Dynamic Scheduling

Introduction

- Instruction-level parallelism (ILP)
  - Overlaps the execution of multiple instructions
  - Pipelining is the simplest way to extract ILP
  - Addition of bypass hardware exposes more ILP
  - More ILP means low CPI and high IPC
  - CPI increases due to interlock cycles arising from structural, data, control hazards
    - Affect the repeat interval of functional units and execution of dependent instructions
    - Advanced ILP techniques aim at reducing the effect of these interlock cycles

Ready – takes care of RAW hazard
eg if an instr is "ready" ⇒ all dependencies are fulfilled

Introduction

- Instruction-level parallelism (ILP)
  - Two major techniques: dynamic scheduling and static scheduling
    - Scheduling refers to selecting an instruction for execution from a pool of fetched instructions
      - Makes sense only if you can fetch at a faster rate than you can execute; departs significantly from our understanding of linear single-instruction fetch-execute pipeline
    - Observe the similarity with job scheduling in OS
    - Goal is to maximize IPC
    - Intuitive algorithm: execute any of the ready instructions in a cycle
      - An instruction is ready if its operand values are available either through bypass paths or from the register file
      - Is it an optimal schedule? Any correctness concerns?
      - Can we apply any of our learning from OS scheduling?

Introduction

- Instruction-level parallelism (ILP)
  - Dynamic scheduling implements the instruction selection logic in hardware
    - This hardware is enabled every cycle to decide what should execute in a cycle
    - Conceptually easier to do because all dependencies are clearly visible
    - Examples: MIPS R10000 onwards; Alpha 21264, 21364; Intel Pentium Pro, II, III, 4, Core, ⋯; Sun UltraSPARC III, IV; AMD Athlon, Athlon64, Opteron, ⋯; IBM Power3, Power4, Power5, ⋯; IBM/Apple G3, G4, G5, ⋯
  - Static scheduling is a compiler technique where the compiler re-arranges instructions to hide interlock stall cycles
    - Preserving correctness forces it to be conservative
    - Example: Intel Itanium

```
add $1, $2, $3
 :
add $2, ---
```

Not data flow hazard

compiler chose the same name. They are indep instr.

dependences

```
lw  $2, 0($10)
add $3, $2, $2
```

interlock stall cycles (1 or more)

copying

```
SW $12, 0($20)
LW $2, 0($10)
```

compiler can't figure out if these are actually indep.

what if 0+$10 = 0+$20

sw ka RF before addiu $a0 ka WB then WAR hazard saved

Introduction

- Loop iterations are usually good source of ILP

```
int x[100], y[100], i;
for i=0 to 99
   x[i] += y[i];
```

- MIPS translation

```
label: lw $v0, 0($a0)        v0 ← x0
       lw $v1, 400($a0)      v1 ← y0
       addiu $a1, $a1, 1     a1 ← i
       addu $v0, $v0, $v1    v0 ← v0+v1
       sw $v0, 0($a0)        x0 ← v0
       slti $v0, $a1, 100    i < 100 : v0
       bnez $v0, label         v0 < label next
       addiu $a0, $a0, 4     a0 ← a0+4
```

WAW, WAR

  - Minimum number of cycles to complete?

Introduction

- Loop iterations are usually good source of ILP

```
int x[100], y[100], i;
for i=0 to 99
   x[i] += y[i];
```

- MIPS translation

```
label: lw $v0, 0($a0)      IF ID EX MEM WB
       lw $v1, 400($a0)    IF ID EX MEM WB
       addiu $a1, $a1, 1   IF ID EX MEM WB
       addu $v0, $v0, $v1     IF ID EX  MEM WB
       sw $v0, 0($a0)         IF ID EX  MEM WB
       slti $v0, $a1, 100        IF ID EX  MEM WB
       bnez $v0, label           IF ID EX  MEM WB
       addiu $a0, $a0, 4   IF ID EX MEM WB
```

this WB should not happen (after slti X)

  - Minimum number of cycles to complete? for next cycle lw instr

Instr 1 2 3 4 — compiler prepares this → many may be NOPs. processor just issues these in parallels

variable

VLIW – very long Instr word (fixed len)

EPIC – Explicitly parallel Instr computer (variable len, marker bits to show end of instr)

