CONTROL DATA 6400/6500/6600/6700
COMPUTER SYSTEMS
CENTRAL PROCESSOR INSTRUCTIONS

BRANCH UNIT
00  PS  Program Stop
010  XJ  Central Exchange Jump
013  XJ  Central Exchange Jump
02  JP  Go to K + Ji
030  ZR  Go to K if Xi = zero
031  NZ  Go to K if Xi ≠ zero
032  PL  Go to K if Xi > 0
033  NG  Go to K if Xi < 0
034  IR  Go to K if Xi is in range
035  OR  Go to K if Xi is out of range
036  DF  Go to K if Xi is definite
037  ID  Go to K if Xi is indefinite
04  EQ  Go to K if Ji = Ji
05  NE  Go to K if Ji ≠ Ji
06  GE  Go to K if Ji ≥ Ji
07  LT  Go to K if Ji < Ji

BOOLEAN UNIT
10  HX  Xj  Xmit XI to Xi
11  HX  Xj  X  Logical Product
12  HX  Xj  X  Logical Sum
13  HX  Xj  X  Logical Difference
14  HX  Xj  X  Xmit Xi comp to Xi
15  HX  Xj  X  Log Prod Xi & X comp
16  HX  Xj  X  Log Sum Xi & X comp
17  HX  Xj  X  Log Diff Xi & X comp

SHIFT UNIT
20  LX  jk  Left Shift Xi by jk
21  AX  jk  Right Shift Xi by jk
22  LX  Bj  Left Shift Xi by Bj to Xi
23  AX  Bj  Right Shift Xi by Bj to Xi
24  NIX  Bj  Normalize Xi to Xi & Bj
25  ZIX  Bj  Round & Normalize Xi
26  CIX  Bj  Unpack Xi to Xi & Bj
27  FXX  Bj  Xi  Pack Xi from Xi & Bj
28  MX  jk  Form jk mask in Xi

ADD UNIT
30  FXX  Xj  Xk  Xk Complement
31  FXX  Xj  X  Floating Diff
32  DXX  Xj  X  Floating DP Sum
33  DXX  Xj  X  Floating DP Diff
34  RXX  Xj  X  Round Floating Sum
35  RXX  Xj  X  Round Floating Diff

LONG ADD UNIT
36  LX  Xj  Xk  Integer Sum
37  LX  Xj  Xk  Integer Difference

MULTIPLY UNIT
40  FXX  Xj  Xk  Floating Product
41  RXX  Xj  Xk  Round Floating Product
42  DXX  Xj  Xk  Floating DP Product

DIVIDE UNIT
44  FXX  Xj/Xk  Floating Divide
45  RXX  Xj/Xk  Round Floating Divide
46  NO  No Operation
47  CXX  Xk  Sum of ones in Xi to Xi

INCREMENT UNIT
50  SAi  A j + K  Sum of Aj & K to Ai
51  SAi  Bj  Sum of Bj to Ai
52  SAi  Xj  K  Sum of Xi & K to Ai
53  SAi  Xj  X  Sum of Xi & Xi to Ai
54  SAi  A j + K  Sum of Aj & K to Ai
55  SAi  Aj  Diff of Aj & K to Ai
56  SAi  Bj  Sum of Bj & Bj to Ai
57  SAi  Bj  Aj  Diff of Bj & Aj to Ai
60  SBI  Aj  K  Sum of Aj & K to Bi
61  SBI  Bj  K  Sum of Bj & K to Bi
62  SBI  Xj  K  Sum of Xi & K to Bi
63  SBI  Xj  X  Sum of Xi & Xi to Bi
64  SBI  Aj  Bj  Diff of Aj & Bj to Bi
65  SBI  Bj  Aj  Diff of Bj & Aj to Bi
66  SBI  Bj  Aj  Diff of Bj & Bj to Bi
67  SBI  Bj  Bk  Diff of Bj & Bk to Bi

EXTENDED CORE STORAGE
001  Rdr  Bj  K  Read ECS
012  WE  Bj  K  Write ECS

EXIT MODE
00  0  Address out of range
01  1  Operate out of range (Infinite)
02  2  Indeterminate operand

SYMBOLS
Comp = Complement
DP = Double Precision
Xmit = Transmit
Complex Pipelining: Out-of-Order Execution, Register Renaming, and Exceptions

Joel Emer
Computer Science and Artificial Intelligence Laboratory
M.I.T.
CDC 6600-style Scoreboard

Instructions are issued in order. An instruction is issued only if
- It cannot cause a RAW hazard
- It cannot cause a WAW hazard

⇒ There can be at most one instruction in the execute phase that can write to a particular register

WAR hazards are not possible
- Due to in-order issue + operands read immediately

Scoreboard:
Two bit-vectors

Busy[FU#]: Indicates FU's availability
These bits are hardwired to FU's.

