

CS356 Unit 12b

Advanced Processor Organization

Goals

- Understand the terms and ideas used in a modern, high-performance processor
- Various systems have different kinds of processors and you should understand the pros and cons of each kind of processor
- Terms to listen for and understand the concept:
 - Superscalar/multiple issue, loop unrolling, register renaming, out-of-order execution, speculation, and branch prediction

A New Instruction

- In x86, we often perform
 - `cmp %rax, %rbx`
 - `je L1` or `jne L1`
- Many instruction sets have a **single instruction** that both compares and jumps (limited to registers only)
 - `je %rax, %rbx, L1`
 - `jne %rax, %rbx, L1`
- Let us assume x86 supports such an instruction in our subsequent discussion

INSTRUCTION LEVEL PARALLELISM

Have We Hit The Limit

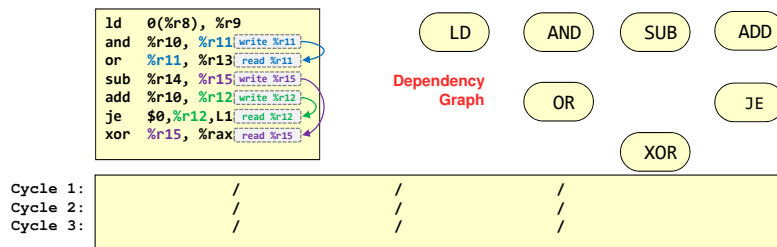
- Under ideal circumstances, pipeline would allow us to achieve a throughput (IPC = Instruction per clock) of _____
- Can we do better? Can we execute more than one instruction per clock?
 - Not with a single pipeline
 - But what if we had _____
 - What if we fetched multiple _____ per clock and let them run down the pipeline in parallel
- Let's exploit _____!

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
 - _____ Level Parallelism (ILP): Overlapping instructions within a single process/thread of execution
 - _____ Level Parallelism (TLP): Overlap execution of multiple processes / threads
 - _____ 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; }
- We'll focus on ILP in this unit

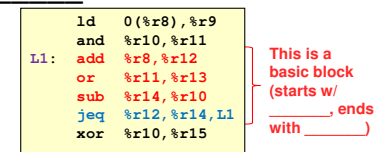
Instruction Level Parallelism (ILP)

- 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 _____) is what truly _____ ordering
 - We call these dependencies _____ Hazards
- Independent instructions can be _____
- Control hazards also provide ordering constraints



Basic Blocks

- Basic Block (def.) = Sequence of instructions that will _____ be executed _____
 - No conditional branches out
 - No branch targets coming in
 - Also called "straight-line" code
 - Average size: _____ instrucs.
- Instructions in a basic block can be overlapped if there are no data dependencies
- _____ dependences really limit our window of possible instructions to overlap
 - Without extra hardware, we can only overlap execution of instructions within a basic block



Superscalar

- When airplanes broke the sound barrier we said they were super-sonic
- When processor (HW) can complete _____ instruction per clock cycle we say they are super-scalar
- **Problem:** The HW can execute 2 or more instructions during the same cycle but the SW may be written and compiled assuming 1 instruction executing at a time.
- **Solutions**
 - _____ the code and rely on the _____ to safely order instructions that can be run in parallel (static scheduling)
 - Build the _____ to be smart, _____ instructions on the fly while guaranteeing correctness (dynamic scheduling)



This Photo by Unknown Author is licensed under CC BY-NC-ND

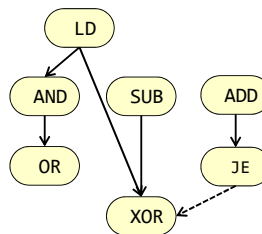
Superscalar (Multiple Issue)

- Multiple "pipelines" that can fetch, decode, and potentially execute more than 1 instruction per clock
 - k-way superscalar = Ability to complete up to k instructions per clock cycle
- Benefits
 - Theoretical throughput greater than 1 (IPC > 1)
- Problems
 - Hazards
 - Dependencies between instructions limiting parallelism
 - Branch/jump requires flushing all pipelines
 - Finding enough parallel instructions

Data Flow and Dependency Graphs

- The compiler produces a sequential order of instructions
- Modern processors will transform the sequential order to execute instructions in parallel
- Instructions can be executed in any valid _____ of the dependency graph

```
ld 0(%r8), %r9
and %r9, %r11
or %r11, %r13
sub %r14, %r15
add %r10, %r12
je $0,%r12,L1
xor %r15, %r9
```



Compiler-based solutions

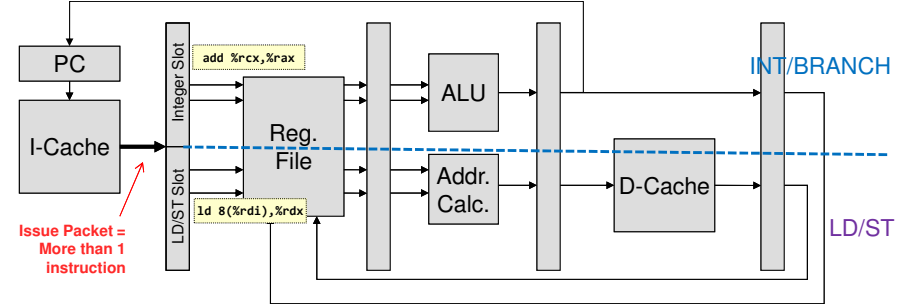
STATIC MULTIPLE ISSUE MACHINES

Static Multiple Issue

- _____ is responsible for finding and packaging instructions that can execute in parallel into issue packets
 - 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 (a.k.a. `_ery_ong_nstruction_ord`)
 - Intel's Itanium used this technique (static multiple issue) but called it EPIC (`_xplicitly_arallel_nstruction_omputer`)

