CSE 502: Computer Architecture

Out-of-Order Schedulers
Data-Capture Scheduler

- Dispatch: read available operands from ARF/ROB, \textit{store} in scheduler
- Commit: Missing operands filled in from bypass
- Issue: When ready, operands sent directly from scheduler to functional units
Components of a Scheduler

- Buffer for unexecuted instructions
- Method for tracking state of dependencies (resolved or not)
- Arbiter
  - Method for choosing between multiple ready instructions competing for the same resource
- Method for notification of dependency resolution

“Scheduler Entries” or “Issue Queue” (IQ) or “Reservation Stations” (RS)
Scheduling Loop or **Wakeup-Select Loop**

- **Wake-Up Part:**
  - Executing insn notifies dependents
  - Waiting insns. check if all deps are satisfied
    - If yes, “wake up” instruction

- **Select Part:**
  - Choose which instructions get to execute
    - More than one insn. can be ready
    - Number of functional units and memory ports are limited
Scalar Scheduler (Issue Width = 1)
Superscalar Scheduler (detail of one entry)
Interaction with Execution

[Diagram showing interaction between select logic and payload RAM with labels D, S_L, S_R, A, opcode, Val_L, Val_R, etc.]
Again, But Superscalar

Scheduler *captures* values
Issue Width

• Max insns. selected each cycle is *issue width*
  – Previous slides showed different issue widths
    • four, one, and two

• Hardware requirements:
  – Naively, issue width of N requires N tag broadcast buses
  – Can “specialize” some of the issue slots
    • E.g., a slot that only executes branches (no outputs)
Simple Scheduler Pipeline

A: Select | Payload | Execute

- tag broadcast

B: Wakeup | Capture | Select | Payload | Execute

- enable capture on tag match
- result broadcast

C: Wakeup | Capture

- tag broadcast
- enable capture

Cycle i | Cycle i+1

Very long clock cycle
Deeper Scheduler Pipeline

A:
- Select
- Payload
- Execute
- tag broadcast
- result broadcast

B:
- Wakeup
- Capture
- enable broadcast
- enable capture

C:
- Wakeup
- Capture
- Enable capture
- tag broadcast

Cycle i
Cycle i+1
Cycle i+2
Cycle i+3

Faster, but Capture & Payload on same cycle
Even Deeper Scheduler Pipeline

A: Select → Payload → Execute
- Tag broadcast
- Result broadcast and bypass

B: Wakeup
- Select → Payload → Execute
- Enable capture
- Capture

C: Wakeup
- Select → Payload → Execute
- Capture
- Capture

Cycle i
- Cycle i+1
- Cycle i+2
- Cycle i+3
- Cycle i+4

No simultaneous read/write!

Need second level of bypassing
Very Deep Scheduler Pipeline

A: Select → Payload → Execute
B: Select → Select → Payload → Execute
C: Wakeup → Capture → Select → Payload → Execute
D: Wakeup → Capture → Capture → Select → Payload → Execute

A → C and C → D must be bypassed, B → D OK without bypass

Dependent instructions can’t execute back-to-back
Pipelineing Critical Loops

• Wakeup-Select Loop hard to pipeline
  – No back-to-back execute
  – Worst-case IPC is $\frac{1}{2}$

• Usually not worst-case
  – Last example had IPC $\frac{2}{3}$

Studies indicate 10-15% IPC penalty
IPC vs. Frequency

• 10-15% IPC not bad if frequency can double

  1000ps
  2.0 IPC, 1GHz
  2 BIPS

  500ps 500ps
  1.7 IPC, 2GHz
  3.4 BIPS

• Frequency doesn’t double
  – Latch/pipeline overhead
  – Stage imbalance

  900ps
  450ps 450ps
  1.5GHz

  900ps
  350 550
Non-Data-Capture Scheduler

- Fetch & Dispatch
- Scheduler
  - ARF
  - PRF
  - Functional Units

Unified PRF

- Fetch & Dispatch
- Scheduler
  - Unified PRF
  - Functional Units

Physical register update
Pipeline Timing

Data-Capture

Select → Payload → Execute
Wakeup → Select → Payload → Execute

Non-Data-Capture

Select → Payload → Read Operands from PRF → Execute
Wakeup → Select → Payload → Read Operands from PRF → Exec

Substantial increase in schedule-to-execute latency

“Skip” Cycle
Handling Multi-Cycle Instructions

Instructions can’t execute too early.

