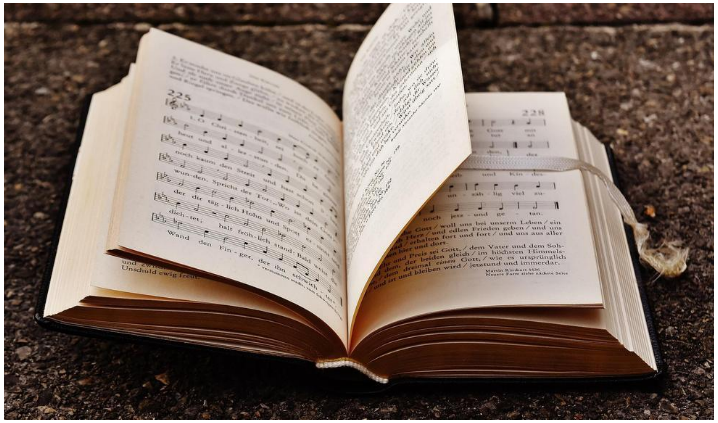


Chapter 5: Alternative Approaches to Issue and Commit

Background Required to Understand this Chapter

- OOO Pipelines
- Wakup/select Mechanism
- Instruction commit

Chapter 4

Contents

1. Load Speculation
2. Replay Mechanisms
3. Simpler Version of an OOO Processor
4. Compiler based Techniques
5. EPIC based Techniques: Intel Itanium

Aggressive Speculation

Branch prediction is one form of speculation

- If we detect that a branch has been mispredicted
- **Solution**: flush the pipeline

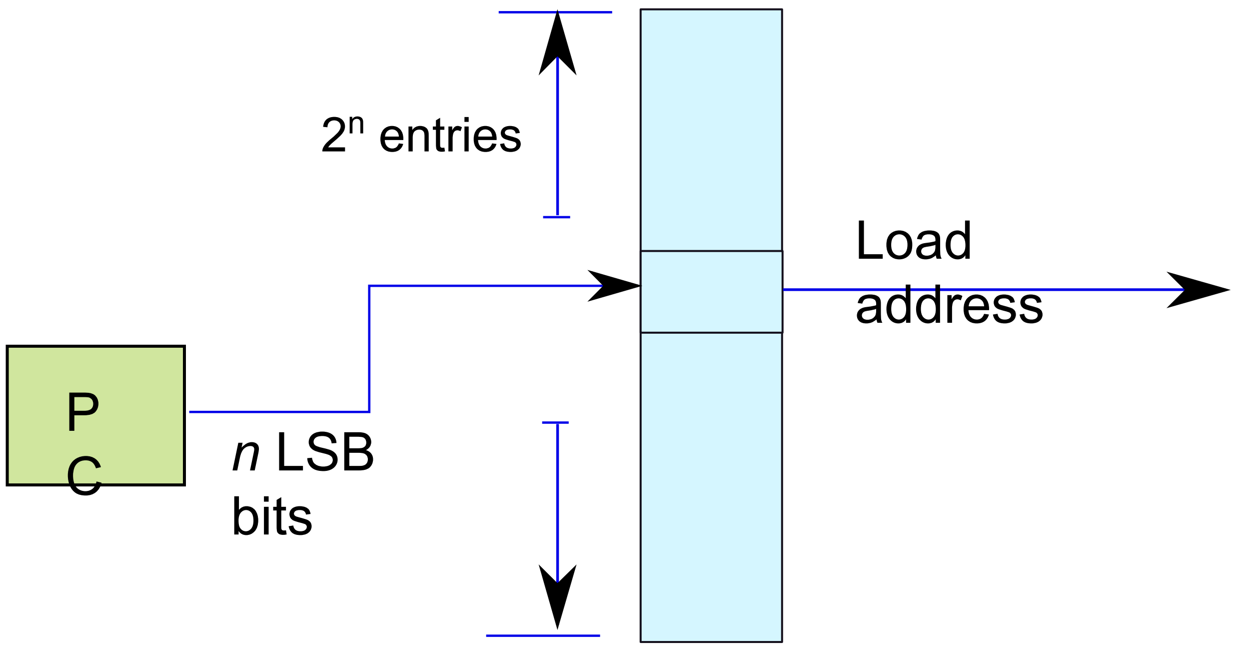
This is not the only form of speculation

- Another very common type: load latency speculation or value speculation
- Assume that a load will hit in the cache
- Speculatively wakeup instructions
- Later on if this is not the case: DO SOMETHING

Types of Aggressive Speculation

- Address Speculation
- Load-Store Dependence Speculation
- Latency Speculation
- Value Prediction

Address Speculation: Predict the memory address of a load or store



Predict last address scheme

- Use a simple predictor

Stride based Address Pattern

C code

```
int sum = 0, arr[10];
for (i=0; i < 10; i++){
    sum += arr[i];
}
```

Assembly code

```
// Let us assume that the base address of arr is in r0
mov r1, 0              // i = 0
mov r2, 0              // sum = 0

.loop: cmp r1, 10      // compare i with 10
beq .exit              // if (r1 == 10) exit
ld  r3, [r0]           // load arr[i] to r3
add r2, r2, r3         // sum += arr[i]
add r0, r0, 4          // increment the memory address
add r1, r1, 1          // increment the loop index
b .loop
```

Predicting the Stride

Last address (A)	Stride (S)	Pattern (P)

- Last address (A): The memory address computed the last time the instruction with this PC was executed.
- A stride-based access pattern is followed if:
  current address – last address = S
- Then we set the pattern bit, P
- Alternatively, if P is set, we predict the next address to be
  - $A + S$

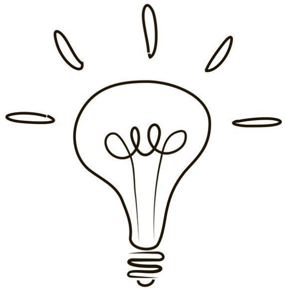
Load-Store Dependence Speculation

Predict a collision (same memory address) between a load and a store

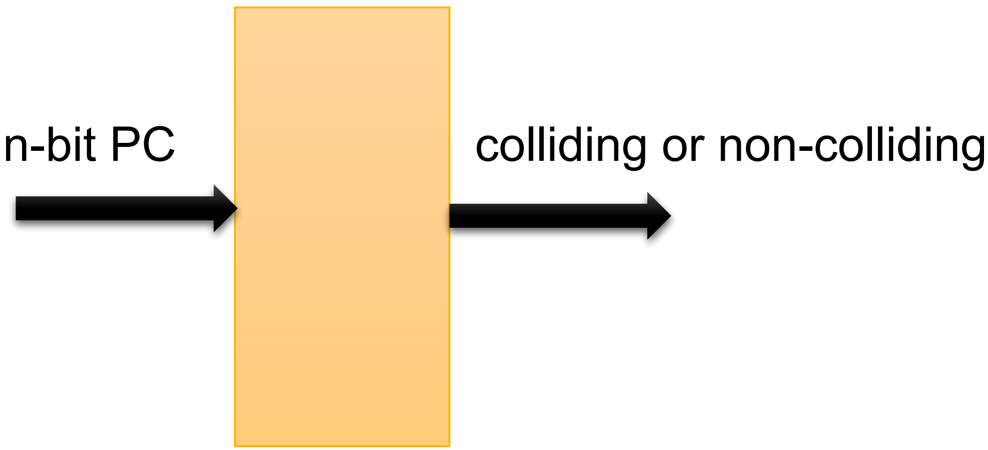
If there are no collisions, send the load directly to the cache.

Forward values across unresolved stores.

Collision History Table



- Loads show consistent behavior
  - They are either colliding or non-colliding



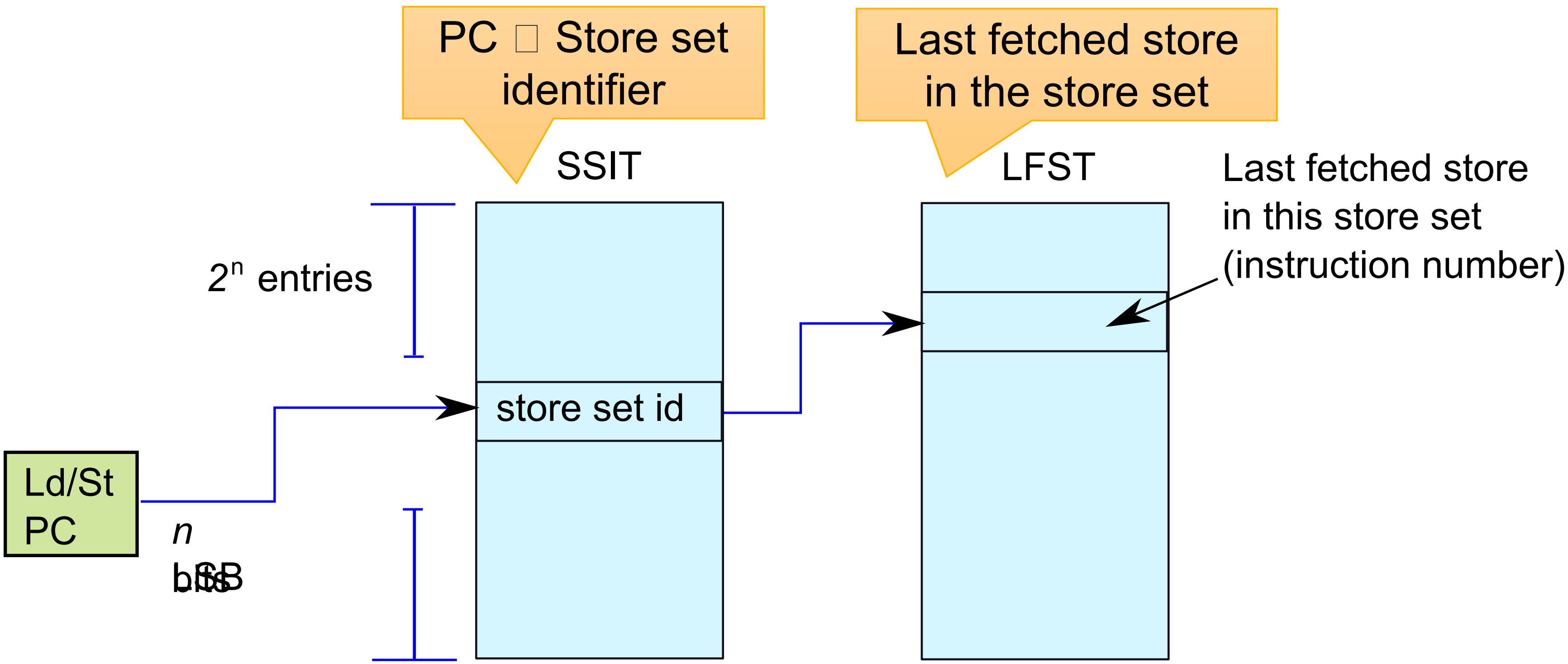
Collision History Table (CHT)

Using the CHT

- When we compute the address of a load
  - We access the CHT
- If it is predicted to be colliding
  - Wait for all prior stores to be resolved
- Else
  - Send the load to the d-cache
- Once the address is resolved
  - Update the CHT, recover the state (if necessary)

We can augment it with the store□load distance (D). A load waits till there are less than D entries before it in the LSQ.

Store Sets



Explicitly remember load-store dependences

Basic Idea

- For every load, we have an associated store set
  - Stores that have forwarded values to it in the past
- A store may be a part of a single store set

Load

1. Read the store set id
2. Get the instruction number of the latest store (*S*) from the LFST
3. The load waits for store *S* to get resolved and then receives the forwarded data.