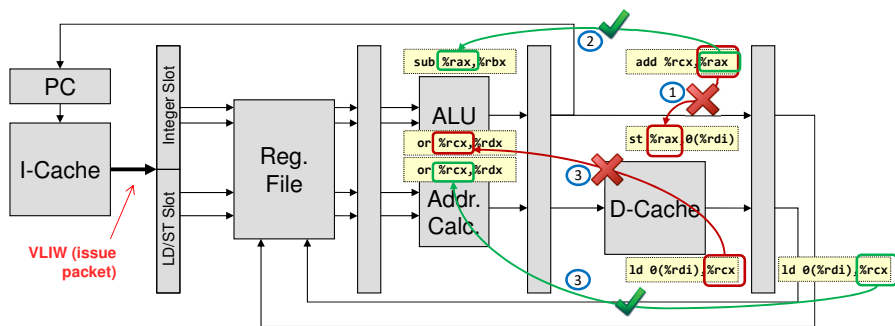
Example 2-way VLIW machine

- One issue slot for INT/BRANCH operations & another for LD/ST instructions
- I-Cache reads out an entire issue packet (more than 1 instruction)
- HW is added to allow many registers to be accessed at one time
 - Just more multiplexers
- Address Calculation Unit (just a simple adder)



2-way VLIW Scheduling

- No forwarding w/in an issue packet (between instructions in a packet)
- Full forwarding to previous instructions
 - Those behind in the pipeline
- Still 1 stall cycle necessary when LD is followed by a dependent instruction



Sample Scheduling

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

```
void f1(int* A, int n)
{
    for( ; n != 0; n--, A++)
        *A += 5;
}
```

```
# %rdi = A
# %esi = n = # of iterations
L1: ld 0(%rdi),%r9
    add $5,%r9
    st %r9,0(%rdi)
    add $4,%rdi
    add $-1,%esi
    jne $0,%esi,L1
```

time	Int./Branch Slot	LD/ST Slot

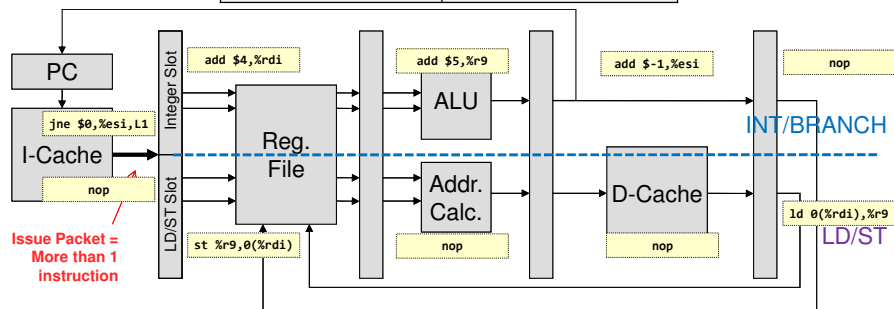
w/o modifying original code but with code movement
IPC = ___ instrucs. / ___ cycles = ___

	Int./Branch Slot	LD/ST Slot

w/ modifications and code movement
IPC = ___ instrucs. / ___ cycle = ___

Annotated Example

Int./Branch Slot	LD/ST Slot
	ld 0(%rdi),%r9
add \$-1,%esi	
add \$5,%r9	
add \$4,%rdi	st %r9,0(%rdi)
jne \$0,%esi,L1	



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**
 - Copy the body of the loop k times and iterate only n/k times
 - Instructions from different body iterations can be run in parallel

```
void f1(int* A, int n)
{
  for( ; n != 0; n--, A++)
    *A += 5;
}
```

```
// Loop unrolled 4 times
void f1(int* A, int n)
{ // assume n is a multiple of 4
  for( ; n != 0; n-=4, A+=4){
    *A += 5;
  }
}
```

Loop Unrolling

```
void f1(int* A, int n) {
  for( ; n != 0; n--, A++)
    *A += 5;
}
```

```
// Loop unrolled 4 times
for(i=0; i < MAX; 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;
}
```

```
# %rdi = A
# %esi = n = # of iterations
L1: ld 0(%rdi),%r9
   add $5,%r9
   st %r9,0(%rdi)
   add $4,%rdi
   add $-1,%esi
   jne $0,%esi,L1
```

Unrolled Code

```
# %rdi = A
# %esi = n = # of iterations
L1: ld 0(%rdi),%r9
   add $5,%r9
   st %r9,0(%rdi)
   ld 4(%rdi),%r9
   add $5,%r9
   st %r9,4(%rdi)
   ld 8(%rdi),%r9
   add $5,%r9
   st %r9,8(%rdi)
   ld 12(%rdi),%r9
   add $5,%r9
   st %r9,12(%rdi)
   add $16,%rdi
   add $-4,%esi
   jne $0,%esi,L1
```

Original Code

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

Code Movement & Data Hazards

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

```
L1: add %r8, %r9
   add %r9, %r10
   ld 0(%r11),%r9
Original
```

LD instruction is REALLY independent (only needs %r11).
Could LW instruction be moved between the 2 add's?

```
L1: add %r8, %r9
   add %r9, %r10
   ld 0(%r11), %r9
   sub %r9, %r12
Original
```

Could LD instruction be run in parallel with or before first add?

```
L1: ld 0(%r11), %r9
   add %r8, %r9
   add %r9, %r10
   sub %r9, %r12
Proposed
```

