On Parallelism: Spirit of Place

Andrew Reilly, Dolby Australia

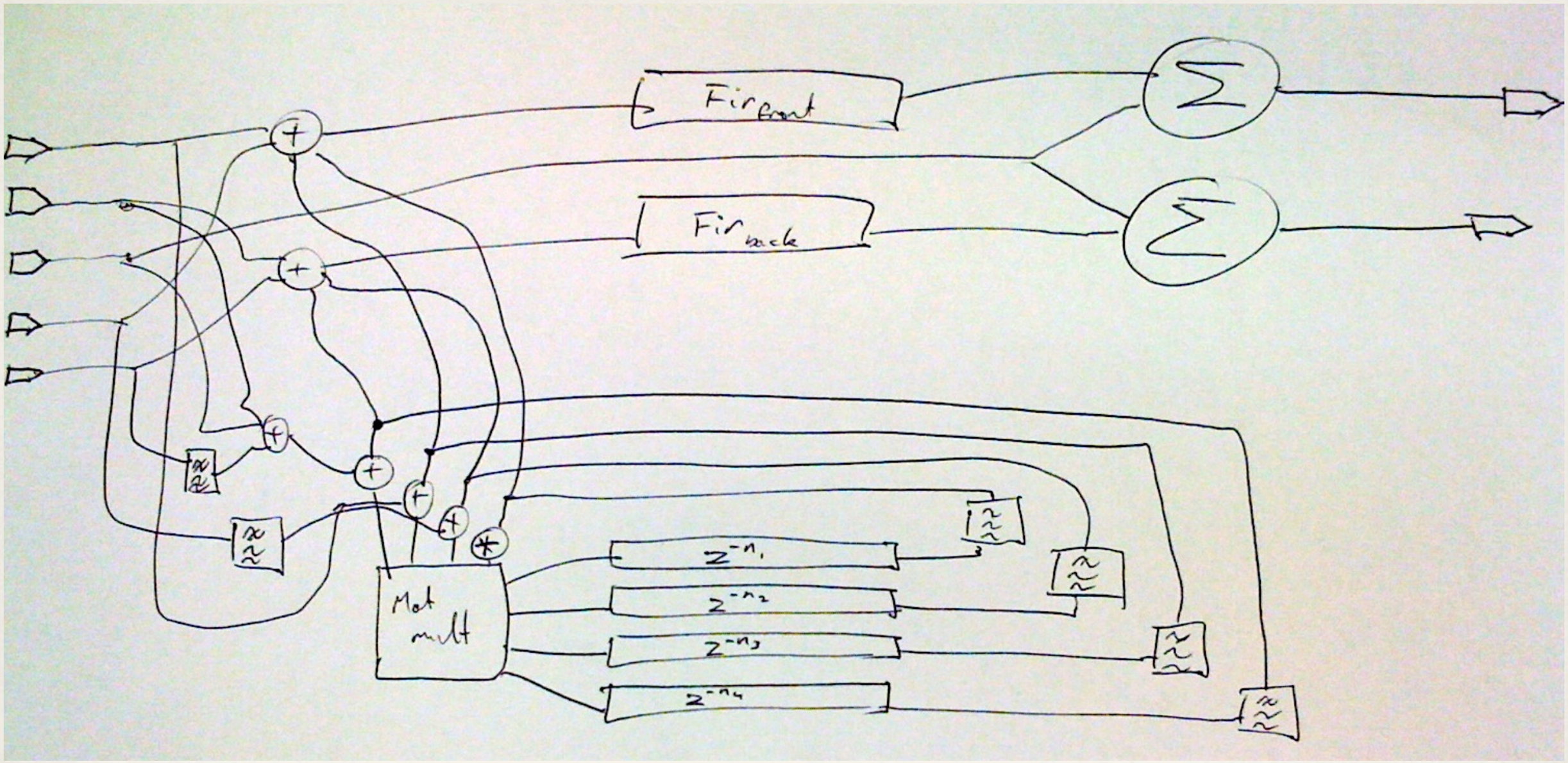
Sydney: 8 March 2013

On Parallelism: Overview

"To define it rudely but not inaptly, engineering is the art of doing that well with one dollar which any bungler can do with two after a fashion."