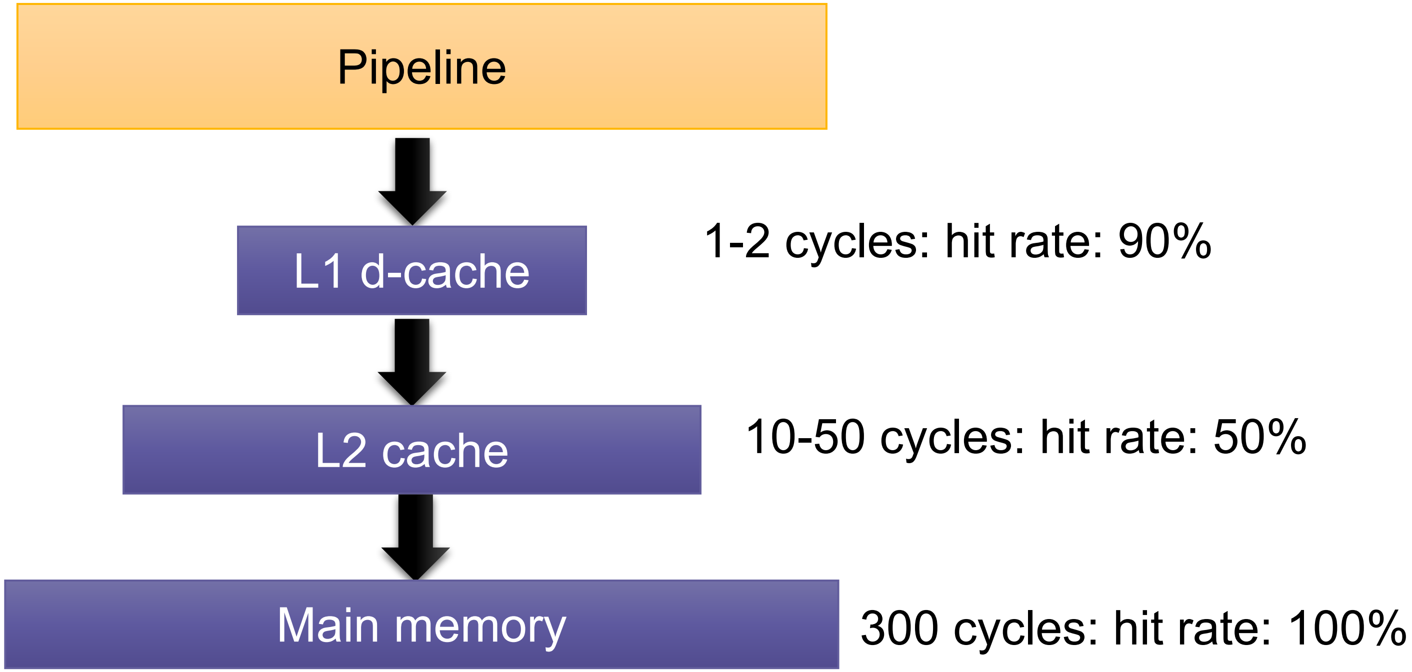
Store

1. Read the store set id
2. Set the instruction number of the current store in the corresponding entry of the LFST.
3. Can be used to speculatively forward data to loads in its store set.

Whenever we detect a load-store dependence, we update the SSIT and LFST

Load Latency Speculation

- A load might hit in the L1 cache (2 cycles) or might go to the lower levels of the memory system.
- We don't know for sure

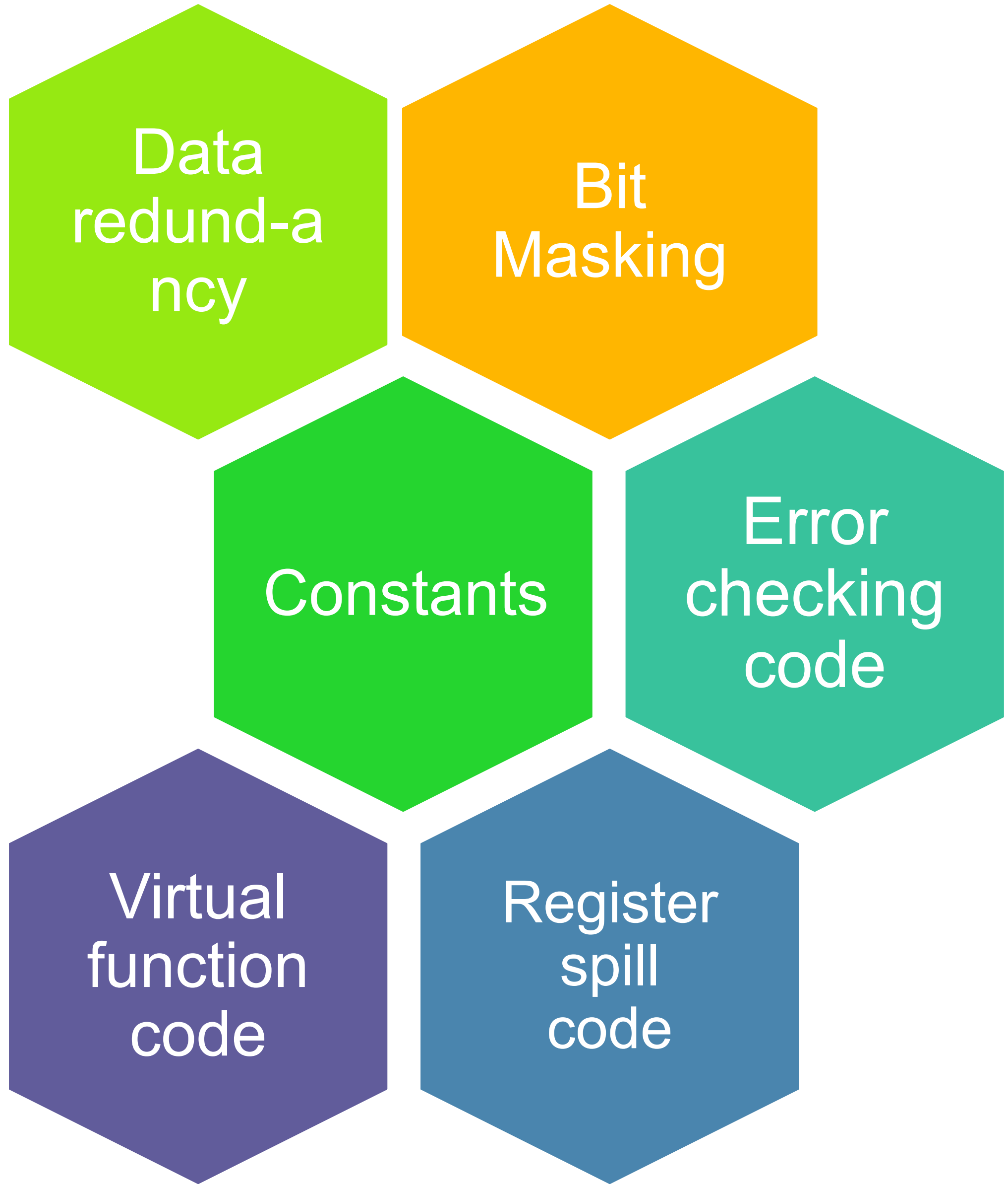


Make a guess

"I bet you can't guess what it is."

- For load instructions, predict if it will hit in the data cache or not. If it will, do an early broadcast.
- Design a hit-miss predictor. Same idea as branch predictors.

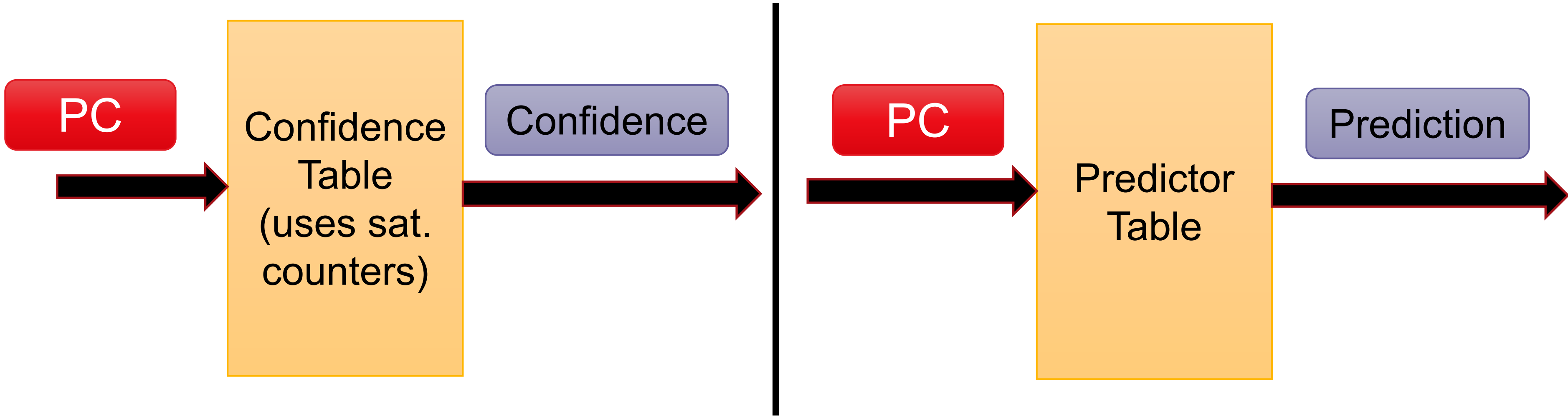
Value prediction: Why are values predictable?



Value Predictor

- Last value
- Stride based
- Based on profiling results

Using an additional predictor for confidence



- First, use the confidence table to find out if it makes sense to predict
- Simultaneously, make a prediction using a predictor table (value, memory dependence, ALU result)
- Predictor table can contain 1 value, or the last ***k*** values
- Make a prediction, and use it if it has high confidence
- Update both the tables when the results are available
- If needed recover with a replay/flush mechanism

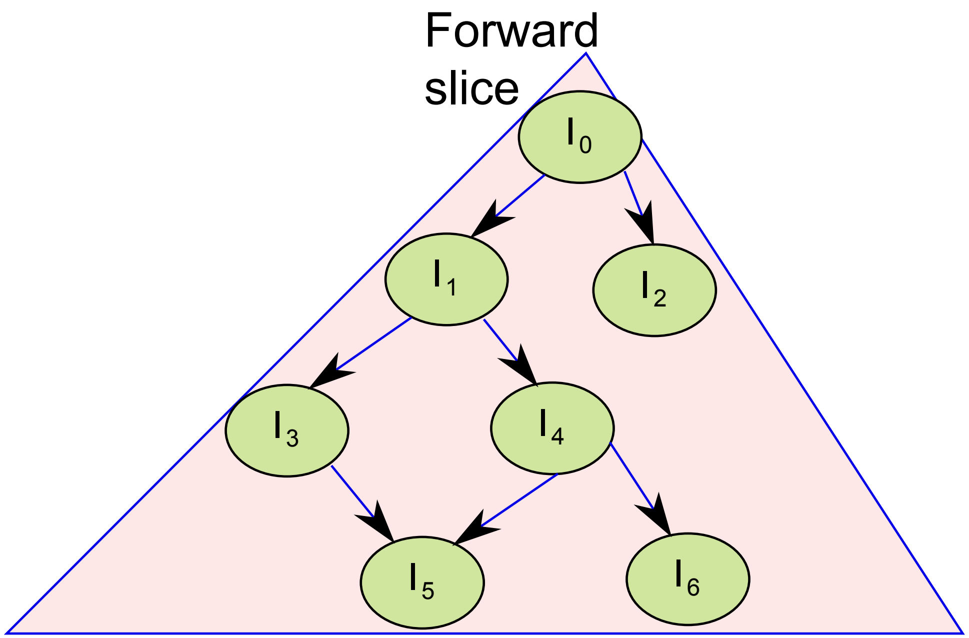
Contents

1. Load Speculation
2. Replay Mechanisms
3. Simpler Version of an OOO Processor
4. Compiler based Techniques
5. EPIC based Techniques: Intel Itanium

Replay

- Flushing the pipeline for every misspeculation is not a wise thing
- Instead, flush a part of the pipeline (or only those instructions that have gotten a wrong value)
- Replay those instructions once again (after let's say the load completes its execution)
- When the instructions are being replayed, they are guaranteed to use the correct value of the load
- Identify and replay the forward slice □

Forward Slice of Instruction I_0



A forward slice contains an instruction's consumers, its consumers and so on.

