Overview of this Lecture

- Comparing the principles of data-flow and control-flow processors (or "von Neumann" processors after the work of von Neumann, Eckert and Mauchly)
- Problems with control-flow processors
- Data-flow implementation techniques:
  - static data-flow
  - coloured dynamic data-flow
  - tagged token dynamic data-flow
- Evaluation of data-flow
- Review and future directions

Comparing Control-flow & Data-flow

Example function: \( f(a,b,c) := a \cdot b + a\div c \)

Control-flow

```
load r0, (sp+a)
load r1, (sp+b)
mul r0, r1, r2
load r1, (sp+c)
div r0, r1, r0
add r2, r0, r0
rts
```

Data-flow

```
data path = arc
  matching or joining

return
```

Problems with Control-flow

- Typically optimised to execute sequential code from low latency memory:
  - concurrency simulated via interrupts and a software scheduler which:
    - has to throw the register file away and reload
    - disrupts caching and pipelining
  - jump/branch operations also disrupt the pipeline
  - load operations easily cause the processor to stall (cannot execute another thread whilst waiting).

Notes:
- multiple pipelines and an increasing disparity between processor and main memory speed only accentuate these problems
- perform badly under heavy load (esp. multithreaded environments)
- multiprocessor code is difficult to write

Implementation 1 — Static Data-flow

Source: J. Dennis et al. at MIT

Characteristics:
- at most one token on an arc
- backward signalling arcs for flow control
- tokens are just address, port and data triplets \((x, p, d)\)

Example instruction format:

```
<table>
<thead>
<tr>
<th>address (e.g.)</th>
<th>op-code</th>
<th>operands</th>
<th>dests.</th>
<th>dests. clear</th>
<th>sigs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x30</td>
<td>mul</td>
<td>a, b, c</td>
<td>a, b</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>0x31</td>
<td>add</td>
<td>a, b</td>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>0x32</td>
<td>div</td>
<td>a, b, c</td>
<td>a, b</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>0x33</td>
<td>ret</td>
<td>a, b</td>
<td>undef</td>
<td>undef</td>
<td></td>
</tr>
</tbody>
</table>
```

Notes:
- instruction ordering in the memory is unimportant
- \( a = \) space for operand to be stored
- \( b = \) space for destination clear to be stored (initially clear)
- \( i \) and \( r \) indicate left or right port
- \((a), (b)\) and \((c)\) are difficult to determine — dependent on calling code
- functions are difficult to implement because:
  - mutual exclusion required on writing to function input arcs
  - backward signal arcs have to be determined
- solution: code copying (horrible!)

Implementation 2 — Coloured Data-flow

Example machine: Manchester data-flow prototype

Characteristics:
- many tokens on an arc and no backward signal arcs for flow control
- tokens have a unique identifier, a colour, which identifies related data items
- matching tokens for dyadic operations by matching colours
- thus, function calls by each caller using a unique colour

Instruction format: similar to static data-flow but no backward signals and operand storage is more complex.

Problems:
- matching colours is expensive
- implemented using hashing with associated overflow
- difficult to pipeline
- garbage collecting unmatched tokens is expensive
- uncontrolled fan-out can cause a token explosion problem

Implementation 3 — Tagged-token Data-flow

Example machines: Monsoon machine (MIT) and EM4 (Japan)

Characteristics:
- dynamic data-flow, so many tokens per arc
- separates the token storage from the program into activation frames (similar to stack frames for a concurrent control-flow program)
- function calls generate a new activation frame for code to work in
- tokens have an associated activation frame instead of a colour
- activation frames are stored in a linear memory with an empty/full flag for every datum, \((type, value, port, presence)\)