Register Renaming

- Unrolling is not enough because even though each iteration is independent there are conflicts in the use of registers (%r9 in this case)
 - Can't move another 'ld' instruction up until 'st' is complete due to a WAR hazard even though there is not a true data dependence
- Since there is no true dependence (ld does not need data from 'st' or 'add' above) we can solve the problem by register renaming
- Register Renaming:** Using different registers to solve WAR / WAW hazards

```
# %rdi = A
# %esi = n = # of
iterations
L1: ld  0(%rdi),%r9
     add $5,%r9
     st  %r9,0(%rdi)
     ld  4(%rdi),%r9
     add $5,%r9
     st  %r9,4(%rdi)
     ld  8(%rdi),%r9
     add $5,%r9
     st  %r9,8(%rdi)
     ld  12(%rdi),%r9
     add $5,%r9
     st  %r9,12(%rdi)
     add $16,%rdi
     add $-4,%esi
     jne $0,%esi,L1
```



Scheduling w/ Unrolling & Renaming

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

```
# %rdi = A
# %esi = n = # of
iterations
L1: ld  0(%rdi),%r9
     add $5,%r9
     st  %r9,0(%rdi)
     ld  4(%rdi),%r10
     add $5,%r10
     st  %r10,4(%rdi)
     ld  8(%rdi),%r11
     add $5,%r11
     st  %r11,8(%rdi)
     ld  12(%rdi),%r12
     add $5,%r12
     st  %r12,12(%rdi)
     add $16,%rdi
     add $-4,%esi
     jne $0,%esi,L1
```

Int./Branch Slot	LD/ST Slot

w/ Loop Unrolling and Register Renaming
(Notice how the compiler would have to modify the code to effectively reschedule)

IPC = _____

Data Dependency Hazards Summary

- RAW = 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

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 known as loop carried dependence

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

Memory Hazard Issue

- Suppose %rsi and %rdi are passed in as arguments to a function, can we reorder the instructions below?
 - ____, if _____ we have a data _____ in _____
- Data dependencies can occur via _____ and are harder for the compiler to find at compile time forcing it to be conservative

```
ld 0(%rsi),%edx
addl $5,%edx
st %edx,0(%rsi)

ld 0(%rdi),%eax
addl $1,%eax
st %eax,0(%rdi)
```

Can we reorder these instructions?

Int./Branch Slot	LD/ST Slot
	ld 0(%rsi),%edx
	ld 0(%rdi),%eax
addl \$5,%edx	
addl \$1,%eax	st %edx,0(%rsi)
	st %eax,0(%rdi)

Can we move the 2nd 'ld' up to enhance performance? _____

Memory Disambiguation

- Data dependencies occur in MEMORY and not just registers
- Memory RAW dependencies are also made harder because of different ways of addressing the same memory location
 - Can the following be reordered?
 - st %eax, 4(%rdi)
 - ld -12(%rsi), %ecx
 - _____! What if %rsi = _____
- Memory disambiguation refers to the process of determining if a sequence of stores and loads reference the _____ address (ordering often needs to be maintained)
- We can only reorder LD and ST instructions if we can disambiguate their addresses to determine any RAW, WAR, WAW hazards
 - LD -> LD is always fine (RAR)
 - ST -> LD (RAW), LD -> ST (WAR) or ST -> ST (WAW) hazards that need to be disambiguated

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 GPR's
 - (128) FPR's
- On-chip L3 cache (12 MB or cache memory)

Static Multiple Issue Summary

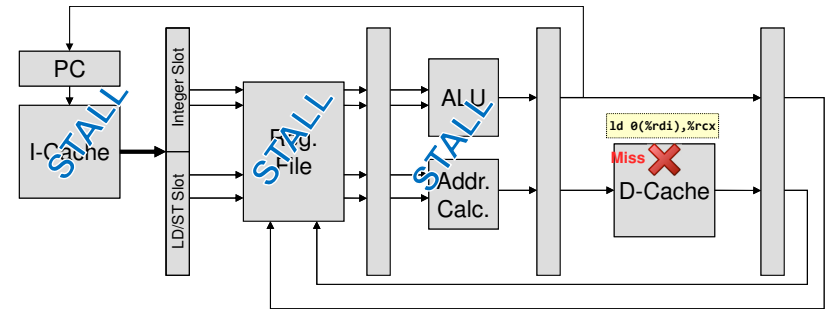
- Compiler is in charge of reordering, renaming, unrolling original program code to achieve better performance
- Processor is designed to fetch/decode/execute multiple instructions per cycle in the order determined by the compiler
- Pros: HW can be _____ and thus _____
 - More cores
 - Potentially higher clock rates
- Cons: Requires _____
 - No support for legacy software

HW-based solutions

DYNAMIC MULTIPLE ISSUE MACHINES

Overcoming the Memory Latency

- What happens to instruction execution if we have a cache miss?
 - All instructions behind us need to _____!
 - Could take potentially _____ of clock cycles to fetch the data
- Can we over come this?