Non-Selective Replay

Trivial Solution: Flush the pipeline between the dispatch and execute stages

Smarter Solution

- It is not necessary to flush all the instructions between the schedule and execute stages
- Try to reduce the set of instructions
- Define a window of vulnerability (WV) for *n* cycles after a load is selected. A load should complete within *n* cycles if it hits in the d-cache and does not wait in the LSQ
- However, if the load takes more than *n* cycles, we need to do a replay

Example

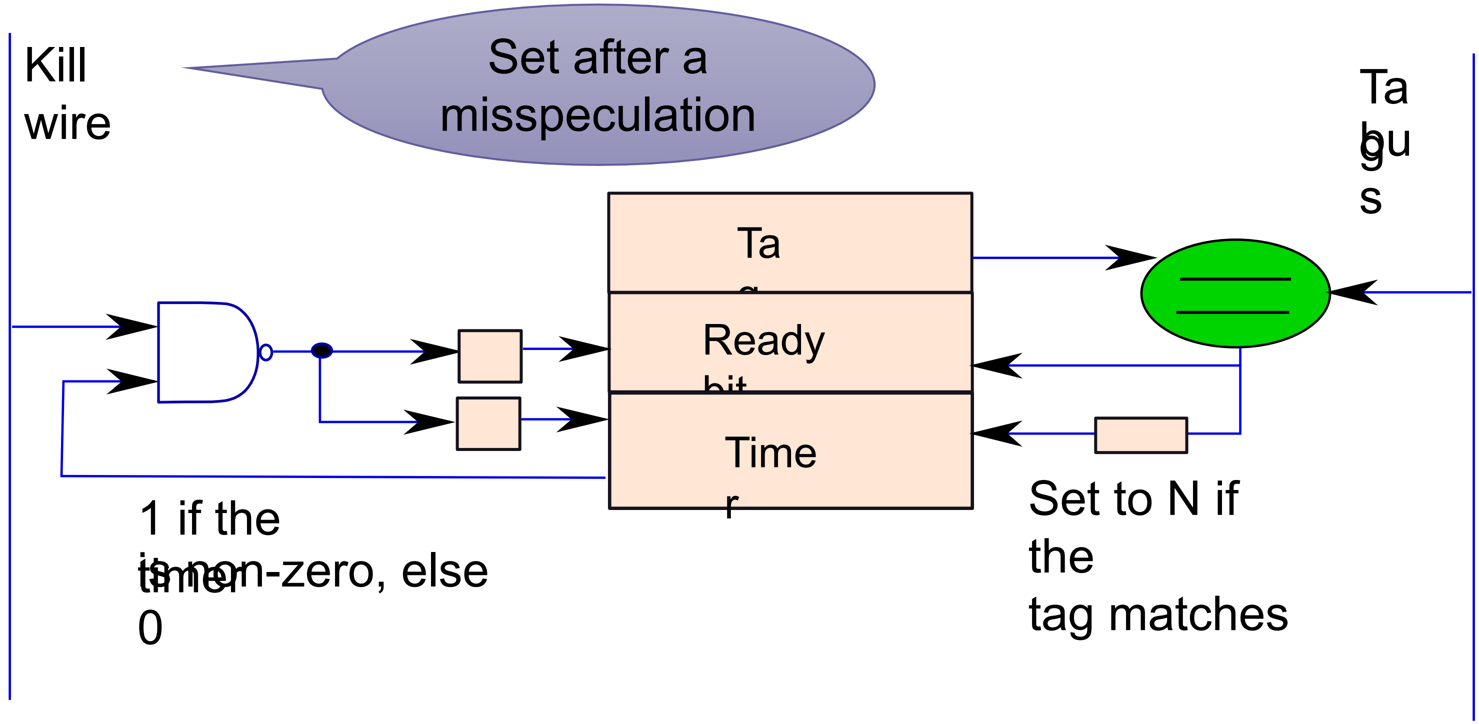
Predict the value

1: ld r1, [r2]
2: add r4, r1, r3
3: add r5, r6, r7
4: add r8, r9, r10

squash them

- Let us say that instructions 2, 3, and 4 had one operand waking up in the WV of instruction 1
- If there is a misspeculation, all three instructions get squashed
- Instruction 1 gets reissued with the correct value later

Instruction Window Entry



- When an operand becomes ready, we set its timer to *n*
- Every cycle it decrements (count down timer)
- Once it becomes 0, we can conclude that this instruction will not be squashed

More about Non-Selective Replay

- We attach the expected latency with each instruction packet as it flows down the pipeline

Wherever there is an additional delay (such as a cache miss)

- Time for a replay
- Set the kill wire

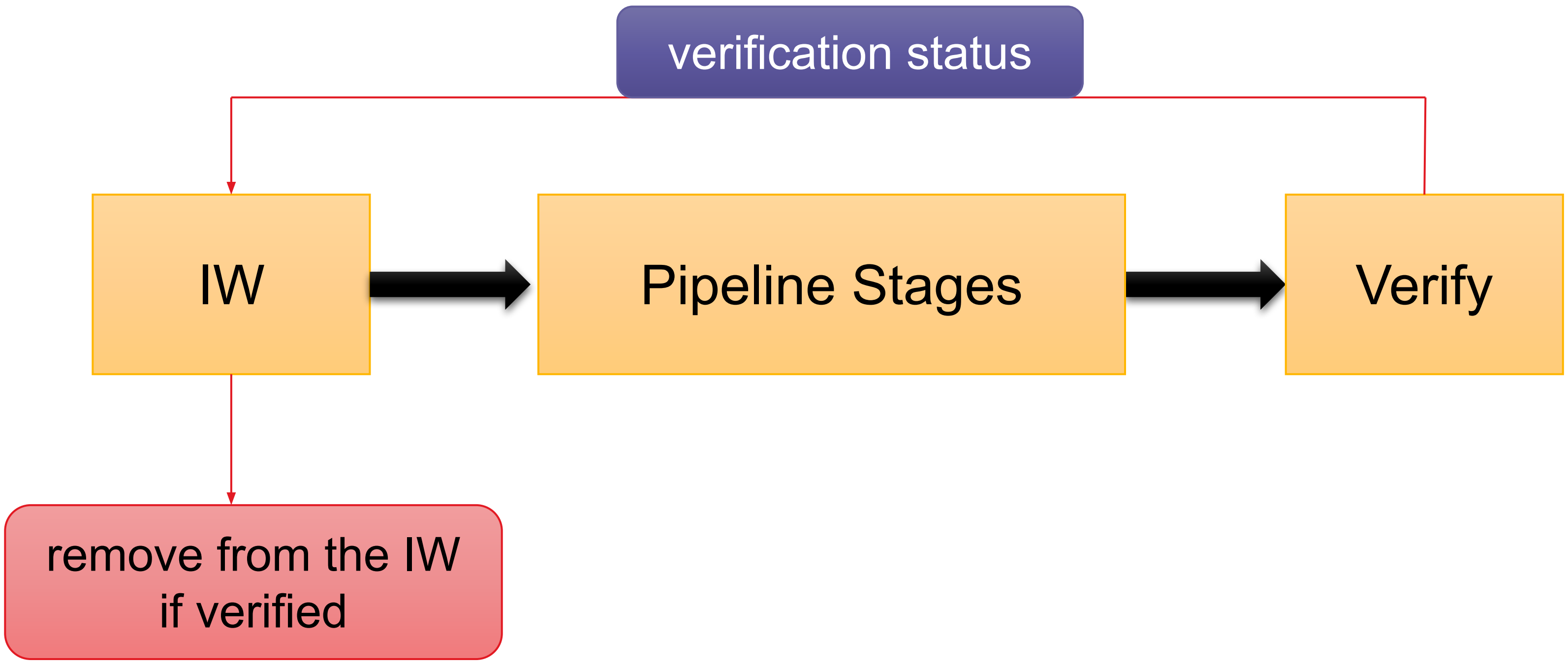
- Each instruction window entry that has a non-zero timer, resets its ready flag

We now have a set of instructions that will be replayed

Methods of replaying instructions

Two methods of replaying

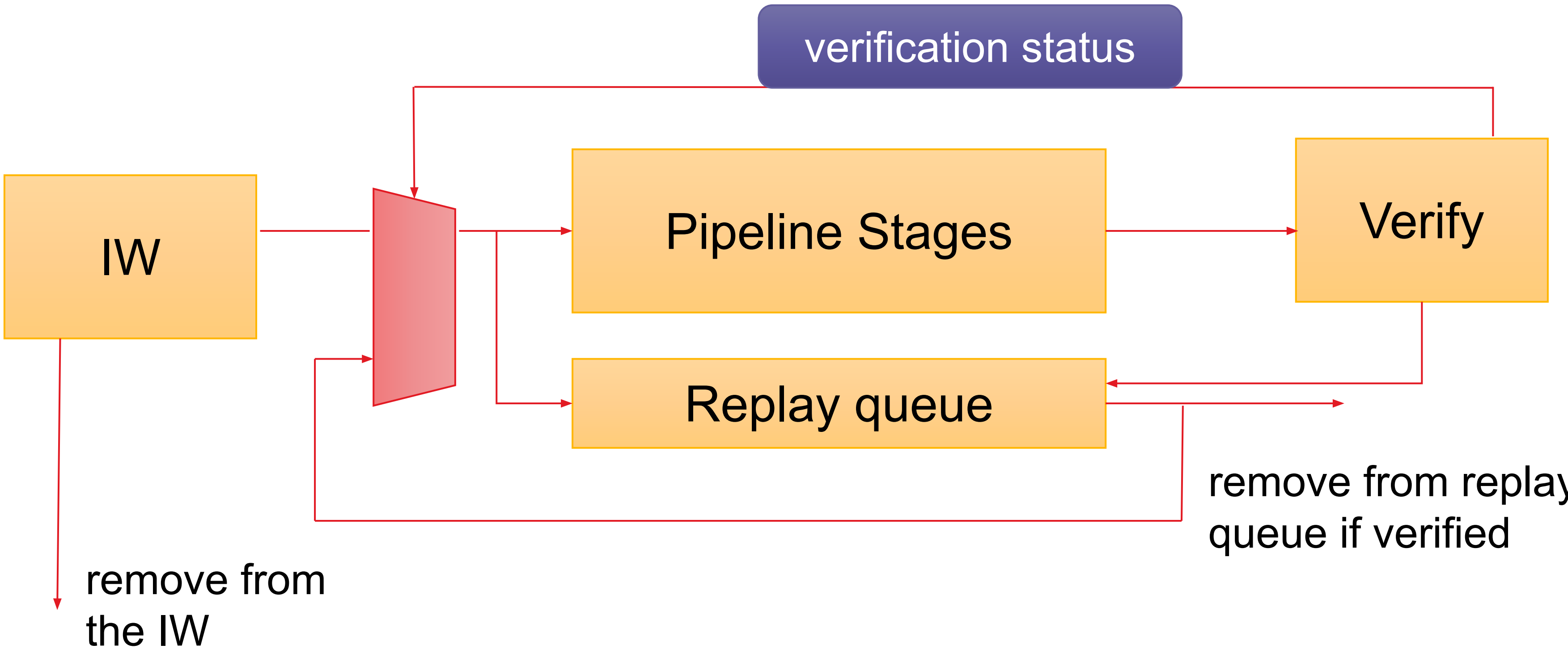
Method 1: Keep instructions that have been issued in the issue queue (see reference)



Two methods of replaying - II

Method 2:

- Move the instructions to a dedicated replay queue after issue
- Once an instruction is verified, remove it from the replay queue



Orphan Instructions

```
ld r1, 8[r4]
add r2, r1, r1
sub r4, r3, r2
xor r5, r6, r7
```

- Assume that the *load* instruction misses in the L1 cache
- The *add, sub,* and *xor* instructions will need to be squashed, and replayed
- For the *add* and *sub* instructions, tag will be broadcast

What about the *xor* instruction?

Say that r6’s ready bit was forcefully set to 0

Orphan Instructions - II

Impractical Method

- Keep track of squashed instructions.
- Re-broadcast tags of orphan instructions.
  - □ We need to dynamically detect which instructions are orphans.

Better Approach

- Let the orphan instruction reach the head of the ROB
- Execute and commit it.

Delayed Selective Replay

- Let us now propose an idea to replay only those instructions that are in the forward slice of the misspeculated load
- Let us extend the non-selective replay scheme
- At the time of asserting the kill signal, plant a poison bit in the destination register of the load
- Propagate the bit along the bypass paths and through the register file
- If an instruction reads any operand whose poison bit is set, then the instruction's poison bit and its destination register are also set.