Introduction

- In this case, loop iterations are independent
  - Two different iterations compute on different data
  - We should be able to execute instructions from different iterations in parallel
  - Branches introduce a problem: reduces the stretch of independent instruction (control dependence)
    - Solution#1: Have branch predictors
    - Solution#2: Unroll the loop

      for i = 0 to 99 at step 2: x[i] += y[i]; x[i+1] += y[i+1];
- Static techniques: loop unrolling, VLIW, EPIC, software pipelining, predication
- Dynamic techniques: register renaming, branch prediction, out-of-order issue, multiple issue, advanced techniques

Instruction scheduling

- What limits ILP?
  - Data dependence or true dependence
    - Flow of data: dependence between producer and consumer; may introduce RAW hazard and stalls (dependence is a property of the program, stall is caused by pipeline organization)
    - Flow can happen through register or memory; how to discover memory dependence?
    - Schedule as many independent instructions as possible: static and dynamic techniques
  - Name dependence or false dependence
    - No data flow between involved pair of instructions: antidependence (may cause WAR) and output dependence (may cause WAW)
    - Solution: renaming; how to do memory renaming?


```
sw $2, 0($10)
sw $3, 0($20)      ] WAW

lw $2, 0($10)
sw $3, 0($20)      ] WAR
```

Instruction scheduling

- What limits ILP?
  - Control dependence
    - Data flow alone in not sufficient for program correctness
    - Control flow must also be preserved
    - Branch prediction
    - What if you go wrong?

Instruction scheduling

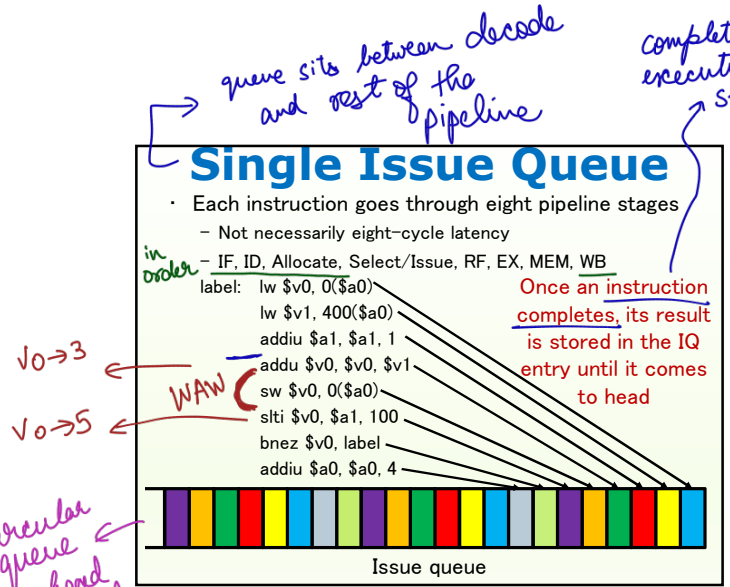
- Data hazards may introduce *unnecessary* stalls

```
div.d   $f0, $f2, $f4
add.d   $f10, $f0, $f8
sub.d   $f12, $f8, $f14  // really independent
```

- Limitations
  - In-order issue and execution
  - Precise exceptions
  - Small number of functional units
- Want out-of-order execution and still maintain precise exception
  - IF, ID, Allocate, Select/Issue, RF, EX, MEM, WB
  - WAW and WAR hazards: they are back!

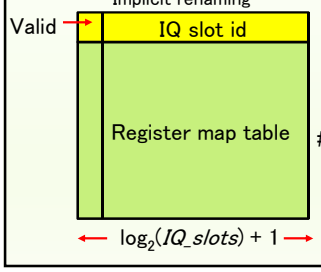
Single Issue Queue

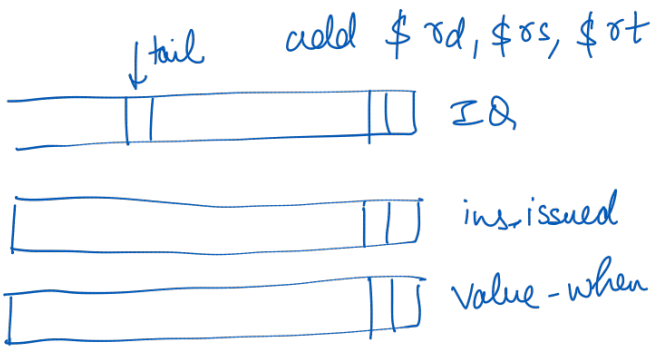
- Each instruction goes through eight pipeline stages
  - Not necessarily eight-cycle latency
  - IF, ID, Allocate, Select/Issue, RF, EX, MEM, WB



Single Issue Queue

- Each instruction goes through eight pipeline stages
  - Not necessarily eight-cycle latency
  - IF, ID, Allocate, Select/Issue, RF, EX, MEM, WB
  - Register to issue queue slot map
    - Implicit renaming





during WB, update the map table only if map table me use reg Ki slot id matches slot id of current instr.