Arthur M. Wellington

- Problem - efficient audio processing on contemporary processors
- Our training and experience - programs and algorithms - sequence
- Contemporary hardware - opportunities for parallelism everywhere
- Effect on code - different trade-offs and costs



Audio Algorithms and Code

Nodes are functions (perhaps with state), directed edges are signals - inputs and outputs

Sydney: 8 March 2013

Input

Mix

Filter

Delays

Filter

Mix

Output

Input

Delays

Filter

Filter

Mix

Mix

Output

Parallelism at cost of latency

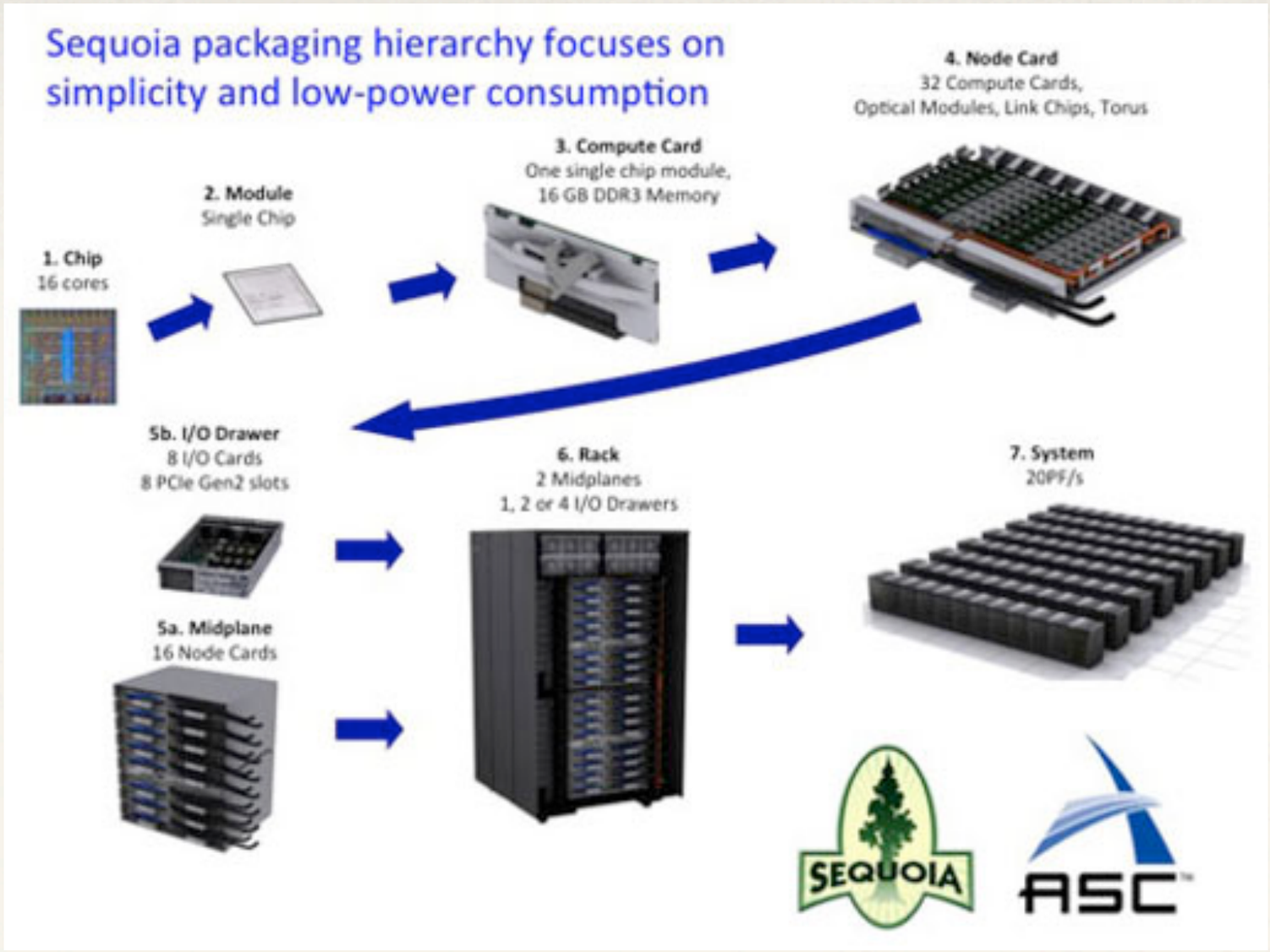
- Previous net-list also has much sequential dependency

- Can execute delay-line **outputs** before inputs - delay breaks dep's

- Can break dependencies by **introducing** delays between nodes

- Problem of load-balancing arises

- Problem of work distribution arises:
  easy to have much more algorithmic parallelism than processors.