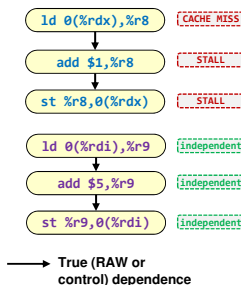


Out-Of-Order Execution

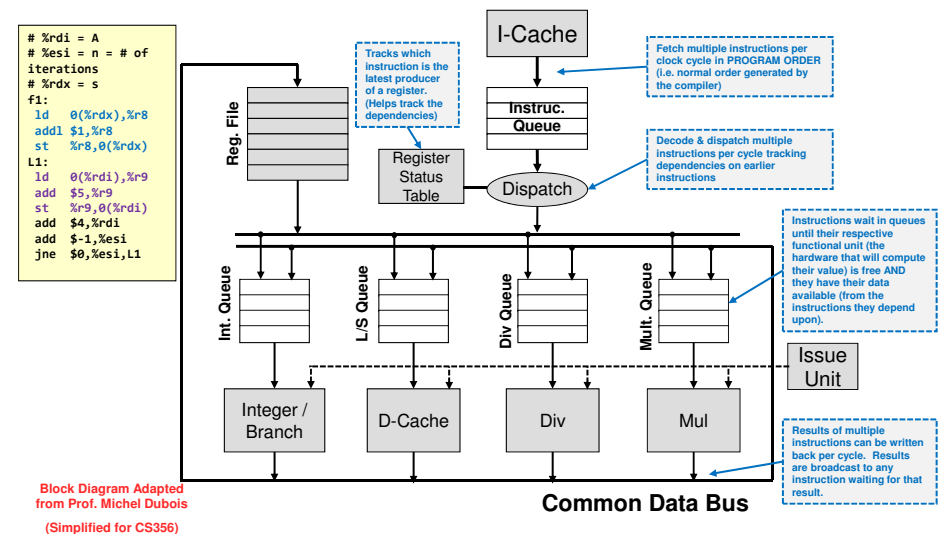
- **Idea:** Have processor find dependencies as instructions are fetched/decoded and execute **independent** instructions that come after **stalled** instructions
 - Known as **Out-of-Order Execution** or **Scheduling**
 - HW will determine the "dependency" graph at runtime and as long as an instruction isn't waiting for an earlier instruction, let it execute!

```
void f1(int* A, int n, int* s)
{ *s += 1;
  for( ; n != 0; n--, A++)
    *A += 5;
}
```

```
# %rdi = A
# %esi = n = # of iterations
# %rdx = s
f1:
  ld 0(%rdx),%r8
  addl $1,%r8
  st %r8,0(%rdx)
  ld 0(%rdi),%r9
  add $5,%r9
  st %r9,0(%rdi)
  add $4,%rdi
  add $-1,%esi
  jne $0,%esi,L1
```

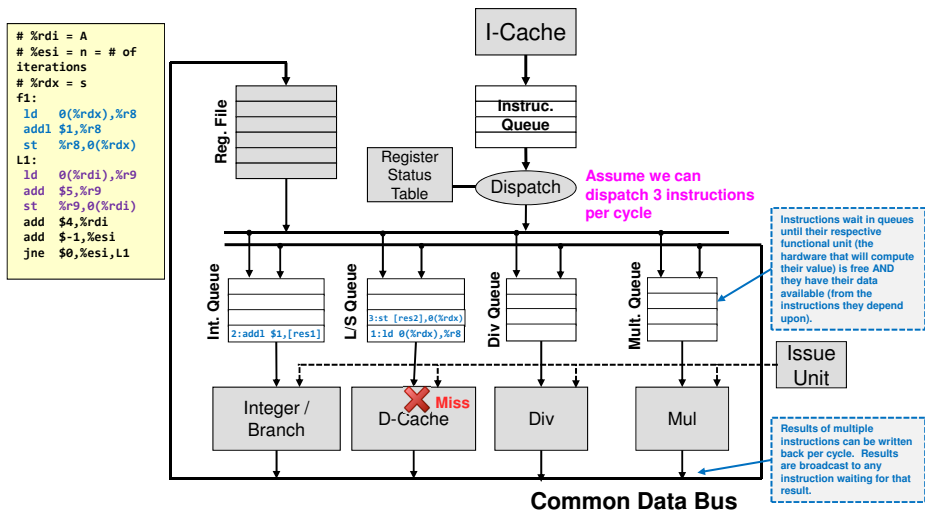


Organization for OoO Execution

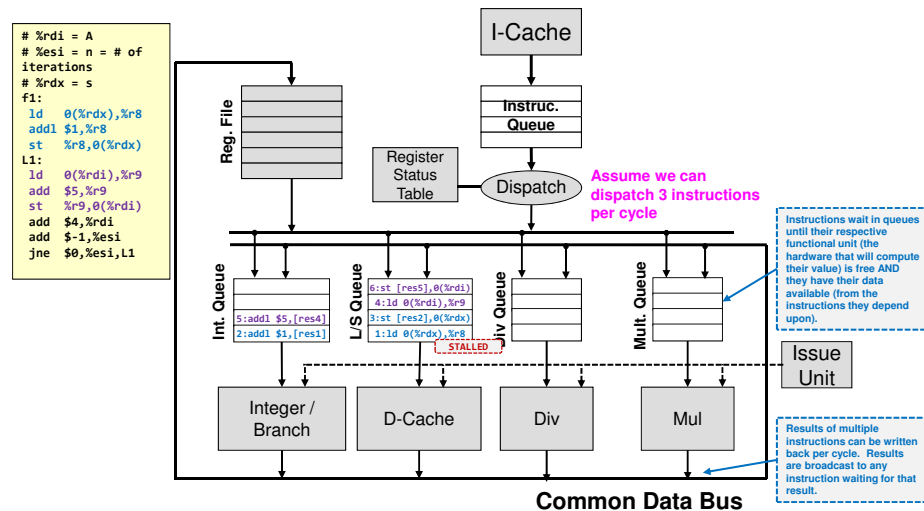


Block Diagram Adapted from Prof. Michel Dubois (Simplified for CS356)