Single Issue Queue

- Each instruction goes through eight pipeline stages
  - Not necessarily eight-cycle latency
  - IF, ID, Allocate, Select/Issue, RF, EX, MEM, WB
  - An instruction at the time of allocation needs to know if the parent instruction(s) has/have already issued
    - An *ins_issued* bitvector and *value_when* vector of length *IQ_slots* keeps track of that
    - When instruction in slot *i* is issued, *ins_issued*[*i*] is set to 1 and *value_when*[*i*] is set to a bit pattern encoding how many cycles later a dependent can be issued so that the dependent can pick up the value from bypass network
      - The bit pattern is shifted right every cycle
    - When a new instruction is allocated in slot *i*, *ins_issued*[*i*] is set to zero

00010

next to next cycle we can get the value of that reg when val becomes 0 we can take the value from bypass network

Single Issue Queue

- Each instruction goes through eight pipeline stages
  - Not necessarily eight-cycle latency
  - IF, ID, Allocate, Select/Issue, RF, EX, MEM, WB
  - Register to issue queue slot map
    - Implicit renaming
    - Handle race between allocate and wakeup
    - Handle race between allocate and WB
  - Memory renaming through issue queue slots
- Want large issue queue to expose ILP, but
  - Large searchable structures are slow due to long wires and power-hungry due to large switching capacitance
- Solution: distribute issue queue to respective functional units
  - Distributes the search over multiple smaller queues

-ve half +ve half

parent is issuing and child is allocating

child getting allocated and parent writing back

or opposite:

+ve: allocate

-ve: wake up/ WB

WAW, RAW

used today

comparators: 2 × IQ-slots × issue width

# of instrs issued in a cycle

WAW

```
sw $2, 0($10)
sw $20, 0($12)
```

RAW

```
sw $2, 0($10)
lw $5, 0($12)
```

```
lw $2, 0($10)
sw $5, 0($12)
```

WAR

store value in IQ slot and only write in order. so earlier instr will read correct value.

Single Issue Queue

- Fields in one issue queue entry
  - Functional unit id and decoded op
  - Source register ids and Immediate operand
  - Destination register id
  - Two parent queue slot ids
  - Parent queue slot ready (one bit for each parent slot), when
    - If both are ready, the instruction is ready to issue
  - Read from RF or parent queue slot id (one bit for each register source)
  - Computed value / Store value / Predicted branch target
  - Done bit
  - Store/Load address / Computed branch target
  - Store/Load address valid bit
  - Exception vector

Single Issue Queue

- Fields in one register map table entry
  - Valid bit
  - Queue slot id
- An instruction is eligible for selection when it is ready
  - An issued instruction
    - Resets the parent ready bits
    - Wakes up dependents according to the wakeup protocol (i.e., sets the ready bit in map table if the entry matches its slot id and compares its slot id with parent slot ids of all queue entries); additional stalls?
    - Reads operands from its parent slots and/or RF as indicated by the "read from" bit and proceeds to its functional unit

Single Issue Queue

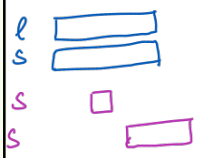
- An instruction on completing execution
  - Sets the done bit
  - Stores the computed value in its queue entry
    - A control transfer instruction stores the computed target
    - A control transfer instruction also invokes misprediction recovery at this point if the computed target does not match the predicted target
- An issuing store instruction only reads the value to be stored and computes the address
  - These are stored in its queue entry and the done bit is also set
  - The actual store (i.e., access to memory) happens when the instruction moves to the head of the issue queue

detects cache hit/miss and starts fetching so that some time is overlapped.