Top Super - 1.5m cores 20PF/s

Each chip has 16 cores, each with cache and SIMD - much like phones and tablets

Sydney: 8 March 2013

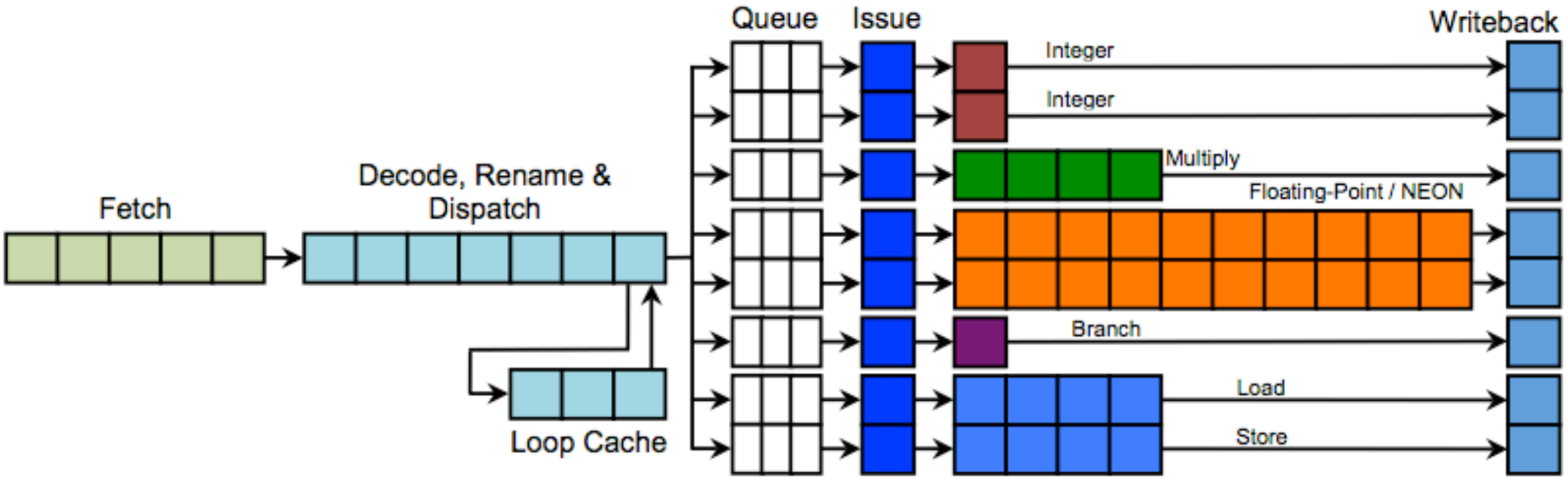


Figure 2 Cortex-A15 Pipeline

One modern ARM core

Note the instruction level (ILP), pipeline level and data-parallel (SIMD) opportunities

Sydney: 8 March 2013

Scales of parallelism

- Pipelining: parallelism from added latency (vs seq. dep.)
- Instruction level parallelism - compilers, sequencers can look after (OoO, VLIW)
- SIMD - the new reality - they're everywhere - four of everything
- Threads and cores - do we need to go there?
- Parallelism has costs as well as benefits - constrain applicability

Pipelines: parallelism from latency

- Cortex-M0, SHARC has three-stage pipe: fetch, execute, writeback (?)

- Cortex-A15 has 13-23 stages, x86 Core series 16-19, TI C66x: 11
  allows higher clock frequencies