- When an instruction finishes execution □

Delayed Selective Replay - II

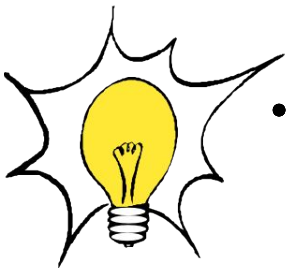
When an instruction finishes execution

- Check if its poison bit is set.
- If yes, squash it
- If no, remove the instruction from the IW (it is verified to be correct)

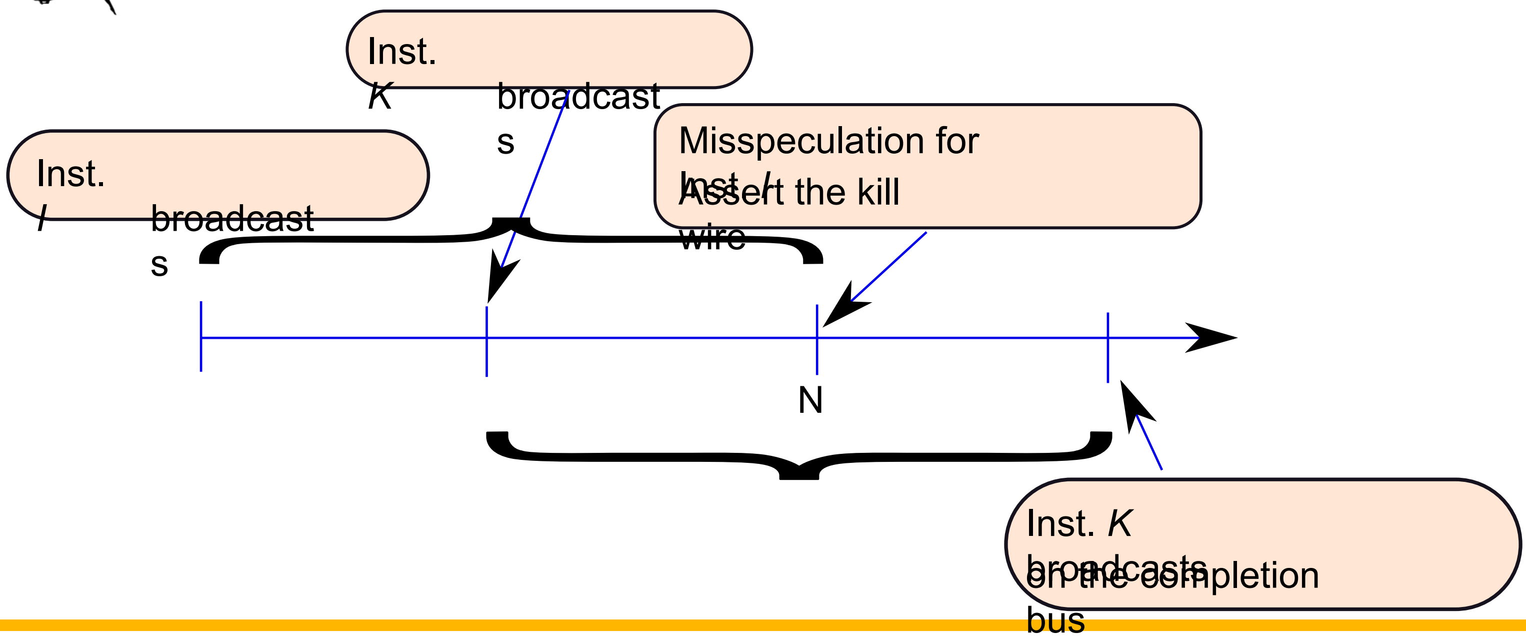
Issues with this scheme

- It is effective, but assumes that we know the value: n
- This might not be possible all the time
- Instructions in the WV that have not been issued might become orphans

Orphan Instructions



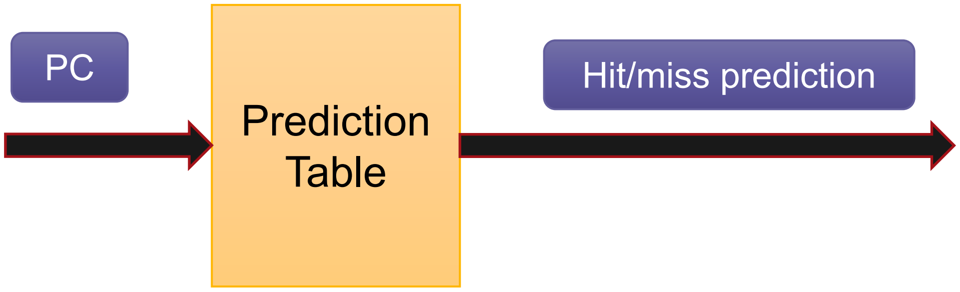
- We can always wait for the instruction to reach the head of the ROB.
- Another scheme: Let's say instruction *J* was orphaned because one of its operands (woken up by inst. *K*) was reset back to a non-ready state.
- Instruction *K* will later come back to rescue *J*, via broadcasting on the completion bus.



Token Based Selective Replay

Let us use a pattern found in most programs:

- Most of the misses in the data cache are accounted for by a relatively small number of instructions
- 90/10 thumb rule □ 90% of the misses are accounted for by 10% of instructions
- Predictor □ Given a PC, predict if it will lead to a d-cache miss
- Use a predictor similar to a branch predictor at the fetch stage



After Predicting a d-cache Miss

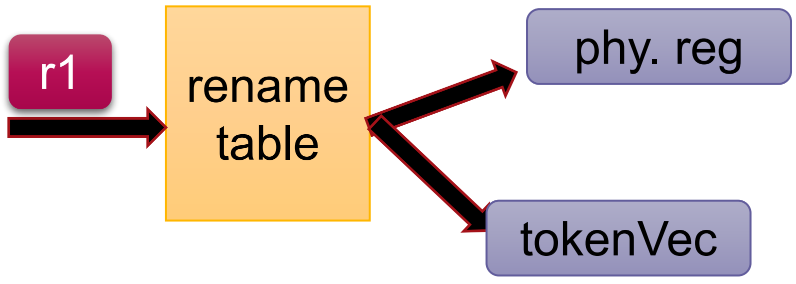
Instructions that are predicted to miss, will have a non-deterministic execution time (most likely) and lead to replays (set *S1*)

Other instructions will not lead to replays (most likely) (set *S2*)

Let us consider an instruction in set S1

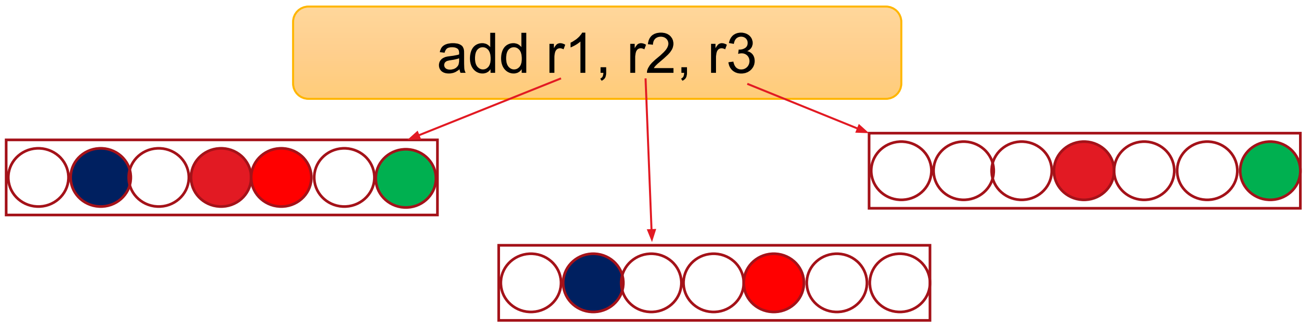
- At decode time, let the instruction collect a free token
- Save the id of the token in the instruction packet
- Example: assume the instruction: *ld r1, 4[r4]* is predicted to miss
  - Save the id of the token in the instruction packet of this instruction
- Say that the instruction gets token #5
  - This instruction is the token head for token #5
- Let us propagate this information to all the instructions dependent on the load
  - If this load fails, all the dependent instructions fail as well

Structure of the Rename Table



- If an instruction is a token head, we save the id of the token that it owns in the instruction packet
- Assume we have a maximum of *N* tokens.
  - *tokenVec is an N-bit vector*
- For the token head instruction, if it owns the ith token, set the ith bit to true in *tokenVec*
- Tokens are propagated the same way as poison bits

While reading the rename table ...

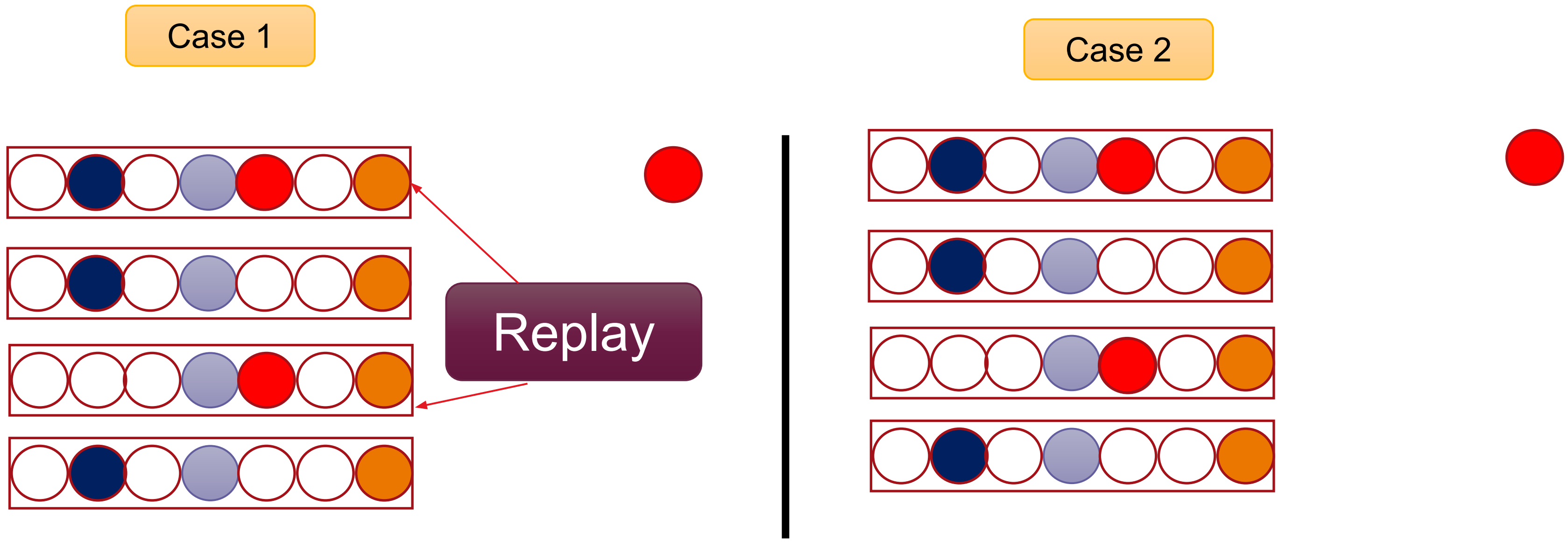


- Read the tokenVecs of the source operands
- Merge the tokenVecs of the source operands
- Save the merged tokenVec for the destination register (in the rename table)

After execution

After the token head instruction completes execution, see if it took additional cycles (verification of latency speculation)

- If YES, broadcast the token id to signal a replay (Case 1)
- If NO, broadcast the token id to all the instructions. They can turn the corresponding bit off. (Case 2)



Instructions in S2

- Assume an instruction that was not predicted to miss actually misses
- No token is attached to it
- Wait till it reaches the head of the ROB; flush the pipeline.

Contents

1. Load Speculation
2. Replay Mechanisms
3. Simpler Version of an OOO Processor
4. Compiler based Techniques
5. EPIC based Techniques: Intel Itanium

A Simpler Design

Physical Register File (PRF) based design

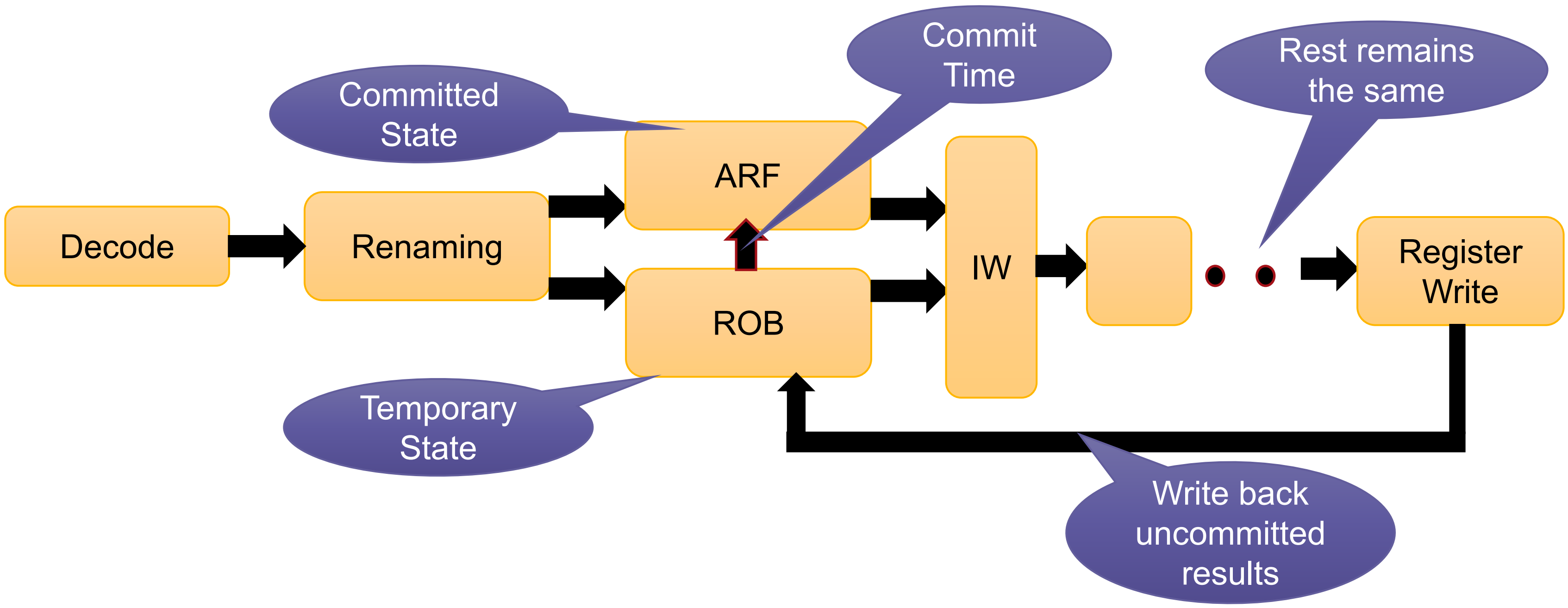
- (+) Fast and efficient
- (−) Physical register management is onerous
- (−) State recovery is complex

Architectural Register File (ARF) based design

- ✓ Have a dedicated architectural register file that stores the committed state
- ✓ Enhance the ROB to store uncommitted values

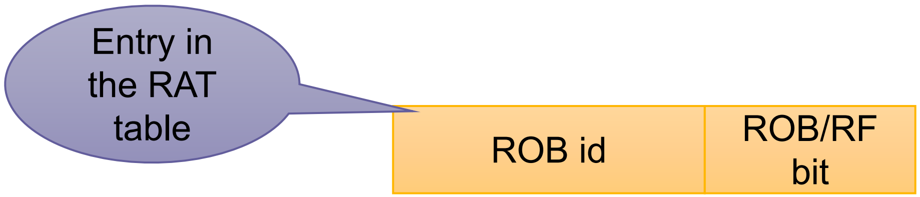
Let us now look at a different kind of OOO processor

- Instead of having a physical register file, let us have an architectural register file(ARF)
- A 16-entry architectural register file that contains the committed architectural state.