Single Issue Queue

- The execution of a load instruction is more involved due to memory dependencies
  - A load selected for issue checks if all stores before it have their done bits set
    - If not, it doesn't issue and depending on the issue protocol, it may keep on trying to issue in subsequent cycles
    - If yes, it issues, computes its address, compares the address and size with those of each store before it
      - If there is a full match (i.e., starting address and size), the load picks up the value from the store; ties are broken in favor of the youngest store before the load (load forwarding)
      - If there is a partial match, theoretically the load can access memory and merge the values; an easy (but lower-performance) solution is to stall the load and issue it when all stores before it have written back
      - If there is no match, the load proceeds to access memory

size = lw/lb/lh



Single Issue Queue

- When an instruction reaches the head of the issue queue and its done bit is set, it can commit (WB)
  - Its exception vector is checked and if set, all issue queue entries are marked invalid (by merging the tail and head pointers) and the fetcher is directed to fetch a special trap instruction that will transfer control to the OS
    - Need to fix the register map table (how?)
  - A store instruction is sent to memory
  - A control transfer instruction updates relevant predictors
  - Value producing instructions write their results back to their destination registers
    - Resets valid bit in map table if its slot id matches the entry of the destination register in the map table
    - Compares its slot id with parent slot ids of all queue entries and toggles the "read from" bit accordingly

invalidate all of those

when does PC update

Single Issue Queue

- Complexity of implementation
  - Number of comparators depend on size of issue queue, issue width, and commit width
  - Two sets of comparators, one enabled during issue and one during commit, cause inconvenience
    - Need a better solution that can eliminate one of these
    - Arises due to two possible places of finding a value: RF and issue queue entry
      - Need to merge these using some protocol: leads to register renaming implemented in today's processors
  - Issue width is limited by the number and mix of functional units, register file read ports, data memory read ports
  - Commit width is limited by register file write ports and data memory write ports

instr we can issue in a cycle

#instr we can commit in one cycle

issued instr — 2x #(IQ len)
committed instr — 1x

2x IQL x (CW + IW) + comp for load stores

Single Issue Queue

- Branch misprediction recovery
  - Easy to handle if delayed until commit like exceptions
    - Lower-performance because the commit of a branch may get delayed due to other long-latency unrelated instructions
      - The processor continues to fetch along the wrong path
  - Want to handle it as soon as the misprediction is discovered
    - Several instructions after it may have completed execution, but still not committed
    - Several instructions before it may not have issued
      - We are not worried about these instructions because they are correct
    - Invalidate all instructions after the branch by bringing the tail pointer of the queue and bitvector/vector forward, redirect fetcher, fix register map from a checkpoint (why?)

copy

ins_issued

value_when

$p^n > 0.5$

Single Issue Queue

- Each instruction goes through eight pipeline stages
  - Not necessarily eight-cycle latency
  - IF, ID, Allocate, Select/Issue, RF, EX, MEM, WB

label: lw $v0, 0($a0)
lw $v1, 400($a0)
addiu $a1, $a1, 1
addu $v0, $v0, $v1
sw $v0, 0($a0)
slti $v0, $a1, 100
bnez $v0, label
addiu $a0, $a0, 4

Ideally, one iteration can complete in three cycles ignoring load delay interlock: 1, 2, 3

Issue queue

Single Issue Queue

- Pipeline diagram for *hypothetical* three-cycle schedule
  - Note: IF, ID, Allocate, WB stages are not shown as these must be done in order

label: lw $v0, 0($a0) IS RF EX MEM
lw $v1, 400($a0) IS RF EX MEM
addiu $a1, $a1, 1 IS RF EX MEM
addu $v0, $v0, $v1 IS RF EX MEM
sw $v0, 0($a0) IS RF EX MEM
slti $v0, $a1, 100 IS RF EX MEM
bnez $v0, label IS RF EX MEM
addiu $a0, $a0, 4 IS RF EX MEM

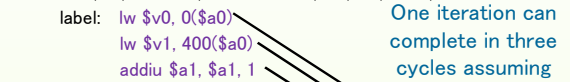
Bypass paths are shown in RED
Dotted bypass path is an alternate path
Unrealizable bypass paths are shown in BLACK

Single Issue Queue

- Each instruction goes through eight pipeline stages
  - Not necessarily eight-cycle latency
  - IF, ID, Allocate, Select/Issue, RF, EX, MEM, WB