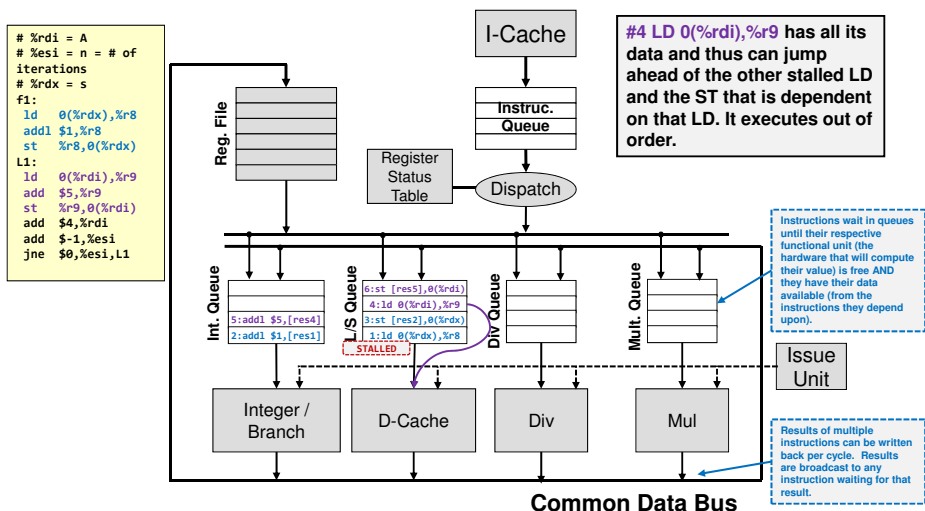
Organization for OoO Execution



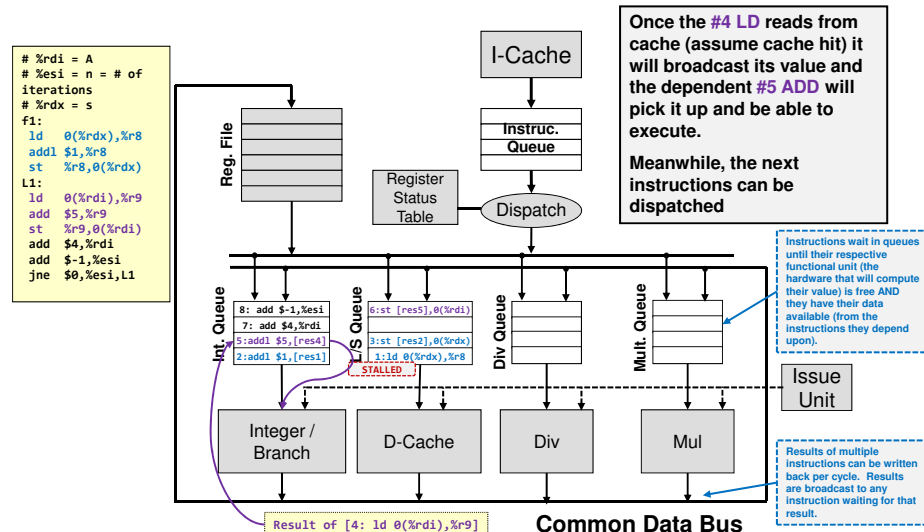
Organization for OoO Execution



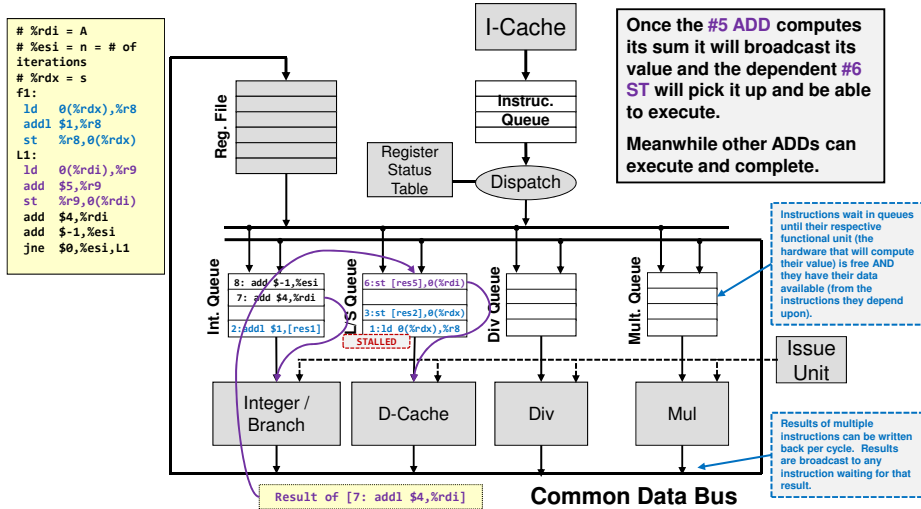
Organization for OoO Execution



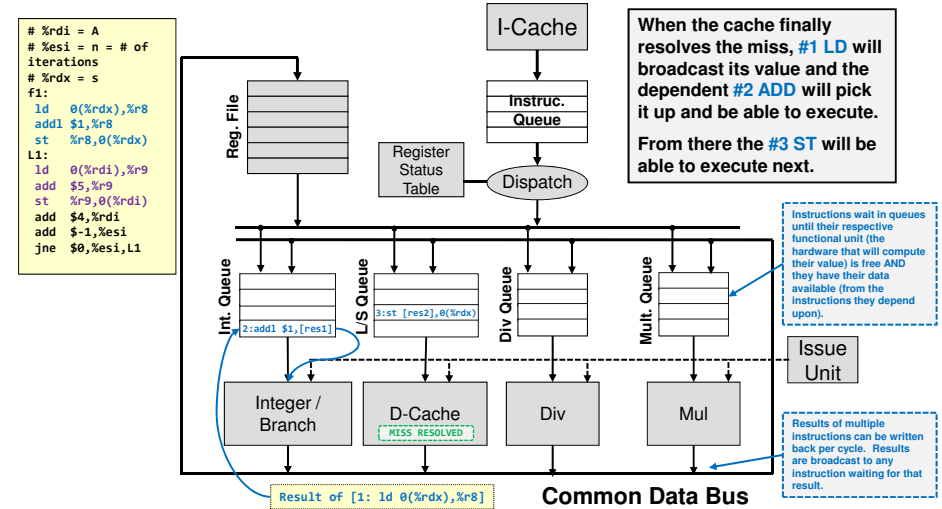
Organization for OoO Execution



Organization for OoO Execution



Organization for OoO Execution



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)