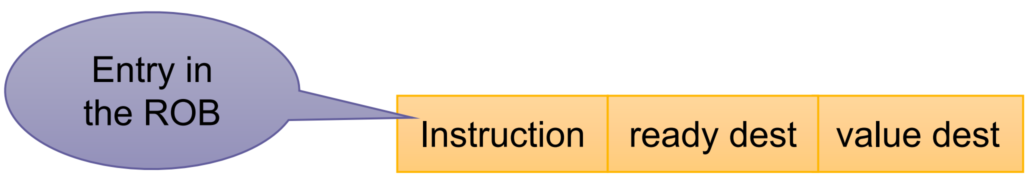


Changes to renaming

Entry in the RAT table



- ROB/RF bit □ 1 (value in the ROB), 0 (value in the ARF)
- Use the ROB if the ROB/RF bit indicates that the value might be there in the ROB
- Entry in the ROB: (ready bit indicates if the value is in the ROB (1) or being generated in the pipeline (0))



Changes to Dispatch and Wakeup

Each entry in the IW now stores the values of the operands

- Reason: We will not be accessing the RF again

What is the tag in this case?

- It is not the id of the physical register.
- It is the id of the ROB entry.

What else?

- Along with the tag, we need to broadcast the value of the operand, if we will not get the value from the bypass network
- This will make the circuit slower

Changes to Wakeup, Bypass, Reg. Write and Commit

- We can follow the same speculative wakeup strategy and broadcast a tag (in this case, id of ROB entry) immediately after an instruction is selected. Tags+values are broadcast when the instruction is in the write-back stage.
- Instructions directly proceed from the select unit to the execution units
- All tags are ROB ids.
- After execution, we write the result to the ROB entry
- Commit is simple. We always have the architectural state in the ARF.
- We just need to flush the ROB.

PRF based design vs ARF based design

➕ points in the PRF based design

- A value resides in only a single location (PRF). Multiple copies of values are never maintained. In a 64-bit machine, a value is 64 bits wide.
- Each entry in the IW is smaller (values are not saved).
- The broadcast also uses 7-bit tags

➖ Restoring state is complicated

➕ points in the ARF based design

- Recovery from misspeculation is easy
- We do not need a free list

➖ Values are stored at multiple places (ARF, ROB, IW)

Contents

1. Load Speculation
2. Replay Mechanisms
3. Simpler Version of an OOO Processor
4. Compiler based Techniques
5. EPIC based Techniques: Intel Itanium

Compiler based Optimizations

Can the compiler optimize the code?

- Reduce code size
- Increase ILP
- Reduce slow instructions with fast ones

Constant Folding

```
int a = 4 + 6;
int b = a * 2;
int c = b * b;
```

We can directly replace *a* with 10, *b* with 20, and *c* with 400

Strength Reduction

```
int b = a * 8;
int d = c / 4;
int e = b * 12;
```

slow

```
int b = a << 3;
int d = c >> 2;
int e = b << 2 + b << 3;
```

fast

Common Subexpression Elimination

```
int c = (a + b) * 10;
int d = (a + b) * (a + b);
```

```
int t1 = a + b;
int c = t1 * 10;
int d = t1 * t1;
```

- Each line in the second example corresponds to one line of assembly code.
- We do not compute (a+b) many times.

Dead Code Elimination

```
int main (){
    int a=0, b=1, c;
    int vals[4];

    printf ("Hello World\n");
    c = a + b;
    vals[1] = c;
}
```

Dead code

Silent Stores

```
int arr[5], a, b, c;

arr[1] = 3;
a = 29;
b = a * arr[0];
arr[1] = 3;            /* Not required */
printf ("%d \n", (arr[1] + b));
```

Silent store

- Silent stores write the same value that is already present

Loop Based Optimizations

Loop Invariant based Code Motion

Original

```
for (i=0; i<N; i++){
    val = 5;
    A[i] = val;
}
```

Loop Invariants Moved

```
val = 5;
for (i=0; i<N; i++){
    A[i] = val;
}
```

- There is no point setting (val = 5) repeatedly.

Induction Variable based Optimization

Original

```
for (i=0; i<N; i++){
    j = 6*i;
    A[i] = B[j] + C[j];
}
```

Induction variable

Optimized

```
j = -6 ;
for (i=0; i<N; i++){
    j = j + 6;
    A[i] = B[j] + C[j];
}
```

Replace a multiply with an add

- An add operation is faster than a multiply operation. Hence, it makes sense to replace multiplies with adds.

Loop Fusion

Original

```
for (i=0; i<N; i++)   /* Loop 1 */
    A[i] = 0;

for (i=0; i<N; i++)   /* Loop 2 */
    B[i] = 0;
```

Fuse the loops

Optimized

```
for (i=0; i<N; i++){  /* Loop 1 */
    A[i] = 0;
    B[i] = 0;
}
```

- Loop fusion reduces the instruction count and the number of branches significantly

Loop Unrolling - I

```
for (i=0; i<10; i++){
    sum = sum + i;
}
```

Original loop

Assembly code

```
mov r0, 0          /* sum = 0 */
mov r1, 0          /* i = 0 */

.loop:
cmp r1, 10
beq .exit          /* if (i == 10) exit */
add r0, r0, r1     /* sum = sum + i */
add r1, r1, 1      /* i = i + 1 */
b .loop            /* next iteration */

.exit:
```

Loop Unrolling - II

C code

```
for (i=0; i<10; i+=2){
     sum = sum + i + (i+1);
}
```

Assembly code

```
mov r0, 0          /* sum = 0 */
mov r1, 0          /* i = 0 */

.loop:
cmp r1, 10
beq .exit          /* if (i == 10) exit */

add r0, r0, r1     /* sum = sum + i */
add r1, r1, 1      /* i = i + 1 */
add r0, r0, r1     /* sum = sum + i */

add r1, r1, 1      /* i = i + 1 */
b .loop            /* next iteration */

.exit:
```

Advantage: fewer total instructions and specifically fewer branch instructions

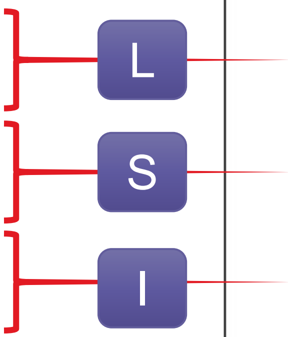
Software Pipelining

C code

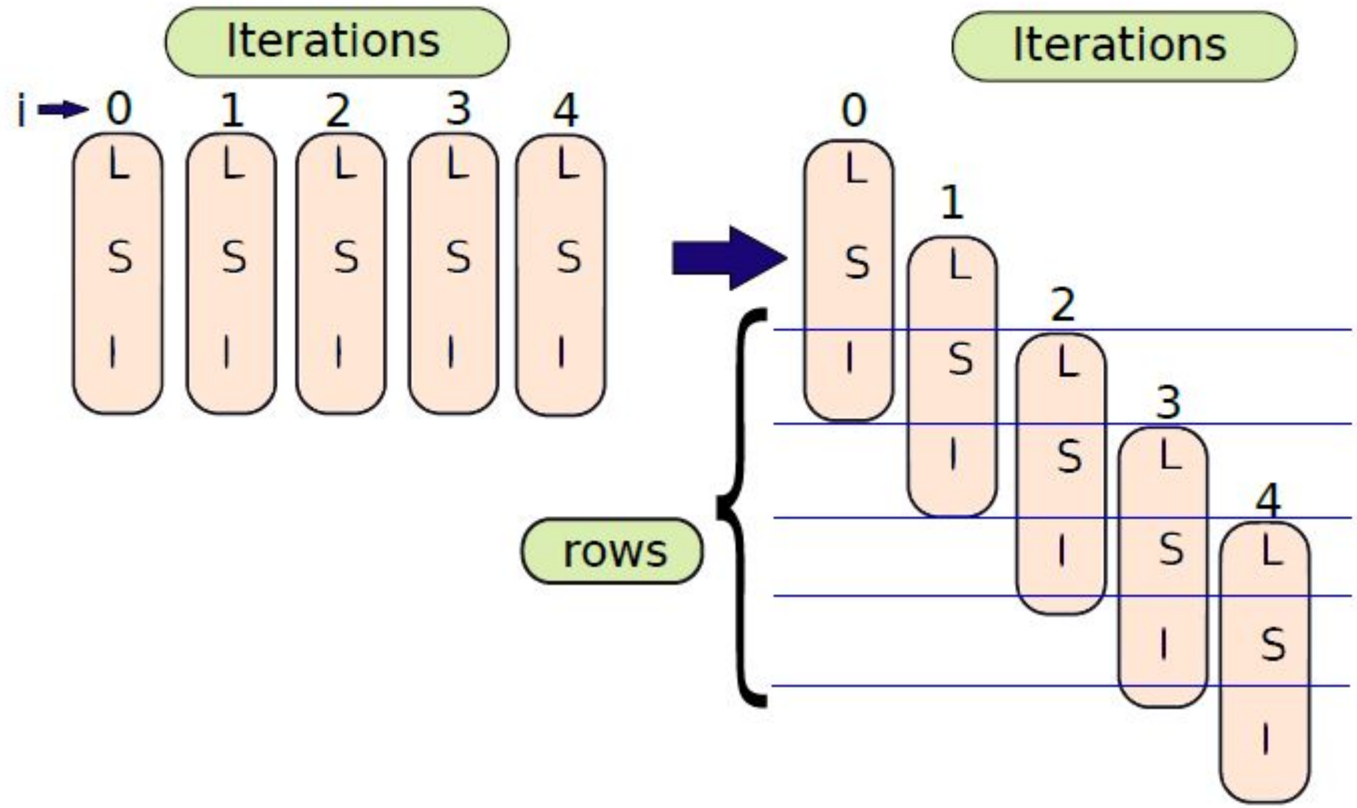
```c
int A[300], B[300];
...
for(i=0; i<300; i++){
    A[i] = B[i];
}
```

Assembly code

```
/* Assume the base address of A is in r0
        and B is in r1 */
mov r2, 0              /* i = 0 */
mov r10, 0             /* offset = 0 */

.loop:
    cmp r2, 300        /* termination check */
    beq .exit

    add r3, r1, r10 /* r3 = addr(B) + offset */
    ld r5, 0[r3]       /* r5 = B[i] */

    add r4, r0, r10 /* r4 = addr(A) + offset */
    st r5, 0[r4]       /* A[i] = r5 (= B[i]) */

    add r2, r2, 1      /* i = i + 1 */
    lsl r10, r2, 2     /* offset = i * 4 */

    b .loop

.exit:
```