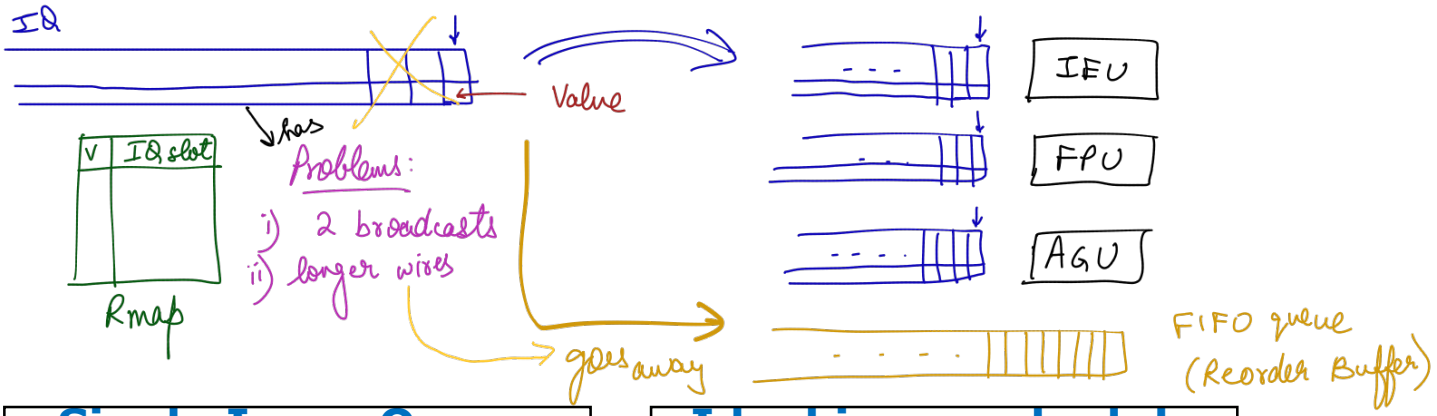
label: lw $v0, 0($a0)
lw $v1, 400($a0)
addiu $a1, $a1, 1
addu $v0, $v0, $v1
sw $v0, 0($a0)
slti $v0, $a1, 100
bnez $v0, label
addiu $a0, $a0, 4

One iteration can complete in three cycles assuming cache hits: 1, 2, 3



Issue queue

What happens to queues on context switch?



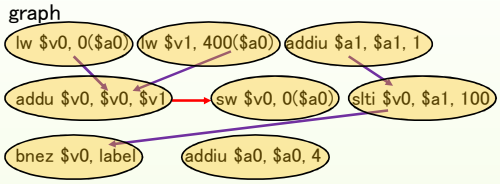
Single Issue Queue

- Pipeline diagram for three-cycle issue schedule
  - Note: IF, ID, Allocate, WB stages are not shown as these must be done in order

```
label: lw $v0, 0($a0)      IS  RF  EX  MEM
       lw $v1, 400($a0)    IS  RF  EX  MEM
       addiu $a1, $a1, 1   IS  RF  EX  MEM
       addu $v0, $v0, $v1      IS  RF  EX  MEM
       sw $v0, 0($a0)          IS  RF  EX  MEM
       slti $v0, $a1, 100  IS  RF  EX  MEM
       bnez $v0, label         IS  RF  EX  MEM
       addiu $a0, $a0, 4   IS  RF  EX  MEM
```

Ideal issue schedule

- Ideal issue schedule is derived from the data flow graph



Vertices are instructions and edges are data dependencies

Number of vertices on the longest path through the DAG is the shortest theoretical issue length = # of cycles req

Early Implementation: Scoreboard

- First introduced in CDC 6600
  - Handles RAW hazards dynamically (data flow)
  - Stalls on WAW and WAR hazards (can't rename regs)
  - Issue is still in-order
  - Scoreboard determines when an instruction can execute based on operand availability
  - WAW hazards stall the issue unit
  - WAR hazards are detected during WB and completed instructions are delayed

WAR

```
add $3, $2, $1
 :
add $2, ....
```

Tomasulo's algorithm

- Significant enhancement over scoreboards
  - Distributes the scoreboard to respective functional units: known as reservation stations (issue queues)
  - Resolves name dependences by using the reservation station entries
  - After an instruction is registered in a RS all dependences generated by this instruction are mentioned in terms of RS[k].rs, RS[k].rt, RS[k].rd
  - WB to register file and cache must still be in-order
  - Pending results can be held in RS entry or a future file
  - Bypass network takes the form of a common bus
  - All launched results must compare *all* pending instructions' sources (quite a bit of hardware)
  - Retirement register file (RRF) of P6 microarchitecture (used in Pentium Pro, II, III) is very similar

→ value field of issue queue

CPI below one

- Goal of static or dynamic scheduling is to have CPI less than 1.0
  - Reorders instructions to reduce the number of interlock cycles
- Major elements of a dynamically scheduled pipeline
  - Good branch predictor
    - Makes sure that the later part of the pipeline works on useful instructions and minimizes wasted work
  - Multiple issue: must fetch, decode, select, and issue multiple instructions every cycle (superscalar)
  - Wakeup logic: preserves data flow; bank of comparators
  - Selection logic: in-order and out-of-order selection
    - Obeys issue constraints such as available RF read ports, memory read ports, and functional units reg or mem read
  - Decoupled execution and commit
    - In-order writeback and store completion