WP[reg#]: Records if a write is pending for a register
Set to true by the Issue stage and set to false by the WB stage
Scoreboard Dynamics

<table>
<thead>
<tr>
<th>Issue time</th>
<th>Functional Unit Status</th>
<th>Registers Reserved for Writes (WP)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Int(1) Add(1) Mult(3) Div(4)</td>
<td></td>
</tr>
<tr>
<td>t0</td>
<td>I₁</td>
<td>f6</td>
</tr>
<tr>
<td>t1</td>
<td>I₂ f2</td>
<td>f6, f2</td>
</tr>
<tr>
<td>t2</td>
<td>f6</td>
<td>f2</td>
</tr>
<tr>
<td>t3</td>
<td>I₃ f0</td>
<td>f6</td>
</tr>
<tr>
<td>t4</td>
<td>f0</td>
<td>f6</td>
</tr>
<tr>
<td>t5</td>
<td>I₄ f0 f8</td>
<td>f0, f8</td>
</tr>
<tr>
<td>t6</td>
<td>f8</td>
<td>f0</td>
</tr>
<tr>
<td>t7</td>
<td>I₅ f10</td>
<td>f8</td>
</tr>
<tr>
<td>t8</td>
<td>f8</td>
<td>f8, f10</td>
</tr>
<tr>
<td>t9</td>
<td>f8</td>
<td>f8, f10</td>
</tr>
<tr>
<td>t10</td>
<td>I₆ f6</td>
<td>f6</td>
</tr>
<tr>
<td>t11</td>
<td></td>
<td>f6</td>
</tr>
</tbody>
</table>

Instructions:
- I₁: DIVD
- I₂: LD
- I₃: MULTD
- I₄: DIVD
- I₅: SUBD
- I₆: ADDD

September 28, 2022
**In-Order Issue Limitations**

*An example*

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Source(s)</th>
<th>Destination(s)</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD F2, 34(R2)</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>LD F4, 45(R3)</td>
<td></td>
<td></td>
<td>long</td>
</tr>
<tr>
<td>3</td>
<td>MULTD F6, F4, F2</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>SUBD F8, F2, F2</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>DIVD F4, F2, F8</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>ADDD F10, F6, F4</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

In-order: 1 (2,1) . . . . . . . 2 3 4 4 3 5 . . . 5 6 6

In-order restriction prevents instruction 4 from being dispatched
Out-of-Order Issue

How can we address the delay caused by a RAW dependence associated with the next in-order instruction?

- Issue stage buffer holds multiple instructions waiting to issue.
- Decode adds next instruction to buffer if there is space and the instruction does not cause a WAR or WAW hazard.
- Can issue any instruction in buffer whose RAW hazards are satisfied (for now at most one dispatch per cycle).

*Note:* A writeback (WB) may enable more instructions.
In-Order Issue Limitations

An example

L07-7

<table>
<thead>
<tr>
<th>No.</th>
<th>Operation</th>
<th>F1, F2, F3</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD</td>
<td>F2, 34(R2)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>LD</td>
<td>F4, 45(R3)</td>
<td>long</td>
</tr>
<tr>
<td>3</td>
<td>MULTD</td>
<td>F6, F4, F2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>SUBD</td>
<td>F8, F2, F2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>DIVD</td>
<td>F4, F2, F8</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>ADDD</td>
<td>F10, F6, F4</td>
<td>1</td>
</tr>
</tbody>
</table>

In-order: 1 (2,1) . . . . . . . 2 3 4 4 3 5 . . . 5 6 6
Out-of-order: 1 (2,1) 4 4 . . . . 2 3 5 . 3 . 5 6 6

WAR/WAW hazards prevent instruction 5 from being dispatched

Out-of-order execution did not produce a significant improvement!
How many Instructions can be in the pipeline

Throughput is limited by number of instructions in flight, but which feature of an ISA limits the number of instructions in the pipeline?

*Number of Registers*

Out-of-order dispatch by itself does not provide a significant performance improvement!