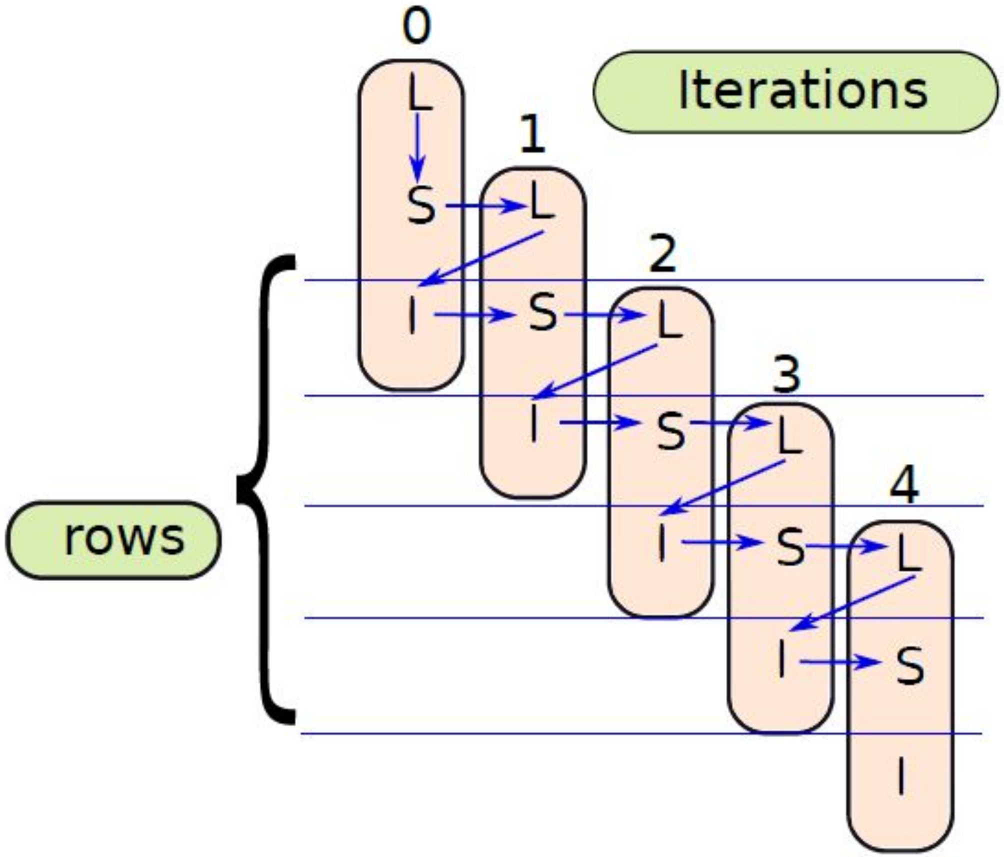


Visualization of the Execution Process



We can create our loops differently. Execute instructions from different iterations.

Can we execute instructions in this order?



Order of operations in a row

I^0 □ S^1 □ L^2

I^1 □ S^2 □ L^3

I^2 □ S^3 □ L^4

Treat each row as a pipeline stage. Execute instructions from different iterations roughly at the same time.

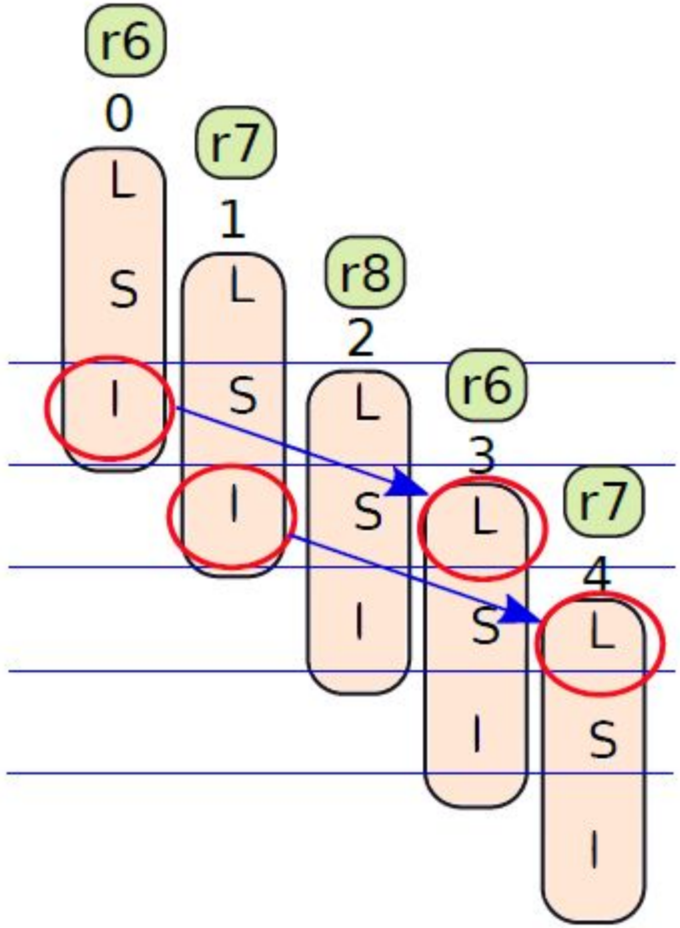
Advantages of Software Pipelining

- Consider this order:

  I^0 □ S^1 □ L^2 □ I^1 □ S^2 □ L^3 □ I^2 □ S^3 □ L^4

- The gap between the L, S, and I blocks is one block
  - This means that we can absorb delays
  - We can accommodate multi-cycle loads without stalls
  - The blocks I, S, and L can possibly be executed concurrently
- There is a problem
  - How do we execute three blocks (belonging to different iterations) possibly concurrently?
  - **Solution**: Use different loop iterators

Different Loop Iterators: Group of 3 iterations



Code with Different Loop Iterators

```
/* First Row */
add r6, r6, 3      /* I0 */
lsl r10, r6, 2

add r4, r0, r11    /* S1 */
st r5, 0[r4]

add r3, r1, r12    /* L2 */
ld r5, 0[r3]

/* Second Row */
add r7, r7, 3      /* I1 */
lsl r11, r7, 2

add r4, r0, r12    /* S2 */
st r5, 0[r4]

add r3, r1, r10    /* L3 */
ld r5, 0[r3]

/* Third Row */
add r8, r8, 3      /* I2  */
lsl r12, r8, 2

add r4, r0, r10    /* S3  */
st r5, 0[r4]

add r3, r1, r11    /* L4 */
ld r5, 0[r3]
```

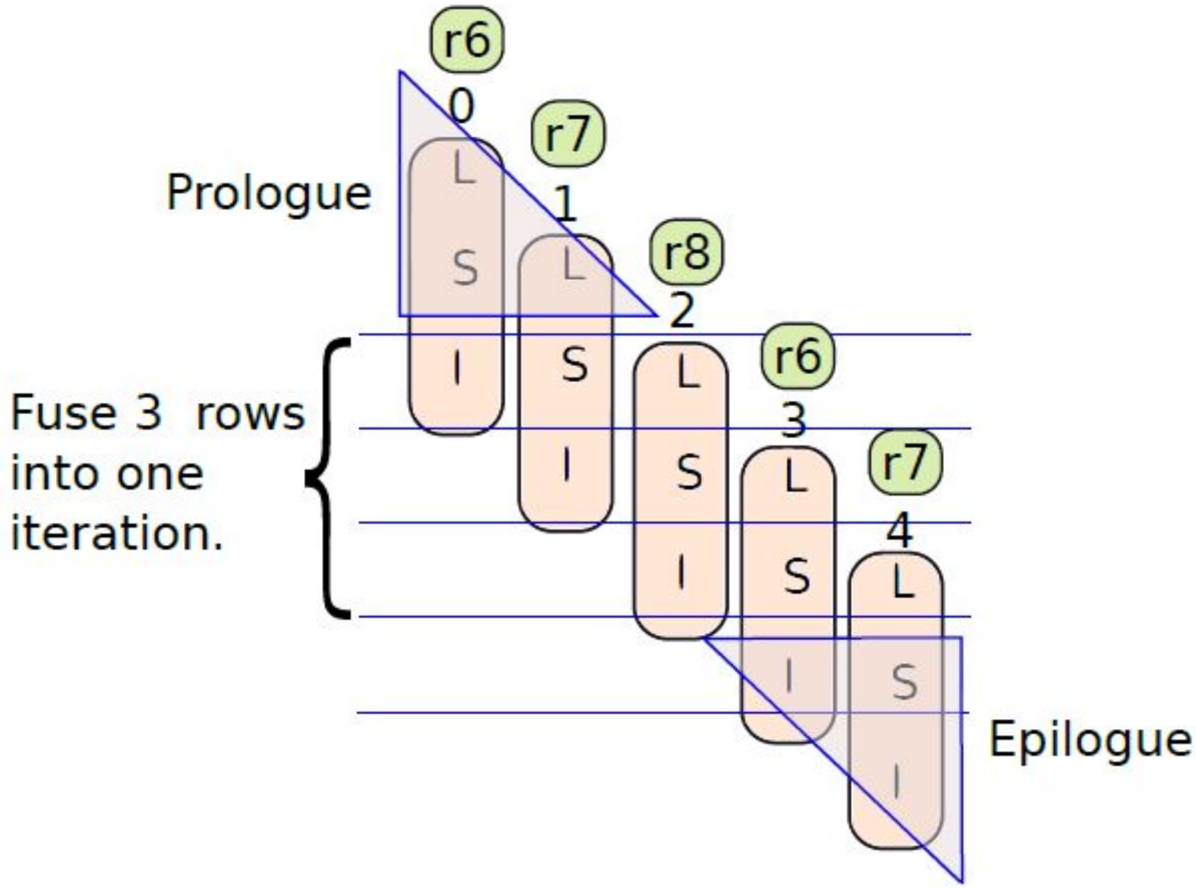
We cannot execute S1 and L2 in parallel because of the dependence on r5

Unroll the loop 3 times

```
/* First Row */
add r6, r6, 3    /* I0 */
lsl r10, r6, 2

add r4, r0, r11 /* S1 */
st  r21, 0[r4]

add r3, r1, r12 /* L2 */
ld  r22, 0[r3]

/* Second Row */
add r7, r7, 3    /* I1 */
lsl r11, r7, 2

add r4, r0, r12 /* S2 */
st  r22, 0[r4]

add r3, r1, r10 /* L3 */
ld  r20, 0[r3]

/* Third Row */
add r8, r8, 3    /* I2  */
lsl r12, r8, 2

add r4, r0, r10 /* S3  */
st  r20, 0[r4]

add r3, r1, r11 /* L4 */
ld  r21, 0[r3]
```

If we had 32 registers, we could do this very easily and elegantly

Epilogue and Prologue



Solution without Unrolling

Original C code

```
int A[300], B[300];
for(i=0; i<300; i++){
    A[i] = B[i];
}
```

Simpler C code

```
int A[300], B[300];
int i = 0;
.loop: if (i<300){
    t = B[i]; /* L */
    A[i] = t; /* S */
    i++;      /* I */
    goto .loop;
}
```

```
i = -1; t = B[0];
.loop if (i < 298) {
    i++;
    A[i] = t;
    t = B[i+1];
}
A[299]  = t;
```

Unrolling and Mixing

SW pipelined version

```
int A[300], B[300];
for(i=0; i<300; i+= 3){
    t1 = B[i];
    t2 = B[i+1];
    t3 = B[i+2];

    t11 = t1 * 5;
    t12 = t2 * 5;
    t13 = t3 * 5;

    A[i]   = t11;
    A[i+1] = t12;
    A[i+2] = t13;
}
```

C code

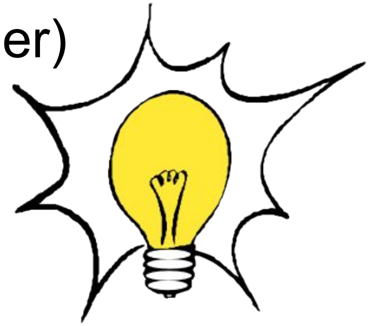
```
int A[300], B[300];
for(i=0; i<300; i++){
    t1 = B[i];
    t2 = t1 * 5;
    A[i] = t2;
}
```

Contents

1. Load Speculation
2. Replay Mechanisms
3. Simpler Version of an OOO Processor
4. Compiler based Techniques
5. EPIC based Techniques: Intel Itanium

Can we outsource the work of renaming and scheduling to the compiler?

- Sounds like a promising idea …
- Less hardware □ less power, less complexity
- Modern software is quite fast and quite intelligent
- Basic idea:
  - Create bundles of several instructions (using the compiler)
  - Schedule a bundle in one go
  - Assume that all dependences are handled.



VLIW Processors

- VLIW (Very Long Instruction Word) processors were the first designs in this space.
  - Bundle instructions into long words
  - If an instruction is 4 bytes, bundle 4 into a 16-byte word
  - Schedule and execute all instructions together
- Problems caused by
  - Conditional *if* statements – control flow not predictable
  - Memory instructions – addresses are computed at runtime

Basic philosophy of many VLIW processors → It is the compiler's job to ensure correctness

If Statements: Predicated Execution

If Statements

Use predicated execution (remember GPUs).