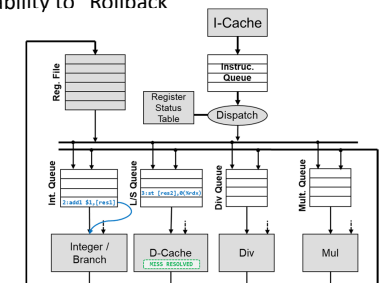
Problems with OoO Execution

- What if an _____ (e.g. page fault) occurs in an earlier instruction AFTER later instructions have already completed
 - OS will save the state of the program and handle the page miss
 - When OS resumes it will restart the process at the ST instruction
 - The subsequent instructions will execute for a 2nd time. BAD!!!
- I need to fetch and dispatch multiple instructions per cycle but when I hit a jump/branch I don't know which way to fetch
- Solution:** _____ Execution with ability to "Rollback"

```
# %rdi = A
# %esi = n = # of iterations
# %rdx = s
f1:
ld 0(%rdx),%r8
addl $1,%r8
st %r8,0(%rdx)
L1:
ld 0(%rdi),%r9
add $5,%r9
st %r9,0(%rdi)
add $4,%rdi
add $-1,%esi
jne $0,%esi,L1
next instruc.
```

Annotations:

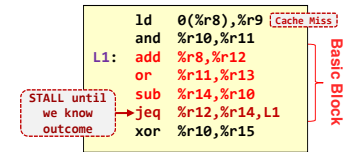
- Resume after
- STALLING
- Page Fault!
- Completed
- Where next



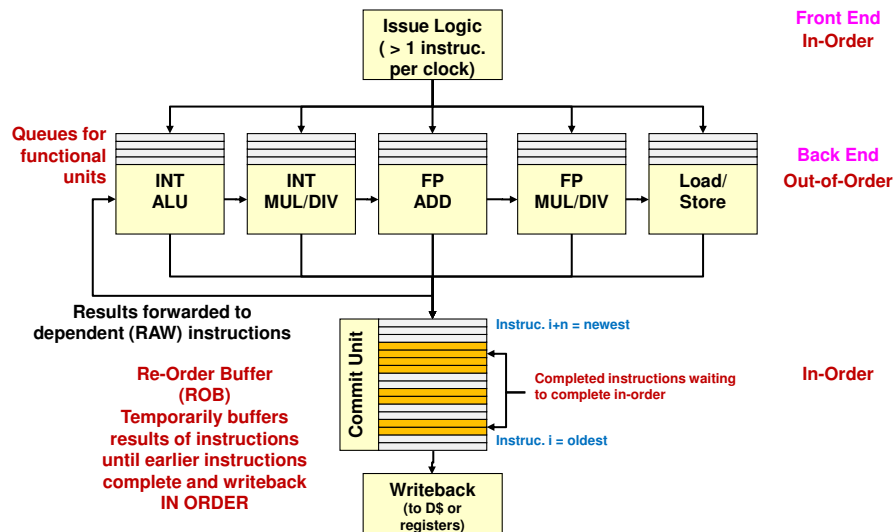
SPECULATIVE EXECUTION

Speculation w/ Dynamic Scheduling

- Basic block size of 5-7 instructions between branches limits ability to issue/execute instructions
 - For safety, we might consider stalling (stop dispatching instructions) until we know the outcome of a jump/branch
- But speculation allows us to predict a branch outcome and continue *issuing* and *executing* down that path
- Of course, we could be wrong so we need the ability to roll-back the instructions we should not have executed if we mispredict
- We add a structure known as the commit unit (or _____)

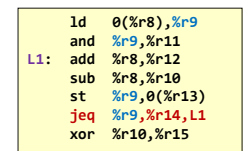


Out-Of-Order Diagram

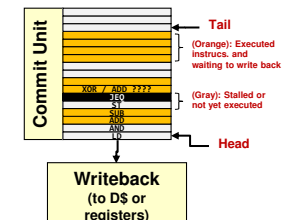


Commit Unit (ROB)

- 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 is also stored in the ROB (rather than writing back to reg. file) until it reaches the head of the ROB
- Commit unit only commits the instruction(s) at the _____ of the queue and only when it is fully _____
 - Ensures that everything committed was correct
 - If we hit an exception or misspeculate/mispredict a branch then throw away everyone behind it in the ROB and start fresh using the correct outcome



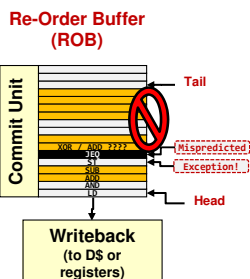
Re-Order Buffer (ROB)



Commit Unit (ROB)