How can we better understand the impact of number of registers on throughput?
Little’s Law

Throughput ($\bar{T}$) = Number in Flight ($\bar{N}$) / Latency ($\bar{L}$)

Example:

- 4 floating point registers
- 8 cycles per floating point operation

$\Rightarrow \frac{1}{2}$ issues per cycle!
Overcoming the Lack of Register Names

Floating Point pipelines often cannot be kept filled with small number of registers.

IBM 360 had only 4 Floating Point Registers

Can a microarchitecture use more registers than specified by the ISA without loss of ISA compatibility?

Yes, Robert Tomasulo of IBM suggested an ingenious solution in 1967 based on on-the-fly register renaming
### Instruction-level Parallelism via Renaming

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
<th>Source(s)</th>
<th>Destination(s)</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD F2</td>
<td>34(R2)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>LD F4</td>
<td>45(R3)</td>
<td></td>
<td>long</td>
</tr>
<tr>
<td>3</td>
<td>MULTD F6, F4</td>
<td>F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>SUBD F8, F2</td>
<td>F2</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>DIVD F4' F2</td>
<td>F8</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>ADDD F10, F6</td>
<td>F4'</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

In-order: 1 (2,1) . . . . . . 2 3 4 4 3 5 . . . 5 6 6
Out-of-order: 1 (2,1) 4 4 5 . . . (2,5) 3 . . 3 6 6

Renaming eliminates WAR and WAW hazards
*renaming ⇒ additional storage*
Handling register dependencies

- Decode does register renaming, providing a new spot for each register write
  - Renaming eliminates WAR and WAW hazards by allowing use of more storage space

- Renamed instructions added to an issue stage structure, called the reorder buffer (ROB). Any instruction in the ROB whose RAW hazards have been satisfied can be dispatched
  - Out-of-order or dataflow execution handles RAW hazards
Reorder Buffer
Smith and Pleszkun, 1985

Instruction slot is candidate for execution when:
- It holds a valid instruction ("use" bit is set)
- It has not already started execution ("exec" bit is clear)
- Both operands are available ("present" bits p1 and p2 are set)

Is it obvious where an architectural register value is?  No
Renaming & Out-of-order Issue

When are names (tags) in sources replaced by data?
Whenever an FU produces data

When can a name (tag) be reused?
Whenever an instruction completes

**Renaming table & reg file**

<table>
<thead>
<tr>
<th>p</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td></td>
</tr>
<tr>
<td>F5</td>
<td></td>
</tr>
<tr>
<td>F6</td>
<td></td>
</tr>
<tr>
<td>F7</td>
<td></td>
</tr>
<tr>
<td>F8</td>
<td></td>
</tr>
</tbody>
</table>

Holds data ($v_i$) or tag ($t_i$)

**Reorder buffer**

<table>
<thead>
<tr>
<th>Ins#</th>
<th>use</th>
<th>exec</th>
<th>op</th>
<th>p1</th>
<th>src1</th>
<th>p2</th>
<th>src2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$t_1$
$t_2$
$t_3$
$t_4$
$t_5$

September 28, 2022
### Renaming & Out-of-order Issue

**An example**

<table>
<thead>
<tr>
<th>Ins#</th>
<th>use</th>
<th>exec</th>
<th>op</th>
<th>p1</th>
<th>src1</th>
<th>p2</th>
<th>src2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>LD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>MUL</td>
<td>0</td>
<td>v2</td>
<td>1</td>
<td>v1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>SUB</td>
<td>1</td>
<td>v1</td>
<td>1</td>
<td>v1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>DIV</td>
<td>1</td>
<td>v1</td>
<td>0</td>
<td>t4</td>
</tr>
</tbody>
</table>

- **Insert instruction in ROB**
- **Issue instruction from ROB**
- **Complete instruction**
- **Empty ROB entry**
Simplifying Allocation/Deallocation

Instruction buffer is managed circularly
- Set “exec” bit when instruction begins execution
- When an instruction completes its “use” bit is marked free
- Increment ptr₂ only if the “use” bit is marked free
Renaming table & reg file

Reorder buffer

Replacing the tag by its value is an expensive operation

- Instruction template (i.e., tag t) is allocated by the Decode stage, which also stores the tag in the reg file
- When an instruction completes, its tag is deallocated
**IBM 360/91 Floating Point Unit**

*R. M. Tomasulo, 1967*

---

**load buffers (from memory)**

---

**store buffers (to memory)**

---

**distribute instruction templates by functional units**

---

**Common bus ensures that data is made available immediately to all the instructions waiting for it**

---

**Floating Point Reg**
Renaming and Out-of-order execution was first implemented in 1969 in IBM 360/91 but was effective only on a very small class of problems and thus did not show up in the subsequent models until mid-nineties. *Why?*

1. Did not address the memory latency problem which turned out to be a much bigger issue than FU latency
2. Made exceptions imprecise

*One more problem needed to be solved*

*Branch/jump penalties*

*More on this in the next lecture*
Reminder: Precise Exceptions

Exceptions are relatively unlikely events that need special processing, but where adding explicit control flow instructions is not desired, e.g., divide by 0, page fault.