Add R1 = R2 + R3
Xor R4 = R1 ^ R5
Mul R1 = R2 × R3
Add R4 = R1 + R5
Delayed Tag Broadcast (1/3)

- Must make sure broadcast bus available in future
- Bypass and data-capture get more complex
Delayed Tag Broadcast (2/3)

Assume issue width equals 2

In this cycle, three instructions need to broadcast their tags!
Delayed Tag Broadcast (3/3)

- Possible solutions

1. One select for issuing, another select for tag broadcast
   - Messes up timing of data-capture

2. Pre-reserve the bus
   - Complicated select logic, track future cycles in addition to current

3. Hold the issue slot from initial launch until tag broadcast

   ![Diagram](sch payl exec exec exec)

   Issue width effectively reduced by one for three cycles
Delayed Wakeup

• Push the delay to the consumer

Tag Broadcast for $R1 = R2 \times R3$

Tag arrives, but we wait three cycles before acknowledging it

$R5 = R1 + R4$

ready!

Must know ancestor’s latency
Non-Deterministic Latencies

• Previous approaches assume all latencies are known
• Real situations have unknown latency
  – Load instructions
    • Latency ∈ \{L1_{lat}, L2_{lat}, L3_{lat}, DRAM_{lat}\}
    • DRAM_{lat} is not a constant either, queuing delays
  – Architecture specific cases
    • PowerPC 603 has “early out” for multiplication
    • Intel Core 2’s has early out divider also

• Makes delayed broadcast hard
• Kills delayed wakeup
The Wait-and-See Approach

- Complexity only in the case of variable-latency ops
  - Most insns. have known latency
- Wait to learn if load hits or misses in the cache

\[ R_1 = 16[\$sp] \]
\[ R_2 = R_1 + \#4 \]

May be able to design cache s.t. hit/miss known before data

Load-to-Use latency increases by 2 cycles (3 cycle load appears as 5)

Penalty reduced to 1 cycle
Load-Hit Speculation

- Caches work pretty well
  - Hit rates are high (otherwise we wouldn’t use caches)
  - Assume all loads hit in the cache

What to do on a cache miss?

\[ \text{R1} = 16[\$sp] \]

\[ \text{R2} = \text{R1} + \#4 \]
Load-Hit Mis-speculation

Each mis-scheduling wastes an issue slot: the tag broadcast bus, payload RAM read port, writeback/bypass bus, etc. could have been used for another instruction.

There could be a miss at the L2 and again at the L3 cache. A single load can waste multiple issuing opportunities.

It’s hard, but we want this for performance.
“But wait, there’s more!”

Not only children get squashed, there may be grand-children to squash as well.

All waste issue slots
All must be rescheduled
All waste power
None may leave scheduler until load hit known

L1-D Miss
Squashing (1/3)

• Squash “in-flight” between schedule and execute
  – Relatively simple (each RS remembers that it was issued)

• Insns. stay in scheduler
  – Ensure they are not re-scheduled
  – Not too bad
    • Dependents issued in order
    • Mis-speculation known before Exec

May squash non-dependent instructions
Squashing (2/3)

• Selective squashing with "load colors"
  – Each load assigned a unique color
  – Every dependent "inherits" parents’ colors
  – On load miss, the load broadcasts its color
    • Anyone in the same color group gets squashed