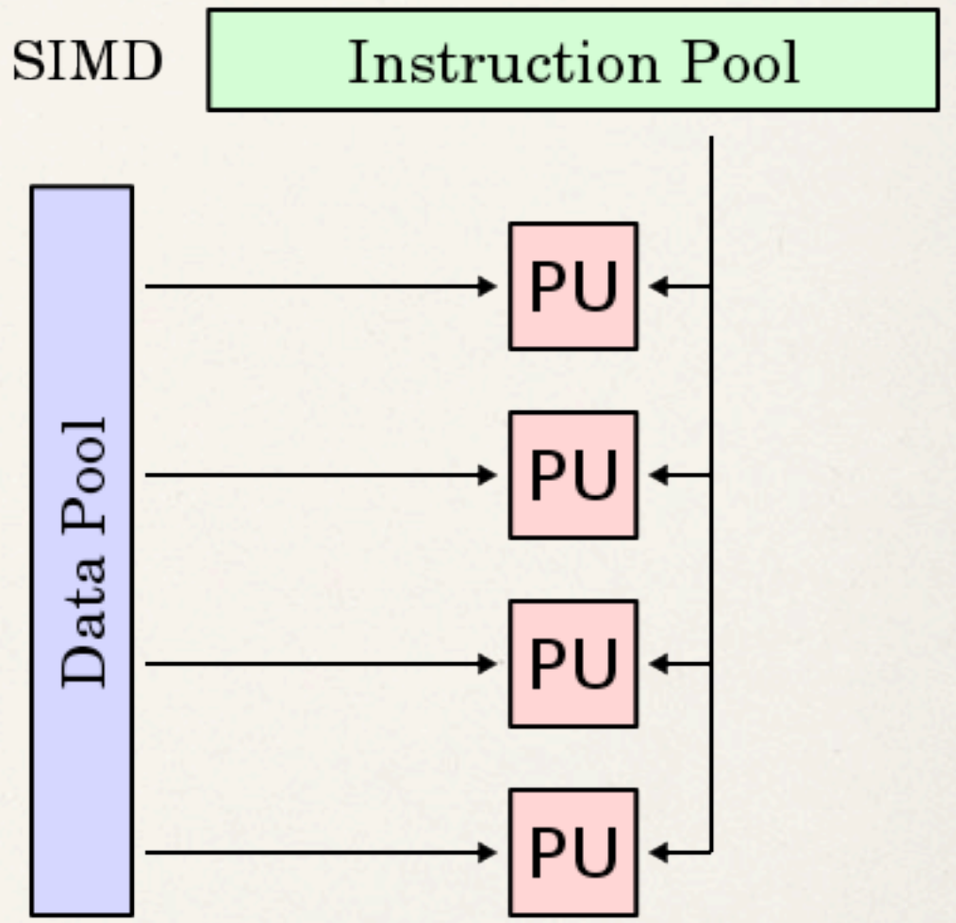
- Not all that latency is visible:
  branch prediction avoids most branch bubbles
  bypass networks allow dependent operations access before writeback
  integer ALU typically zero or one cycle effective latency
  memory reads from L1 cache usually about four cycles
  multiplies or floating point, usually two to four cycles.

Instruction-level parallelism

- make use of common fine-grain independence from
  - software pipelining or out-of-order hardware pipelines+renaming
  - loads+stores, address computations, branches and data processing
  - sequentially dependent within one iteration, but often independent across iterations.

SIMD: data parallel in one instr.

- Instructions that operate on multiple (fixed number) of data in parallel
- register files that store short (fixed length) vectors
- element-by-element operations between vector registers
- otherwise just like other instructions in pipeline