Multiple issue

- Ideally CPI should go down by a factor equal to minimum of issue and commit width in steady-state
  - Assuming full renaming (i.e. no name dependence) only dataflow and control flow limit CPI
  - Also there could be structural hazards (insufficient FUs)
  - Two possible selection algorithms
    - In-order selection
    - Out-of-order selection: need tie-breakers

Might need to pick a subset out of all ready insts due to structural hazards

In-order multi-issue

- Simplest possible design
  - Issue the instructions sequentially (in-order)
  - Scan the issue queue, stop as soon as you come to an instruction which cannot be executed now
    - Possibly due to pending operands or insufficient FUs

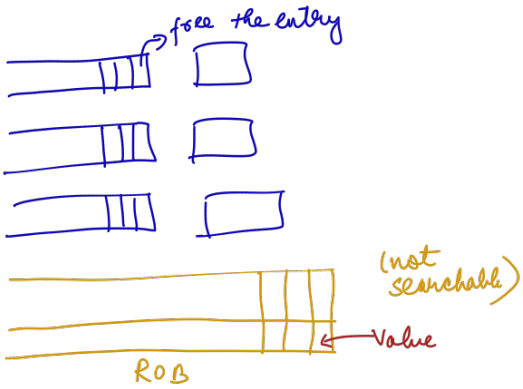
add r4, r5, r6
sub r7, r3, r6
xor r27, r4, r1
and r6, r18, r19
lw r3, 0(r21)

- Cannot issue the last two even though they are independent of the first two

Out-of-order issue

lw r4, 0(r6) Cache miss
addi r5, r4, 0x20
and r10, r5, r19
xor r26, r2, r7
sub r20, r26, r2
andi r27, r8, 0xffff
sll r19, r27, 0x5
beq r20, r19, label
or r12, r15, r16

- Issue first cycle, issue second cycle,…



WAR hazard

lw r4, 0(r6) Cache miss
addi r5, r4, 0x20
and r10, r5, r19
xor r26, r2, r7
sub r20, r26, r2
andi r27, r8, 0xffff
sll r19, r27, 0x5
beq r20, r19, label
or r12, r15, r16

- Write After Read (WAR): in-order commit solves it (need space to buffer results)

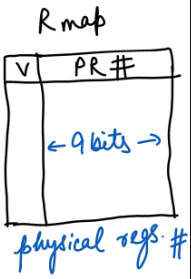
Speculative execution

- Major elements
  - Resolve control dependences by predicting
  - Continue execution past predicted branches (i.e. you not only fetch from predicted path, but also execute them if they are data-independent)
  - Buffer results in some structure (called re-order buffer or active list; allocated together with the issue queue entry)
    - Essentially decouples the value field from the issue queue and allows early recycling of issue queue entries
  - Typically the issue queue is distributed among different types of FUs (known as reservation stations)
    - Eliminates unnecessary comparisons during wakeup e.g., partition between integer and floating-point, etc.
  - Write or *commit* results to register file or memory only when an instruction comes to the top of ROB

AL = active list
RRF = retirement register file

Register renaming

- Recall the problems with our naïve register renaming through issue queue slots (or ROB slots)
- Registers visible to the compiler
  - Logical or architectural registers
  - 32 in number for MIPS, 8 for x86, and is fixed by the ISA
- Physical registers inside the processor
  - Much larger in number, not visible to compiler
- The destination logical register of every instruction is assigned a new physical register number
- The dependencies are tracked based on the namespace of physical registers
- MIPS R10000 has 32 logical and 64 physical regs
- Intel Pentium 4 has 8 logical and 128 physical regs

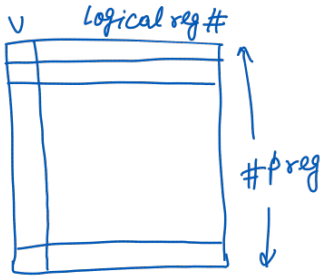


Register renaming

- Assume 64 physical regs and already renamed registers: r6=p54, r19=p38, r2=p0, r7=p20, r15=p3, r16=p23.