- There maybe a branch in the bundle
- If it is *taken*, the rest of the instructions are invalid
- Mark them with an *invalid* bit
- Let these instructions pass through the pipeline (just don't process them)
- Remember *predicated execution* in GPUs

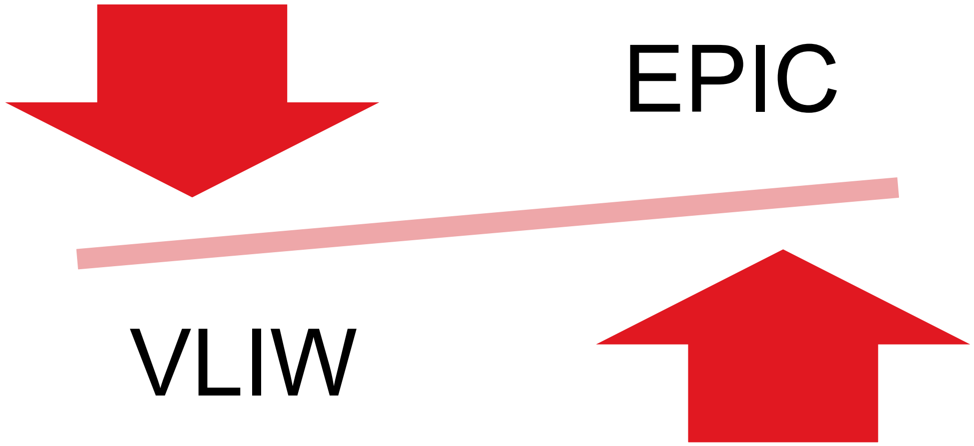
Curious Case of Memory Instructions

```
st r1, 8[r2]
ld r3, 8[r4]
```

- We can have multiple memory instructions in a bundle
- The addresses are computed at runtime
- In this case, we have a hazard
- Same is the case for two store instructions, and a load □ store dependence

Avoid such situations in software or hardware

VLIW vs EPIC

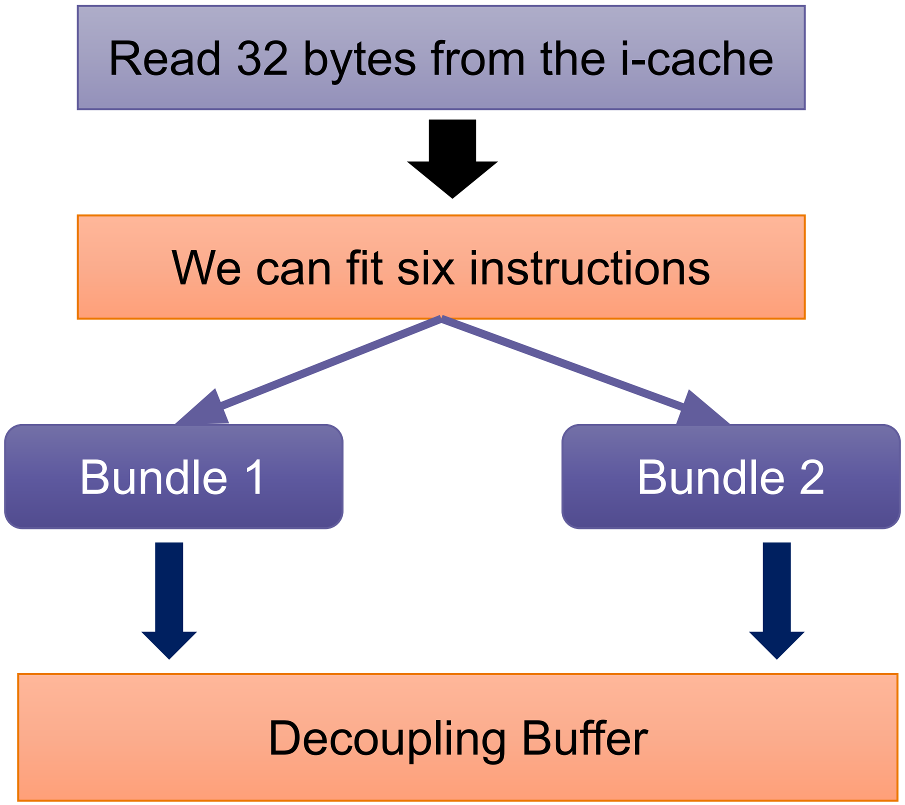


- Given that VLIW processors do not necessarily guarantee correctness, their usability is limited
- Mostly used in digital signal processors
- VLIW processors have been replaced by EPIC processors
- EPIC □ Explicitly Parallel Instruction Computing
- They guarantee correctness
    - Irrespective of the compiler

Intel Itanium Processor

- Unique collaboration between Intel and HP
- Aim:
  - EPIC processor
  - Designed to leverage the best of software and hardware
  - Targeted the server market
  - Primarily gets rid of the scheduler: instruction window, wakeup, select, and broadcast
  - The branch predictor, decode unit, execute units, and advanced load-store handling are still required

Fetch Stage



- Each bundle contains 3 instructions
- The decoupling buffer can hold 8 such bundles

Branch Predictors

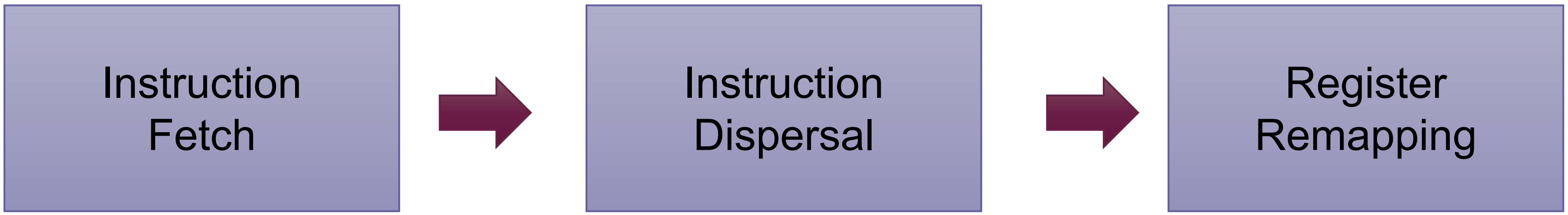
Itanium has four types of branch predictors

- Compiler directed
  - Four special registers: Target Address Registers (TARs)
  - The compiler populates them.
  - Contain a PC and a target
  - Whenever the current PC matches the PC in a TAR predict taken and jump to the target
- Traditional Predictor
  - Large PAp predictor

Branch Predictors – II

- Multi-way Branches
  - Compilers ensure that (typically) the last instruction in a bundle is a branch
  - If there are multiway branches: there are many possible targets for a given bundle
  - Predict the first instruction that is most likely a taken branch and then predict its target
- Loop Exit Predictor
  - The compiler marks the loop instruction
  - It also populates the register with the loop iteration count
  - The predictor keeps decrementing the loop count till it reaches 0. Then it predicts a loop exit.

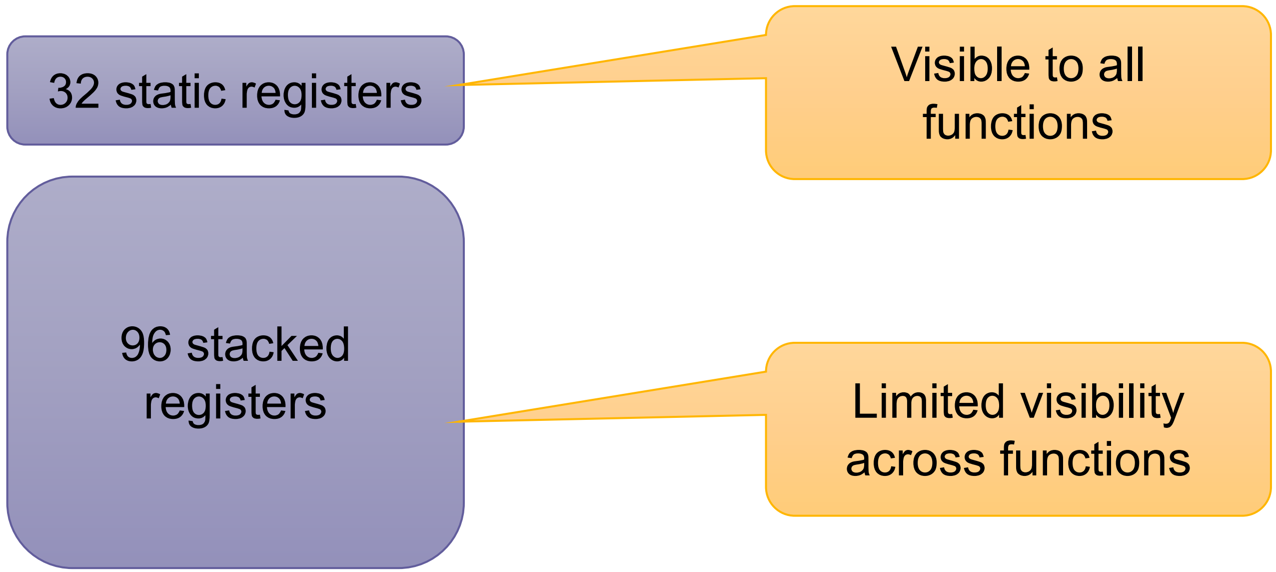
This part of the pipeline



- Itanium has 9 issue ports: 2 for memory, 2 for integer, 2 for floating point, 3 for branch instructions
- Disperse the instructions □ map instructions to issue ports
- Data hazards:
  - Option 1: Avoid data hazards in a bundle or put *nop* instructions or forward results.
  - Option 2: Use *stop bits*. Instructions between two instructions with their *stop bits* set to 1 are independent of each other.
- Structural hazards: Each bundle indicates the resources that it requires. This information is used to avoid structural hazards.

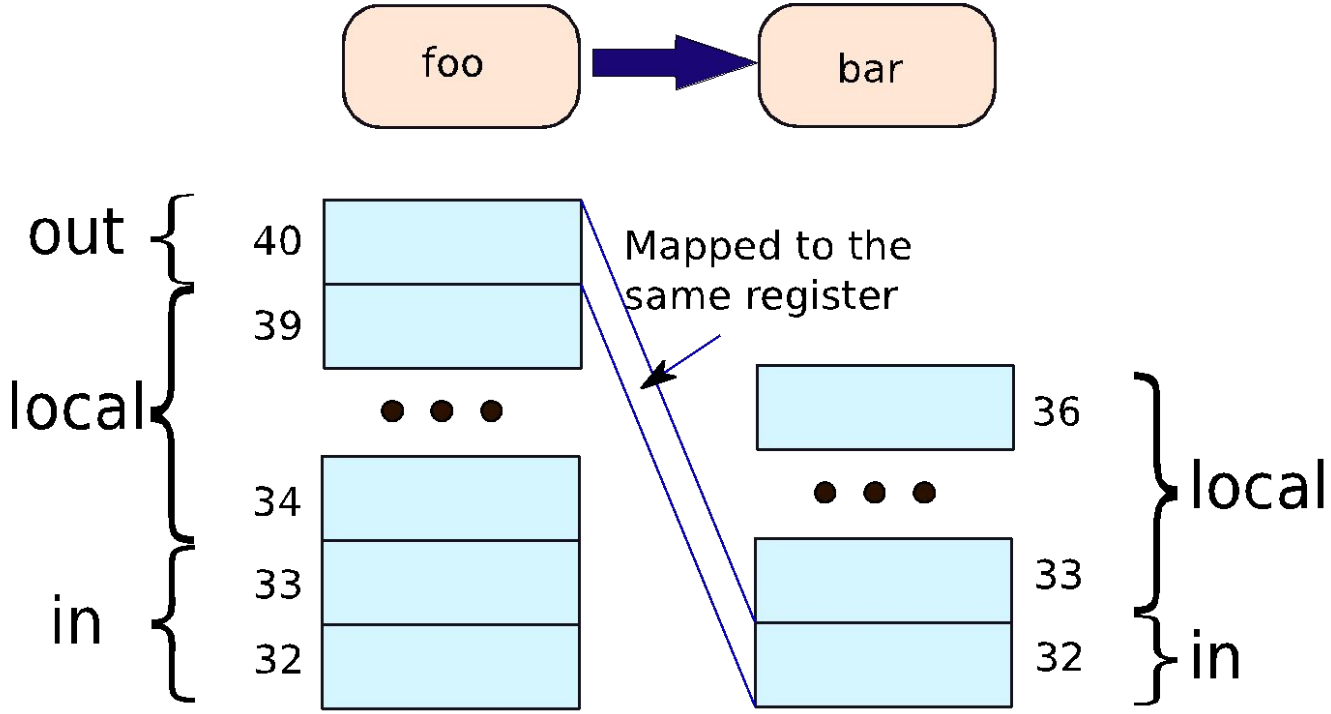
Register Remapping Stage

Large 128-entry register file.