Threads and cores

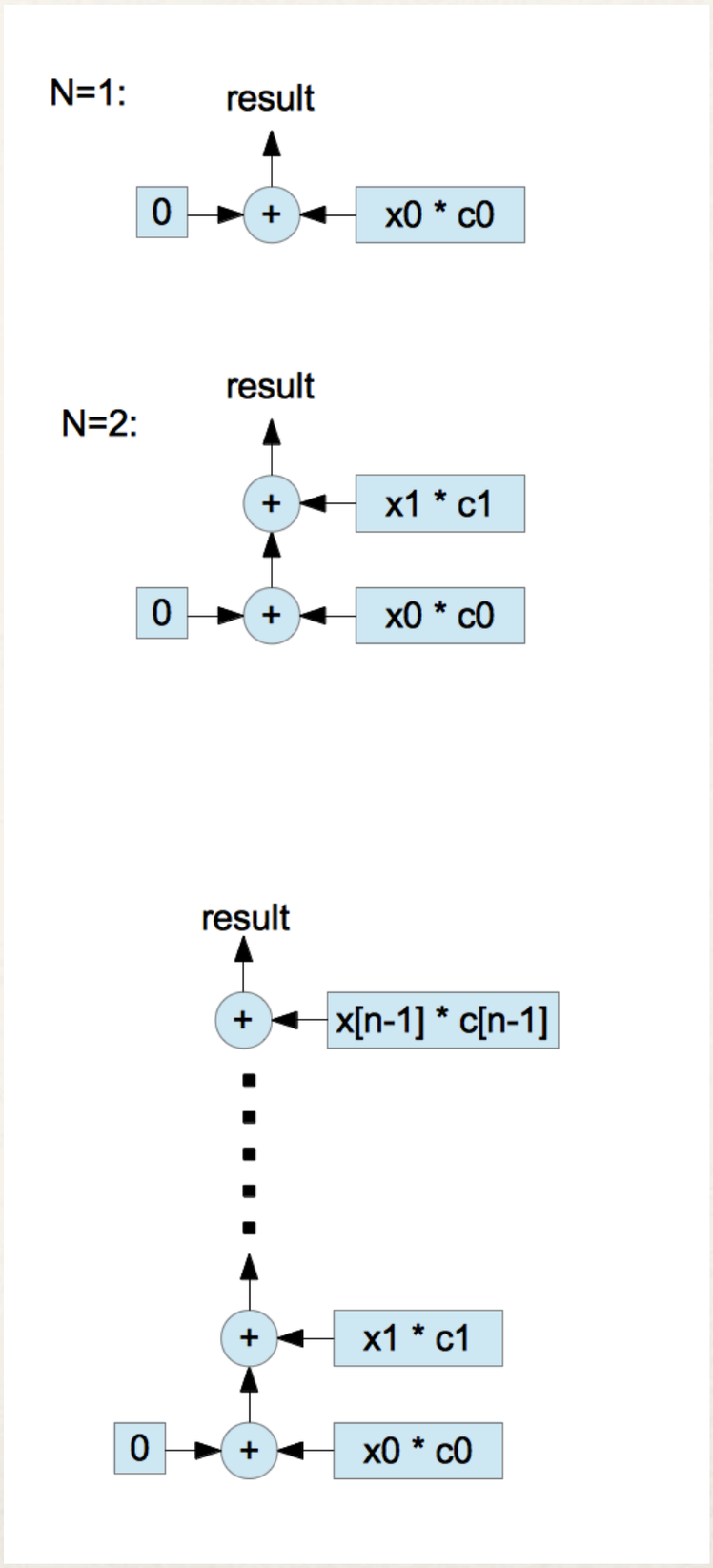
- multiple instruction pointers (and call stacks)
- issues of communication and coordination
- issues of load balancing
- issues of work distribution
- popular OSes are good at undifferentiated throughput
  not so much for meeting deadlines (real-time)
- many extra concerns for programmer (races/deadlocks/consistency)
  leads to bugs