• An instruction may end up with many colors

Tracking colors requires huge number of comparisons
Squashing (3/3)

- Can list “colors” in unary (bit-vector) form
  - Each insn.’s vector is bitwise OR of parents’ vectors

\[
\begin{align*}
\text{Load } R1 &= 16[R2] \\
\text{Add } R3 &= R1 + R4 \\
\text{Load } R5 &= 12[R7] \\
\text{Load } R8 &= 0[R1] \\
\text{Load } R7 &= 8[R4] \\
\text{Add } R6 &= R8 + R7
\end{align*}
\]

Allows squashing just the dependents
Scheduler Allocation (1/3)

- Allocate in order, deallocate in order
  - Very simple!
- Reduces effective scheduler size
  - Insns. **executed** out-of-order
    ... RS entries cannot be reused

Can be terrible if load goes to memory
Scheduler Allocation (2/3)

• Arbitrary placement improves utilization
• Complex allocator
  – Scan availability to find N free entries
• Complex write logic
  – Route N insns. to arbitrary entries
Scheduler Allocation (3/3)

• Segment the entries
  – One entry per segment may be allocated per cycle
  – Each allocator does 1-of-4
    • instead of 4-of-16 as before
  – Write logic is simplified
• Still possible inefficiencies
  – Full segments block allocation
  – Reduces dispatch width
Select Logic

• Goal: minimize DFG height (execution time)
• NP-Hard
  – Precedence Constrained Scheduling Problem
  – Even harder: entire DFG is not known at scheduling time
    • Scheduling insns. may affect scheduling of not-yet-fetched insns.
• Today’s designs implement heuristics
  – For performance
  – For ease of implementation
Simple Select Logic

Scheduler Entries

\[ x_i = \text{Bid}_i \]

\[ \text{grant}_i \]

S entries yields \( O(S) \) gate delay

\[ \text{Grant}_0 = 1 \]
\[ \text{Grant}_1 = \neg \text{Bid}_0 \]
\[ \text{Grant}_2 = \neg \text{Bid}_0 \land \neg \text{Bid}_1 \]
\[ \text{Grant}_3 = \neg \text{Bid}_0 \land \neg \text{Bid}_1 \land \neg \text{Bid}_2 \]
\[ \text{Grant}_{n-1} = \neg \text{Bid}_0 \land \cdots \land \neg \text{Bid}_{n-2} \]

\( O(\log S) \) gates

\[ \text{grant}_0 \]
\[ \text{grant}_1 \]
\[ \text{grant}_2 \]
\[ \text{grant}_3 \]
\[ \text{grant}_4 \]
\[ \text{grant}_5 \]
\[ \text{grant}_6 \]
\[ \text{grant}_7 \]
\[ \text{grant}_8 \]
\[ \text{grant}_9 \]
Random Select

- Insns. occupy arbitrary scheduler entries
  - First ready entry may be the oldest, youngest, or in middle
  - Simple static policy results in “random” schedule
    - Still “correct” (no dependencies are violated)
    - Likely to be far from optimal
Oldest-First Select

• Newly dispatched insns. have few dependencies
  – No one is waiting for them yet

• Insns. in scheduler are likely to have the most deps.
  – Many new insns. dispatched since old insn’s rename

• Selecting *oldest* likely satisfies more dependencies
  – ... finishing it sooner is likely to make more insns. ready
Implementing Oldest First Select (1/3)

Write instructions into scheduler in program order

Compress Up

Newly dispatched
Implementing Oldest First Select (2/3)

- Compressing buffers are very complex
  - Gates, wiring, area, power

Ex. 4-wide
Need up to shift by 4

An entire instruction’s worth of data: tags, opcodes, immediates, readiness, etc.
Implementing Oldest First Select (3/3)

Must broadcast grant age to instructions
Problems in N-of-M Select (1/2)

N layers $\rightarrow O(N \log M)$ delay

O(log M) gate delay / select
Problems in N-of-M Select (2/2)