Allocate different sets of virtual registers to each function.
This avoids spilling.

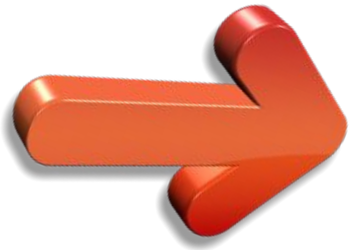
Example: Function *foo* calls function *bar*



We deliberately create an overlap in the virtual register set to pass parameters.

Register Stack Frame

- The *in* and *local* registers are preserved across function calls.
- The *out* registers are used to send parameters to *callee* functions.
- An *alloc* instruction automatically creates such a register stack frame.
- Communicating return values.



Binary Search

```
int bin_search(int arr[], int left, int right, int val){
        /* exit conditions */
        int mid;
        if (right < left) return -1;

        mid = (left + right) / 2;
        if(val == arr[mid])
                return mid;

        /* recursive conditions */
        if(val < arr[mid])
                return bin_search(arr, left, mid - 1, val);
        else
                return bin_search(arr, mid + 1, right, val);
}

int main(){
        ...
        result = bin_search ( ... );
next:
        printf("%d", result);
        ...
}
```

No processing done after receiving the return value. Just pass it on.

This is known as *tail recursion*

Register Stack Frame

- The *in* and *local* registers are preserved across function calls.
- The *out* registers are used to send parameters to *callee* functions.
- An *alloc* instruction automatically creates such a register stack frame.
- Communicating return values.
  - Store the return values in a *static* register
  - In this case, directly jump to the return address in the *main* function.
  - We don't need to process return values.

Support for Software Pipelining and Overflows

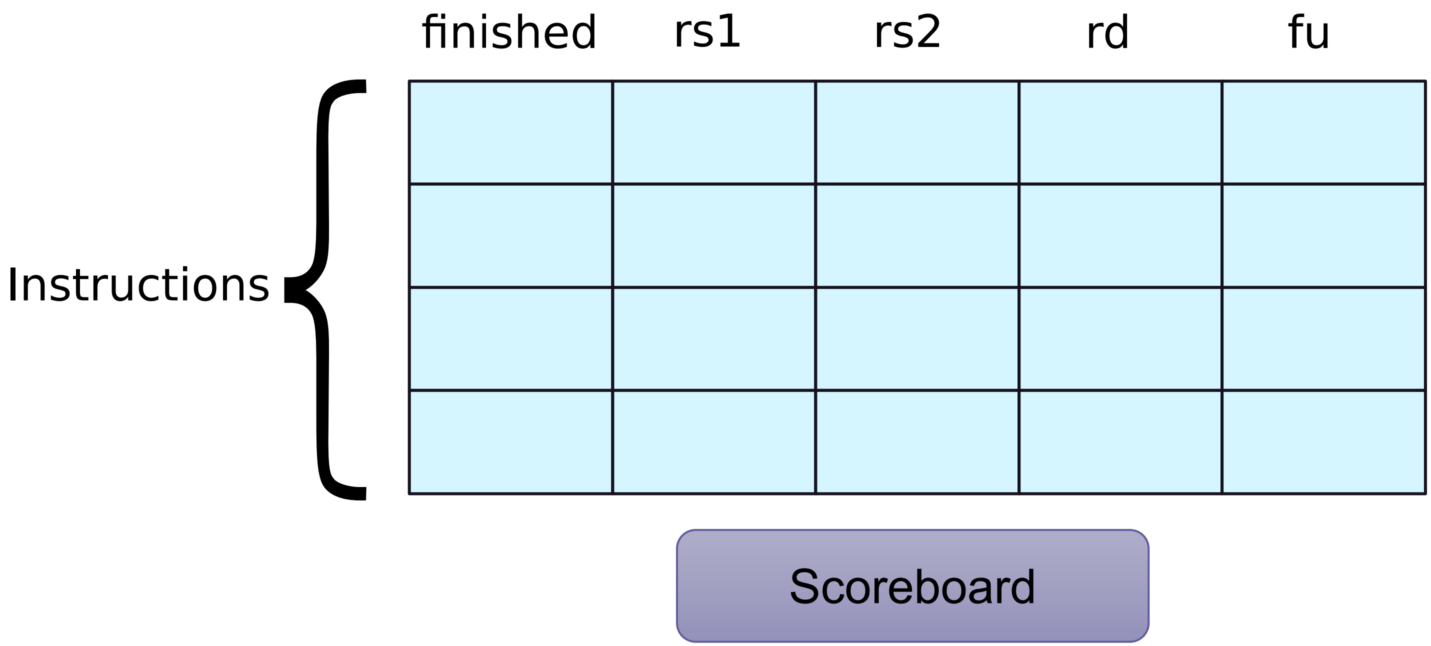
Main Problem: We run out of registers

- Itanium has a Register Stack Engine (RSE)
  - Automatically handles the spilling of registers to memory and restoring them

Software Pipelining

- We use separate registers for the same variable across different iterations.
- This issue is taken care of automatically
- Notion of a rotating register set
  - Assign registers based on the loop iteration number
  - Easier to write the code of SW-pipelined loops

High Performance Execution Engine



- Simple mechanism for OOO execution
- Makes instructions wait till it is safe to execute them
- *finished* field □ 1 if it has finished its execution, 0 otherwise.
- *fu* □ Name of the functional unit

Conditions: Instruction *I*

WAW Hazards

1. Check all the earlier entries
2. For each earlier entry *E* the following expression should be *false*

$$(E.finished = 0) \wedge (E.rd = I.rd)$$

WAR Hazards

1. Check all the earlier entries
2. For each earlier entry *E* the following expression should be *false*

$$(E.finished = 0) \wedge ((E.rs1 = I.rd) \vee (E.rs2 = I.rd))$$

Conditions: II

RAW Hazards

1. Check all the earlier entries
2. For each earlier entry *E* the following expression should be *false*

$$(E.finished = 0) \wedge ((E.rd = I.rs1) \vee (E.rd = I.rs2))$$

Structural Hazards

1. For each earlier entry *E*

$$(E.finished = 0) \wedge (E.fu = I.fu)$$

- Instructions wait in the scoreboard until they are safe
  - No hazards

Predication

Consider the following piece of code

```
if( rand () %2 == 0)
    x = y;
else
    x = z;
```

- If we flush the pipeline upon a branch misprediction
  - It would be quite unfair
- Let the *if* statement just be used to mark an instruction with the result of the comparison
- Store the result in a flags register
- The rest of the instructions are processed regardless of the branch outcome
- Some results modify the architectural state, many do not

Code without Predication

```
/* mappings : x <-> r1 , y <-> r2 , z <-> r3 */
mod r0 , r0 , 2 /* assume r0 contains the output of rand () ,
                   compute the remainder when dividing it by 2
*/

 cmp r0 , 0 /* compare */
 beq . even
 mov r1 , r3 /* odd case */
 b. exit

 .even :
 mov r1 , r2 /* even case */

 . exit :
```

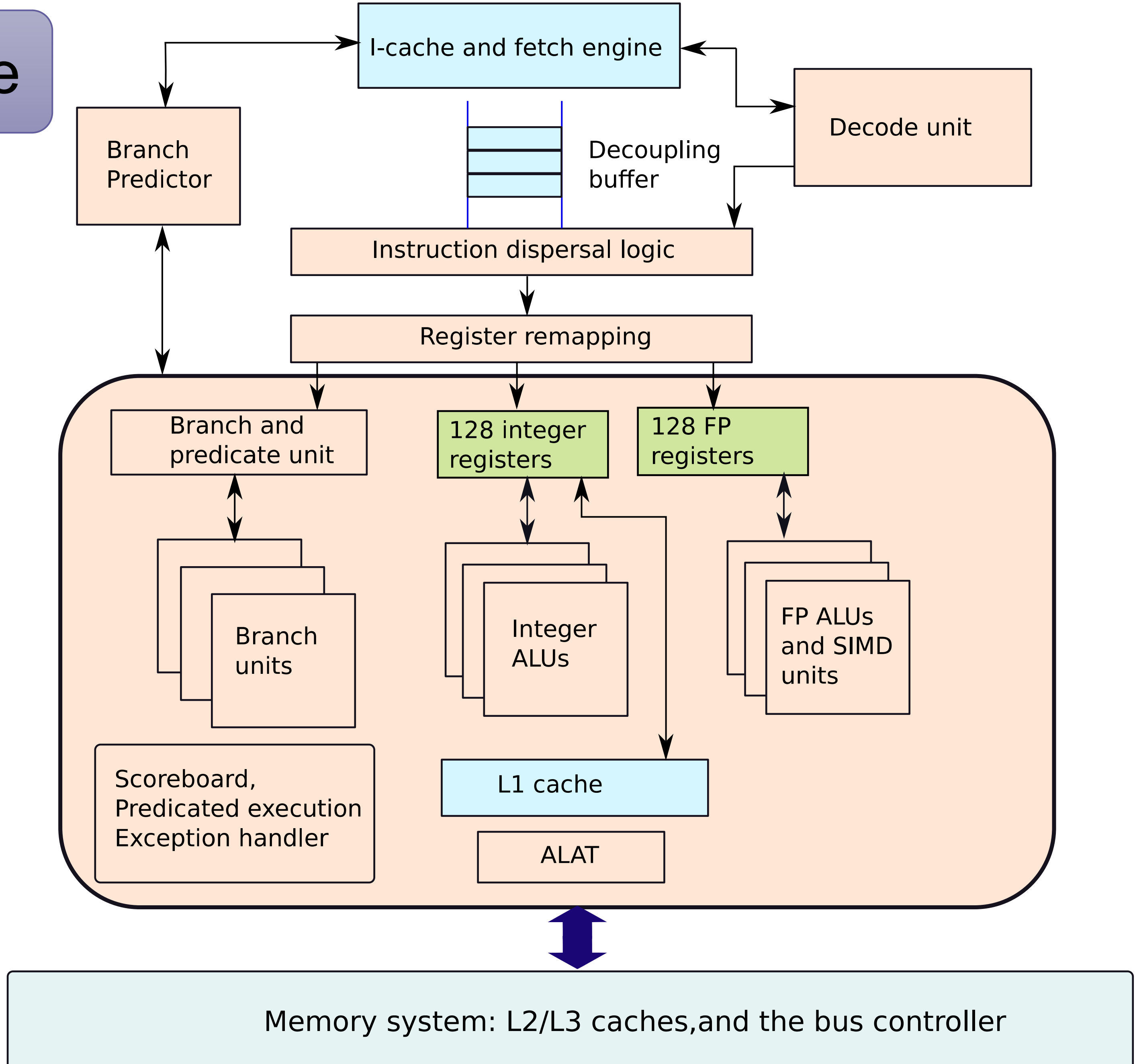
Count the number of branch instructions.

Predicated Instructions

```
/* mappings : x <-> r1 , y <-> r2 , z <-> r3 */

mod r0 , r0 , 2 /* assume r0 contains the output of rand ()
                   compute the remainder when dividing it by */

po ,pe = cmp r0 , 0 /* compare and set the predicates */
[po] mov r1 , r3 /* odd case */
[pe] mov r1 , r2 /* even case */
```

- The comparison generates predicates (flags)
  - *po* □ number is odd, *pe* □ number is even
- If the predicate is correct, the instruction gets executed, otherwise not
- Itanium sets and maintains the predicate registers
- An instruction is executed if all the predicate registers are set to 1 for the instruction.

Pipeline


I-cache and fetch engine
Branch Predictor
Decoupling buffer
Decode unit
Instruction dispersal logic
Register remapping
Branch and predicate unit
128 integer registers
128 FP registers
Branch units
Integer ALUs
FP ALUs and SIMD units
Scoreboard, Predicated execution Exception handler
L1 cache
ALAT
Memory system: L2/L3 caches,and the bus controller

Load Boosting

- *Boost* a load and some instructions that use its value to a point before "where it appears in the code".
- Loads are almost always on the critical path hence, boosting them is beneficial because they can get their data *early*
- Put the load address in the ALAT
  - Advanced Load Address Table
- Subsequently, each store checks the ALAT for a match, and marks it (upon an address match)
- Put a load-check (ld.c) instruction at the original point
  - Check the ALAT
  - If there have been no intervening stores to the same address, the speculation is successful.
  - Else, re-execute the load and its boosted forward slice

Conclusion

There are four kinds of aggressive speculation: load address, dependence, latency, and value speculation

There are three methods of replaying instructions: non-selective, delayed selective and token-based replay

We can use the ROB as the physical register file and use it to buffer temporary values. The ARF can contain the committed state

A host of compiler optimizations can be used to speed up programs and improve their memory access behavior.

EPIC processors guarantee correctness as well as follow the VLIW model that gives primacy to the compiler.

The End