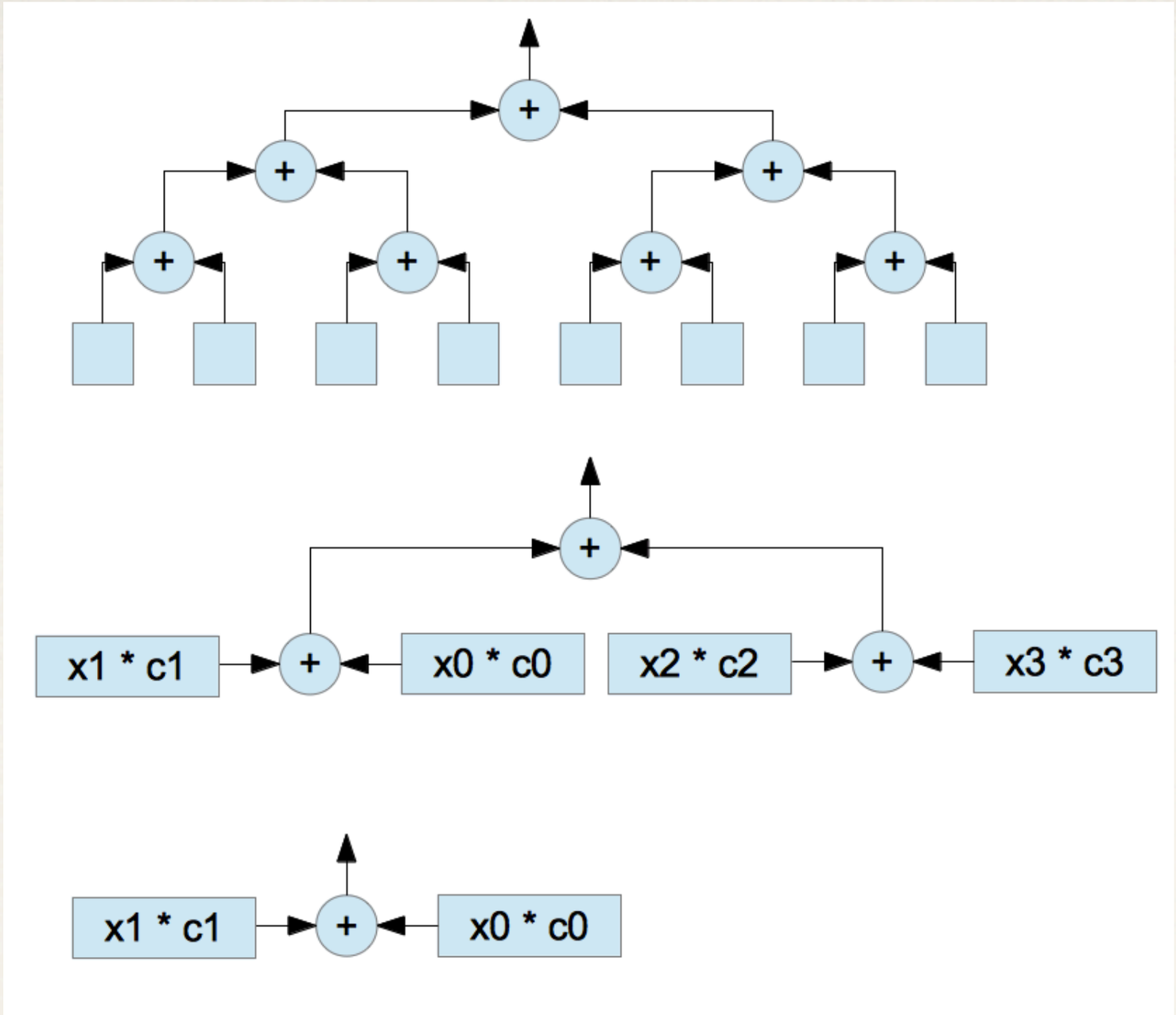
An example: dot product

- for (i = 0, a = 0; i < N; i++) {
  a += x[i] * c[i]; }

- for (i = 0, a = 0; i < N; i++) {
  a += *x++ * *c++; }

- each iteration has:
  add, multiply
  two loads
  two address update/compute
  test and branch

