m branches, 2^m possible combinations of outcomes & thus predictions
 - When beq1=NT and beq2=NT, predict 'bne' = T, when beq1=NT and beq2=T, predict 'bne' = NT, etc.
- Branch predictor indexed by concatenating LSBs of PC and m -bits of last m branch outcomes



(m,n) Global Predictors

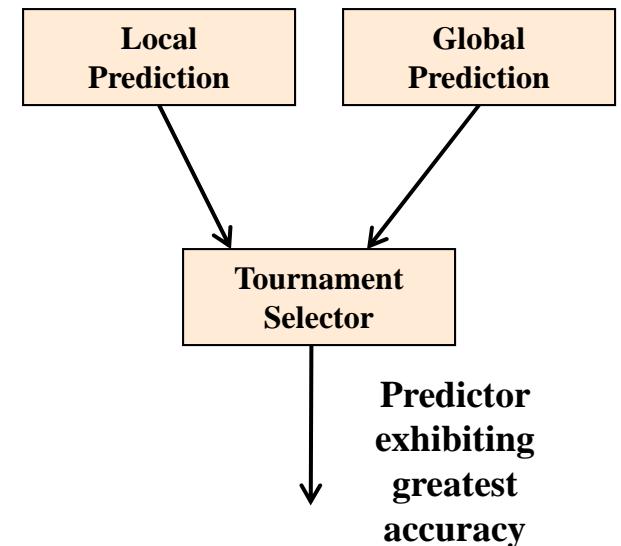
- **m = # of previous branch outcomes**
- **n = # of bits per prediction (1 or 2)**
- **(4,2) Predictor**
 - **History of last 4 branch outcomes (16 combos) maintained and used to index the prediction for this branch**
 - **Each prediction is a 2-bit predictor**

Predictor Tables

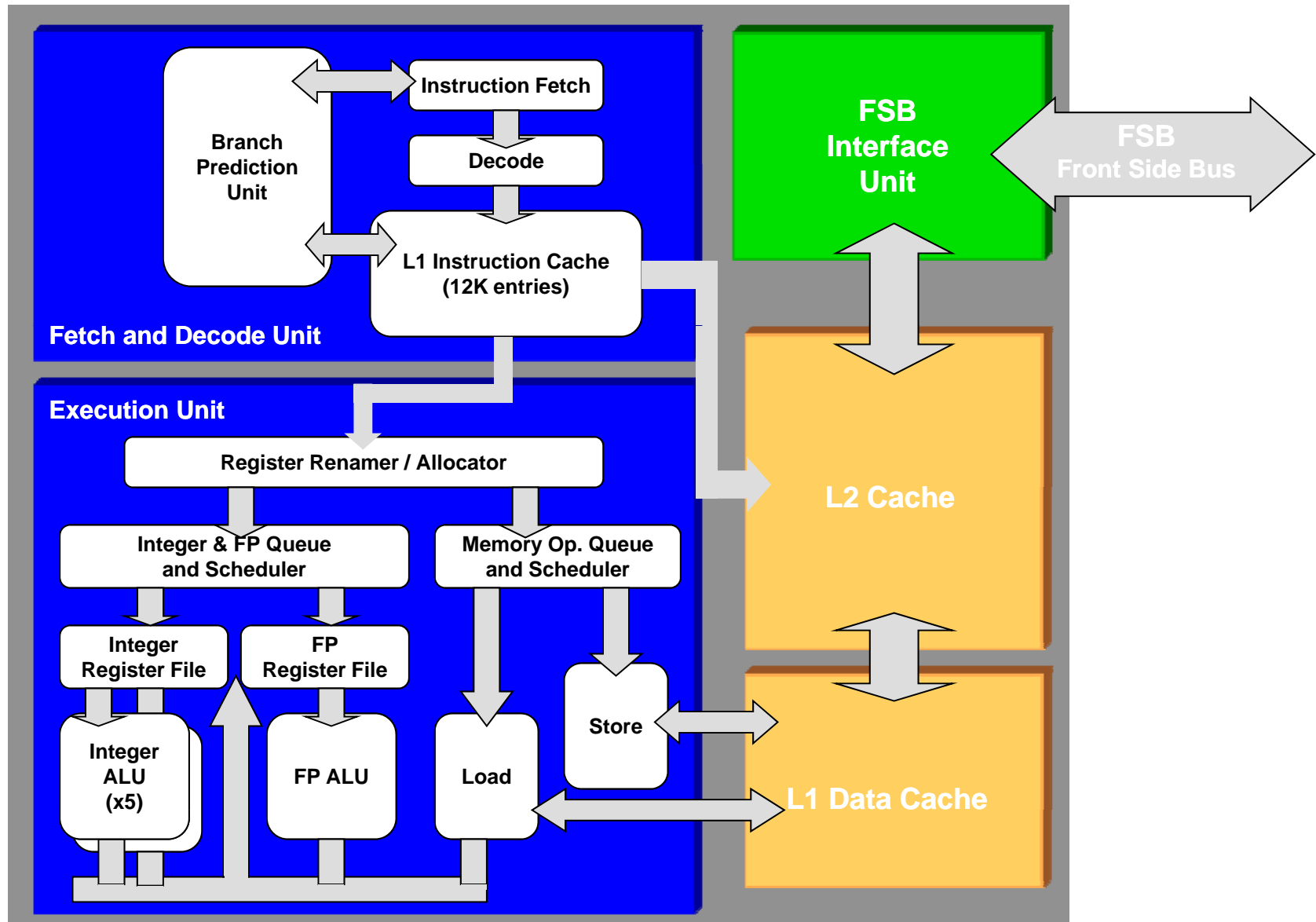
- Table size = Number of entries * Bits per entry
 - Number of entries depends on m-bits of global history and k-LSBs of PC ($2^m * 2^k$ entries)
 - Number of bits per entry = n-bit predictor (1 or 2)
- Example: How many entries would an 8-Kbit predictor table have given a (4,2) predictor? How many LSBs of the PC would be used for indexing?
 - Entries = Size / bits per entry = $8K / 2 = 4K$ entries
 - PC bits = $\log_2(\text{Entries} / 2^m) = \log_2(4K / 16) = \log_2(2^{12} / 2^4) = \log_2(2^8) = 8\text{-bits}$

Tournament Predictor

- Dynamically selects when to use the global vs. local predictor
 - Accuracy of global vs. local predictor for a branch may vary for different branches
 - Tournament predictor keeps the history of both predictors (global or local) for a branch and then selects the one that is currently the most accurate



Pentium 4 Processor



Pros/Cons of Static vs. Dynamic

- **Static**
 - HW can be simpler since compiler schedules code
 - Compiler can see “deeper” in the code to find parallelism
- **Dynamic**
 - Allows for performance increase even for legacy software
 - Can be better at predicting unpredictable branches
 - HW structures do not scale well (ROB, reservation stations, etc.) beyond small sizes
 - Much wasted work (and power)