Exceptions can be viewed as an implicit conditional subroutine call that is inserted between two instructions.

Therefore, it must appear as if the exception is taken between two instructions (say $I_i$ and $I_{i+1}$)

- the effect of all instructions up to and including $I_i$ is complete
- no effect of any instruction after $I_i$ has taken place

The handler either aborts the program or restarts it at $I_{i+1}$.
Effect on Exceptions
Out-of-order Completion

| I₁  | DIVD  | f6, f6, f4 |
| I₂  | LD    | f2, 45(r3) |
| I₃  | MULTD | f0, f2, f4 |
| I₄  | DIVD  | f8, f6, f2 |
| I₅  | SUBD  | f10, f0, f6 |
| I₆  | ADDD  | f6, f8, f2 |

**out-of-order comp**  1  2  2  3  1  4  3  5  5  4  6  6  

Consider exceptions on “DIVD”s

Precise exceptions are difficult to implement at high speed
- want to start execution of later instructions before exception checks finished on earlier instructions
Exceptions

- Exceptions create a dependence on the value of the next PC

- Options for handling this dependence:
  - Stall: No
  - Bypass: No
  - Find something else to do: No
  - Change the architecture: Sometimes: Alpha, Multiflow
  - Speculate!: Most common approach!

- How can we handle rollback on mis-speculation?
  
  Delay state update until commit on speculated instructions

- Note: earlier exceptions must override later ones
Hold exception flags in pipeline until commit point (M stage)

- If exception at commit:
  - update Cause/EPC registers
  - kill all stages
  - fetch at handler PC

Inject external interrupts at commit point
Phases of Instruction Execution

- **Fetch**: Instruction bits retrieved from cache.
- **Decode**: Instructions placed in appropriate issue (aka “dispatch”) stage buffer.
- **Execute**: Instructions and operands sent to execution units. When execution completes, all results and exception flags are available.
- **Commit**: Instruction irrevocably updates architectural state (aka “graduation” or “completion”).
In-Order Commit for Precise Exceptions

- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (⇒ out-of-order completion)
- Commit (write-back to architectural state, i.e., regfile & memory) is in-order

Temporary storage needed to hold results before commit (shadow registers and store buffers)
Extensions for Precise Exceptions

- add <pd, dest, data, cause> fields in the instruction template
- commit instructions to reg file and memory in program order ⇒ buffers can be maintained circularly
- on exception, clear reorder buffer by resetting \( \text{ptr}_1 = \text{ptr}_2 \) (stores must wait for commit before updating memory)

### Reorder buffer

<table>
<thead>
<tr>
<th>Inst#</th>
<th>use</th>
<th>exec</th>
<th>op</th>
<th>p1</th>
<th>src1</th>
<th>p2</th>
<th>src2</th>
<th>pd</th>
<th>dest</th>
<th>data</th>
<th>cause</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( \text{ptr}_2 \) next to commit
\( \text{ptr}_1 \) next available
Rollback and Renaming

Register file does not contain renaming tags any more. How does the decode stage find the tag of a source register? Search the “dest” field in the reorder buffer.
Renaming table is a cache to speed up register name lookup. It needs to be cleared after each exception taken. When else are valid bits cleared? Control transfers
Physical Register Files

- Reorder buffers are space inefficient – a data value may be stored in multiple places in the reorder buffer
- Idea: Keep all data values in a physical register file
  - Tag represents the name of the data value and name of the physical register that holds it
  - Reorder buffer contains only tags

Thus, 64-bit data values may be replaced by 8-bit tags for a 256-element physical register file

More on this in later lectures ...
Branch Penalty

How many instructions need to be killed on a misprediction?

Modern processors may have > 10 pipeline stages between nextPC calculation and branch resolution!

Next lecture: Branch prediction & Speculative execution