lw r4, 0(r6)	*lw* p15, 0(p54) [r4 renamed to p15]
addi r5, r4, 0x20	*addi* p40, p15, 0x20 [r5 renamed to p40]
and r10, r5, r19	*and* p39, p40, p38 [r10 renamed to p39]
xor r26, r2, r7	*xor* p62, p0, p20 [r26 renamed to p62]
sub r20, r26, r2	*sub* p8, p62, p0 [r20 renamed to p8]
andi r27, r8, 0xffff	*andi* p19, p25, 0xffff [r27 renamed to p19]
sll r19, r27, 0x5	*sll* p45, p19, 0x5 [r19 renamed to p45]
beq r20, r19, label	*beq* p8, p45, label
or r12, r15, r16	*or* p59, p3, p23 [r12 renamed to p59]

inverted Rmap table



we just need to copy the set of valid rows for roll back

in/validate on allocate

Register renaming

mult r5, r4, r3 [r5 gets renamed to, say, p50]

add r5, r6, r12 [r5 gets renamed to, say, p45]

- Now it is safe to issue them in parallel: they are really independent (compiler introduced WAW)
- Register renaming maintains a map table that records logical register to physical register map
- After an instruction is decoded, its logical register numbers are available
- The renamer looks up the map table to find mapping for the logical source regs of this instruction, assigns a free physical register to the destination logical reg, and records the new mapping

A modern pipeline

- Fetch, decode, rename/allocate, select/issue, register file read, EX, memory, retire/commit
- Fetch, decode, rename/allocate are in-order stages, each handles multiple instructions every cycle
- Select/Issue, RF, EX, memory are out-of-order
  - An instruction directly updates the destination physical register contents on completion (doesn't wait till commit)
    - Out-of-order register update (why is it correct?)
- Retire is again in-order, but multiple instructions may retire each cycle; register WB is not part of this
- When an instruction issues it broadcasts its destination physical register id to all instructions in issue queue(s)

stores will move values to cache in retirement not before

so that parent[ins.issued] = 1 and children can read value_when of parent and become ready accordingly.

Speculative execution

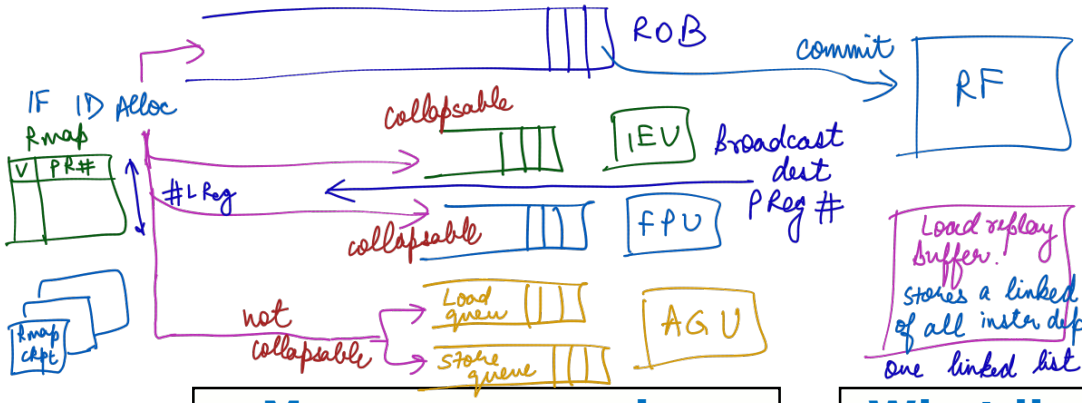
- With register renaming
  - Commit is simpler: no need to transfer values from ROB to RF; in fact no computed value is stored in ROB
    - Branches update predictors, BTB, RAS
    - Killed instructions just drain out
    - Stores write to memory (store queue or speculative store buffer holds the value)
    - ROB entry is recycled
      - Issue queue entries are recycled when instructions complete (except load and store issue queue entries)
  - Branch misprediction recovery needs to restore the register map (in addition to marking all instructions on wrong path as killed)
  - Branches, when going through renaming, checkpoint the map (not the values): limits the number of outstanding branches

Register recycling

- If the renamer runs out of physical registers, the pipeline stalls until at least one register is available
  - Physical registers must be recycled
  - When is it safe to free a physical register?
  - What about wrong path instructions?
- More physical registers ⇨ more in-flight instructions ⇨ possibility of more parallelism
- But cannot make the register file very big
  - Takes time to access
  - Burns power
- Relationship between ROB size and register file size? (Today register file size means number of physical registers)

$|PR| \geq |ROB| + |L\ Reg|$

when ROB is empty



Memory renaming

- Store address renaming is still done through store queue entries
  - Different stores to overlapping addresses can issue out-of-order and compute addresses and read values to be stored from RF
  - Stores commit in-order and this is when memory is updated with the new value
- Load selection protocol
  - Issues out-of-order when ready and all stores before it have already computed address
    - Improvement: speculative memory disambiguation
- Load execution happens in two phases
  - Computes address
  - Compares address with all stores before it

Store queue entry is bigger than others coz it needs to carry value to reg.