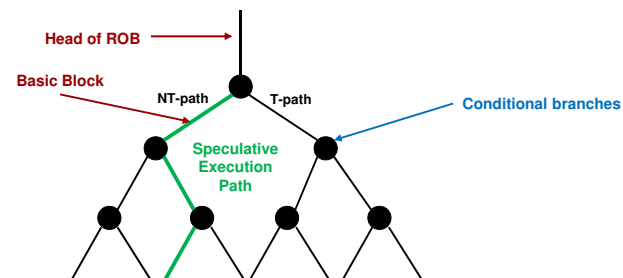
- What happens if the ST instruction that is STALLED ends up causing a page fault...
 - ROB allows us to throw away instructions after it and replay them after the page fault is handled
- When we get to the JEQ, we don't know %r9 so we'll just guess (predict) the outcome and fetch down that predicted path
 - If we mispredict, ROB allows us to throw away instruction results

```
ld 0(%r8),%r9
and %r9,%r11
L1: add %r8,%r12
sub %r8,%r10
st %r9,0(%r13)
jeq %r9,%r14,L1
xor %r10,%r15
```



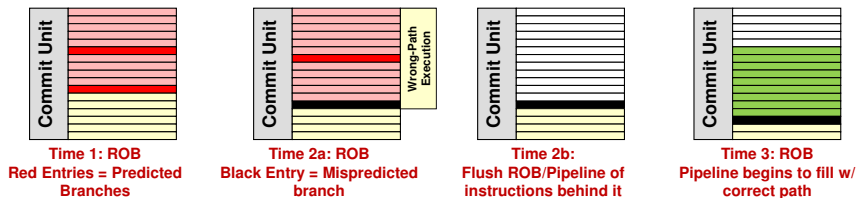
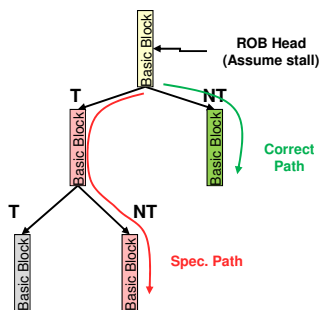
Branch Prediction + Speculation

- To keep the backend fed with enough instructions we need to predict a branch's outcome and perform "speculative" execution beyond the predicted (unresolved) branch
 - Roll back mechanism (flush) in case of misprediction



Speculation Example

- Predict branches and execute most likely path
 - Simply flush ROB entries after the mispredicted branch
 - Need good prediction capabilities to make this useful



Not responsible for this material

BONUS MATERIAL

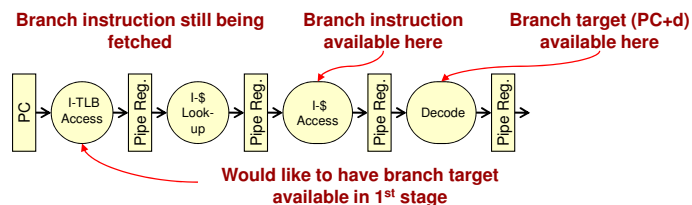
BRANCH PREDICTION

Branch Prediction

- Since basic blocks are often small, multiple 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

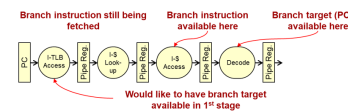
Branch Target Availability

- 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)



Finding the Branch Target

- Key observation: Jump/branches always branch to same place (target is constant for each branch)
 - The first time we fetch a jump we'll have no idea where it is going to jump to and thus have to wait several cycles
 - But let's save the **address where the jump instruction lives** AND **where it wants to jump to** (i.e. the target)
 - Next time the PC gets to the starting address of the jump we can lookup the target quickly
 - Keep all this info in a small "branch target cache/buffer"



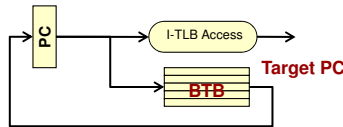
Jump Instruct. Address	Jump Target
0x4004d8	0x4004f7
0x4004f3	0x4004ea
0x4004f5	0x4004fc

```

0000000004004d6 <sum>:
4004d6: 85 f6                test %esi,%esi
4004d8: 7e 1d               jle  4004f7 <sum+0x21>
4004da: 48 89 fa           mov  %rdi,%rdx
4004dd: 8d 46 ff           lea -0x1(%rsi),%eax
4004e0: 48 8d 4c 87 04     lea 0x4(%rdi,%rax,4),%rcx
4004e5: b8 00 00 00 00     mov  $0x0,%eax
4004ea: 03 02             add  (%rdx),%eax
4004ec: 48 83 c2 04       add  $0x4,%rdx
4004f0: 48 39 ca           cmp  %rcx,%rdx
4004f3: 75 f5             jne  4004ea <sum+0x14>
4004f5: eb 05             jmp  4004fc <sum+0x26>
4004fc: c6 05 36 0b 20 00 01 movb $0x0,%eax
400503: c3                movb $0x1,0x200b36(%rip)
retq
    
```

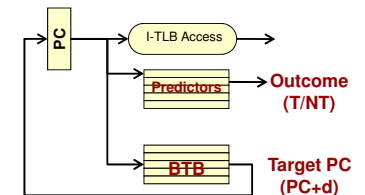
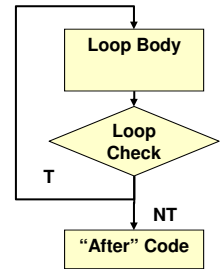
Branch Target Buffer

- 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



Branch Outcome Prediction

- Now that we have predicted the target, we now need to predict the outcome
- Static prediction
 - Have compiler make a fixed guess and put that as a "hint" in the instruction itself
 - Effective for loops
- Dynamic prediction
 - Some jumps are data dependent (e.g. $if(x < y)$)
 - Keep some "history"/records of each branches outcomes from the past & use that to predict the future
 - Store that history in a cache
 - 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 previous branches)

```

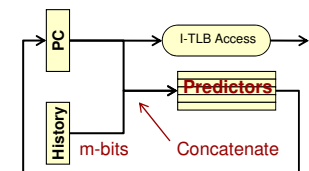
do {
  i--;
  if(x == 2) { ... }
  if(y == 2) { ... }
  if(x == y) { ... }
  // Better: Local or Global
}
while (i > 0);
// Better: Local or Global
    
```

