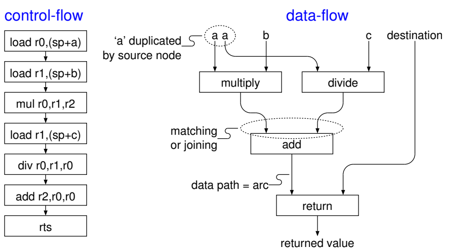
Computer Design — Lecture 16 1

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 2

Example function: f(a,b,c) := a.b + a/c



Problems with Control-flow 3

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 4

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 $\langle a, p, d \rangle$

example instruction format:

op-code	op1	(op2)	dst1 + dc1	(dst2 + dc2)	sig1	(sig2)

where **( )** indicates optional parameters
op-code is the instruction identifier
op1 and **op2** are the space for operands to wait (**op2** missing for monadic operations)
dst1 and **dst2** are the destinations (**dst2** being optional)
dc1 and **dc2** are destination clear flags (initially clear)
sig1 and **sig2** are the signal destinations (handshaking arcs)

Example Static Data-flow Program 5

address (e.g.)	**instruction** op-code	operands	dests.	dests. clear	sigs.
0x30	`mul`	□ ,□ ,	0x31ℓ,nil,	◇ ,◇ ,	(a)ℓ,(b)ℓ
0x31	`add`	□ ,□ ,	0x33ℓ,nil,	◇ ,◇ ,	0x30ℓ,0x32ℓ
0x32	`div`	□ ,□ ,	0x31r,nil,	◇ ,◇ ,	(a)r,(c)ℓ
0x33	`ret`	□ ,□ ,	undef,undef,	◇ ,◇ ,	0x31ℓ,(dest)ℓ

notes:

- instruction ordering in the memory is unimportant
- □ = space for *operand* to be stored
- ◇ = space for *destination clear* to be stored (initially clear)
- ℓ 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 6

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 7

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, $\langle type, value, port, presence \rangle$