SW $2, 0($10)

LW $3, 0($20)

if $10 = $20 then when store is issued then we get to know load was wrongly issued.

Broadcast only once, at time of issue

What limits ILP now?

- Instruction fetch latency (need an instruction cache)
- Branch misprediction control dependence
  - Observe that you predict a branch in decode, and the branch executes in EX stage
  - Several pipeline stages before outcome is known
  - Misprediction amounts to loss of at least nF instructions where F is the fetch width (assume n-cycle penalty)
- Slow memory data dependence
  - Assuming an issue width of 4, frequency of 3 GHz, memory latency of 120 ns, you need to find 1440 independent instructions to issue so that you can hide the memory latency: this is impossible (resource shortage)
- Resource constraints: ROB size, register file size, issue queue size

120 × 3 = 360 cycles

360 × 4 = 1440 instructions

Cycle time reduction

- Execution time = CPI × instruction count × cycle time
- Talked about CPI reduction or improvement in IPC (instructions retired per cycle)
- Cycle time reduction is another technique to boost performance
  - Faster clock frequency
- Pipelining poses a problem
  - Each pipeline stage should be one cycle for balanced progress
  - Need to break pipe stages into smaller stages

ins ct ⇒ ISA and compiler

CPI ⇒ ILP

Cycle time reduction

- Superpipelining
  - Faster clock frequency necessarily means deep pipes
  - Each pipe stage contains small amount of logic so that it fits in small cycle time
  - May severely degrade CPI if not careful
  - Now branch penalty is even bigger (31 cycles for Intel Prescott): branch mispredictions cause massive loss in performance (93 micro-ops are lost, F=3)
  - Long pipes also put more pressure on resources such as ROB and registers because instruction latency increases (in terms of cycles, not in absolute terms)
  - Instructions occupy ROB entries and registers longer
  - The design becomes increasingly complicated (wire delay doesn't scale)

CPI related to cycle time

if deepen pipeline before EX ⇒ branch mispenalty ↑
if after it then bypass network inc.

An alternative

- Deeper pipeline or wider issue?
  - Deeper pipeline gives you faster clock at the expense of increased branch penalty, possibly wider bypass
  - Wider issue reduces CPI by exposing more parallelism

P6 microarchitecture

- Introduced in Pentium Pro, extended with MMX/SSE in Pentium II and III
  - 14-stage RISC pipe: eight stages spent in front-end (fetch: 2, decode: 3, rename: 1, ROB allocate: 1, dispatch to RS: 1), 3 stages in execution unit, 3 stages in commit; for lengthy instructions execution stage gets elongated
  - Fetches 3 IA32 instructions every cycle (up to 16 bytes)
  - Decoder translates these into six RISC-like micro-ops: four allocated to the first, one each to the next two; if any instruction needs more than four micro-ops, it is handled by microinstruction sequencer
  - Renames six micro-ops every cycle (no register renaming, RRF is attached to ROB; also known as RUU)
  - Allocates six ROB entries (out of 40; same size for RRF) every cycle

one of them will be 4.

Register Update Unit

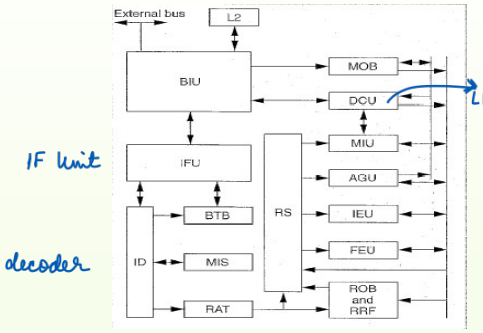
→ micro ops

P6 microarchitecture

- Unified 20-entry reservation station
- Register alias table to keep track of logical register to RRF map
- Between two consecutive front-end stages there is a queue so that some slack can be afforded
- Fetcher uses a BTB and a two-level SAg predictor (4-bit history with each BTB set); 4-way 512-entry BTB
- On a BTB miss static prediction is used: FNBT
- 10-cycle branch penalty
- Commits 3 micro-ops per cycle
- Maximum operating frequency 200 MHz

Single IQ

P6 microarchitecture



IF unit

decoder

Reproduced from IEEE Micro