Global (Correlating) Predictor

- Use the outcomes of the last m branches that were executed to select a prediction
 - Given last m jumps, 2^m possible combinations of outcomes & thus predictions
 - When $jeq1=NT$ and $jeq2=NT$, predict $jne = T$, when $jeq1=NT$ and $jeq2=T$, predict $jne = NT$, etc.
 - Branch predictor indexed by concatenating LSB's of PC and m -bits of last m branch outcomes

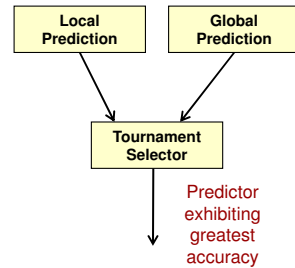
```

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



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



Dynamic Scheduling Summary

- You can understand a modern architecture
 - [https://en.wikichip.org/wiki/intel/microarchitectures/haswell_\(client\)](https://en.wikichip.org/wiki/intel/microarchitectures/haswell_(client))
- Software implications
 - Code with a lot of branches will perform worse than regular code
 - Many cache misses will limit the performance
- But compared to a statically scheduled 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
 - Used in many high-performance embedded processors like GPUs, etc. where code is more regular with high computation demand
- 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 and more waste (time & power)
 - Better for unpredictable, general purpose control code

PHYSICAL VS ARCHITECTURAL REGISTERS

Virtual Registers?

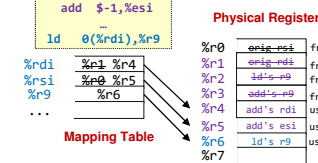
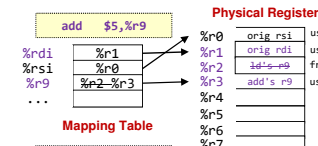
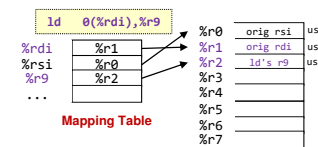
- In static scheduling, the compiler accomplished register renaming by changing the instruction to use other programmer-visible GPR's
- In dynamic scheduling, the HW can "rename" registers on the fly
 - In the code on the left we would want %r9 to be renamed to %r10, %r11, for each iteration
- Solution: A level of indirection
 - Let the register numbers be "virtual" and then perform translation to a "physical" register
 - Every time we write to the same register we are creating a "new version" of the register...so let's just allocate a physically different register

```
# %rdi = A
# %esi = n = # of iterations
# %rdx = s
fi:
ld 0(%rdx),%r8
addl $1,%r8
st %r8,0(%rdx)
L1:
ld 0(%rdi),%r9
add $5,%r9
st %r9,0(%rdi)
add $4,%rdi
add $-1,%esi
jne $0,%esi,L1
L1:
ld 0(%rdi),%r9
add $5,%r9
st %r9,0(%rdi)
add $4,%rdi
add $-1,%esi
jne $0,%esi,L1
```

Trace of instructions over 3 loop iterations. Each iteration is independent if we can rename %r9

Register Renaming

- Whenever an instruction produces a new value for a register, allocate a new physical register and update the table
 - Mark the old physical register as "free"
 - Mark the newly allocated register as "used"
- An instruction that wants to read a register just uses whatever physical register the current mapping table indicates

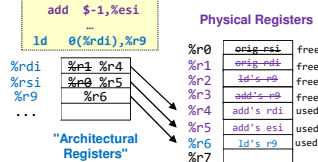
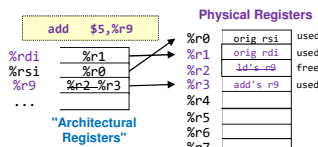
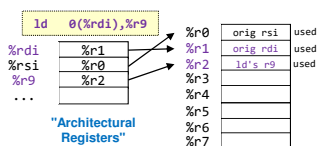


```
# %rdi = A
# %esi = n = # of iterations
# %rdx = s
fi:
ld 0(%rdx),%r8
addl $1,%r8
st %r8,0(%rdx)
L1:
ld 0(%rdi),%r9
add $5,%r9
st %r9,0(%rdi)
add $4,%rdi
add $-1,%esi
jne $0,%esi,L1
L1:
ld 0(%rdi),%r9
add $5,%r9
st %r9,0(%rdi)
add $4,%rdi
add $-1,%esi
jne $0,%esi,L1
```

Trace of instructions over 3 loop iterations. Each iteration is independent if we can rename %r9

Architectural vs. Physical Registers

- Architectural registers** = The (16) x86 registers visible to the programmer or compiler
 - Truly just names ("virtual")
 - The mapping table needs 1 entry per architectural register
- Physical registers** = A greater number of actual registers than architectural registers that is used as a "pool" for renaming
- Often a large pool of physical registers (80-128) to support large number of instructions executing at once or waiting in the commit unit



```
# %rdi = A
# %esi = n = # of iterations
# %rdx = s
fi:
ld 0(%rdx),%r8
addl $1,%r8
st %r8,0(%rdx)
L1:
ld 0(%rdi),%r9
add $5,%r9
st %r9,0(%rdi)
add $4,%rdi
add $-1,%esi
jne $0,%esi,L1
L1:
ld 0(%rdi),%r9
add $5,%r9
st %r9,0(%rdi)
add $4,%rdi
add $-1,%esi
jne $0,%esi,L1
```

Trace of instructions over 3 loop iterations. Each iteration is independent if we can rename %r9