- Select logic handles functional unit constraints
  - Maybe two instructions ready this cycle
    ... but both need the divider

Assume issue width = 4

Four *oldest and ready* instructions

ADD is the 5th oldest ready instruction, but it should be issued because only one of the ready divides can issue this cycle
### Partitioned Select

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIV</td>
<td>1</td>
</tr>
<tr>
<td>LOAD</td>
<td>5</td>
</tr>
<tr>
<td>XOR</td>
<td>3</td>
</tr>
<tr>
<td>MUL</td>
<td>6</td>
</tr>
<tr>
<td>DIV</td>
<td>4</td>
</tr>
<tr>
<td>ADD</td>
<td>2</td>
</tr>
<tr>
<td>BR</td>
<td>7</td>
</tr>
<tr>
<td>ADD</td>
<td>8</td>
</tr>
</tbody>
</table>

- **Add(2)**
- **Div(1)**
- **Load(5)**
- **Store**

**DLI**

- **Load**
- **Store**
- **Idle**

- **N possible insns. issued per cycle**

- **5 Ready Insts**
- **Max Issue = 4**
- **Actual issue is only 3 insts**
Multiple Units of the Same Type

Possible to have multiple popular FUs
Bid to Both?

No! Same inputs → Same outputs
Chain Select Logics

Works, but doubles the select latency
Select Binding (1/2)

During dispatch/alloc, each instruction is bound to one and only one select logic.

- **ADD**: 5, 4, 1
- **XOR**: 2, 2
- **SUB**: 8, 1
- **ADD**: 4, 1
- **CMP**: 7, 2
Select Binding (2/2)

Not-Quiite-Oldest-First: 
Ready insns are aged 2, 3, 4 
Issued insns are 2 and 4

Wasted Resources: 
3 instructions are ready 
Only 1 gets to issue
Make N Match Functional Units?

Too big and too slow
Execution Ports (1/2)

• Divide functional units into P groups
  – Called “ports”
• Area only $O(P^2M \log M)$, where $P \ll F$
• Logic for tracking bids and grants less complex (deals with P sets)
Execution Ports (2/2)

- More wasted resources
- Example
  - SHL issued on Port 0
  - ADD cannot issue
  - 3 ALUs are unused
Port Binding

- Assignment of functional units to execution ports
  - Depends on number/type of FUs and issue width

8 Units, N=4

Int/FP Separation
Only Port 3 needs to access FP RF and support 64/80 bits

Even distribution of Int/FP units, more likely to keep all N ports busy

Each port need not have the same number of FUs; should be bound based on frequency of usage
Port Assignment

• Insns. get port assignment at dispatch

• For unique resources
  – Assign to the only viable port
  – Ex. Store must be assigned to Port 1

• For non-unique resources
  – Must make intelligent decision
  – Ex. ADD can go to any of Ports 0, 1 or 2

• Optimal assignment requires knowing the future

• Possible heuristics
  – random, round-robin, load-balance, dependency-based, ...
Decentralized RS (1/4)

- Area and latency depend on number of RS entries.
- De-centralize the RS to reduce effects:

Select logic blocks for RS\(_i\) only have gate delay of \(O(\log M_i)\)
Decentralized RS (2/4)

- Natural split: INT vs. FP

Often implies non-ROB based physical register file:

One “unified” integer PRF, and one “unified” FP PRF, each managed separately with their own free lists
Decentralized RS (3/4)

- Fully generalized decentralized RS

- Over-doing it can make RS and select smaller ... but tag broadcast may get out of control

Can combine with INT/FP split idea
Decentralized RS (4/4)

• Each RS-cluster is smaller
  – Easier to implement, less area, faster clock speed

• Poor utilization leads to IPC loss
  – Partitioning must match program characteristics
  – Previous example:
    • Integer program with no FP instructions runs on 2/3 of issue width
      (ports 4 and 5 are unused)