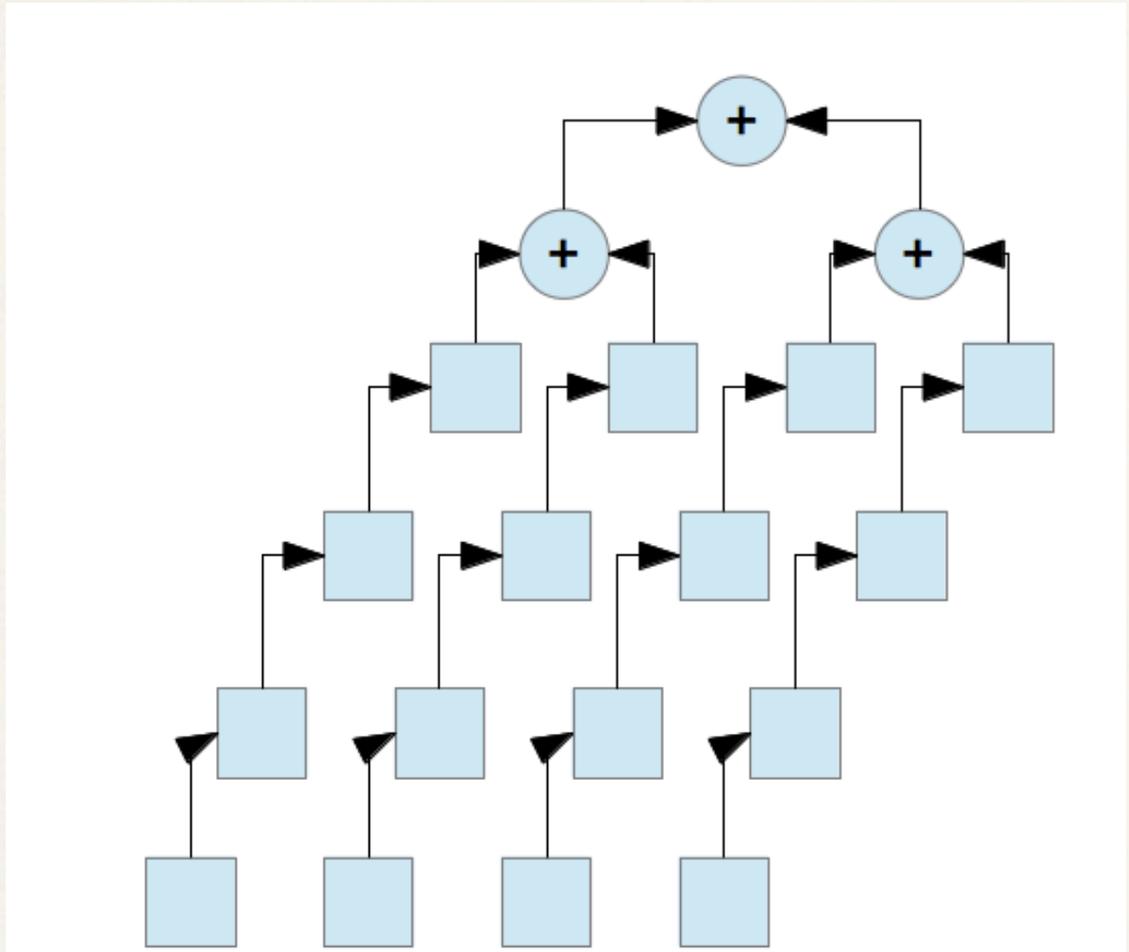




Avoiding “accumulator” pattern

Now we have more independent operations than we can use!

Sydney: 8 March 2013



Mapping to available hardware

Can use four-way SIMD to accumulate, with final sum-tree.

Sydney: 8 March 2013

In code

- for (i = 0, a = 0; i < N; i++) {
  a += x[i] * c[i]; }

- for (i = 0, av = {0}; i < N; i+=4) {
  av = vaddq_f32(a, vmpyq_f32(vld1q_f32(x+i), vld1q_f32(c+i))); }
  a = (av[0]+av[1]) + (av[2] + av[3]);

- an auto-vectorising compiler will do this for you **if** it can prove that x[] and c[] do not overlap, and are properly aligned and that N is a multiple of 4. Maybe.

- This code still stalls because of pipeline latencies.

Dealing with pipelines: unrolling

- Say add/multiply latency is two cycles: unroll by two:

- ```
  for (i = 0, a = 0, b = 0; i < N; i+=2) {
    a += x[i] * c[i];
    b += x[i+1] * c[i+1]; }
  a = a + b;
  ```

- ```
  for (i = 0, av = {0}, bv = {0}; i < N; i+=8) {
    av = vaddq_f32(av, vmpyq_f32(vld1q_f32(x+i), vld1q_f32(c+i)));
    bv = vaddq_f32(bv, vmpyq_f32(vld1q_f32(x+i+4), vld1q_f32(c+i+4))); }
  av = av + bv;
  a = (av[0]+av[1]) + (av[2] + av[3]);
  ```

- Unfortunately float addition is usually four cycles...

Extra twist: Hyperthreading

- Intel, MIPS, Qualcomm QDSP, others
- As we've seen, can take 16 independent ops to keep a modern pipeline full: not easy to arrange. (requires lots of registers!)
- Try: mash multiple threads together into one pipeline
- Need extra state (registers, thread tracking IDs)
- Threads compete for cache space and data fetch bandwidth
- Qualcomm went from 6 to 3 threads, guess: because lots of threads was harder to code than densely-pipeline/SIMD optimized.

Benefits of parallelism

- Fewer cycles to do job
- go to sleep faster - saves power
- more of the chip doing useful work: more efficient
- SIMD/OpenCL: more work per control overhead: more efficient

Costs of parallelism

- latency (usually)
- space (can't automatically re-use storage buffers)
- overhead (communication, synchronization)
- difficulty --> bugs
- difficulty --> harder to prove correctness
- space: more silicon, more expensive parts

Take-away ideas

- Be aware of necessary sequential dependencies (eg polylog)

- Be aware of opportunities to express parallelism - vector of polylogs allows sequential steps of log to be interleaved with independent iterations.

- Push loops into leaf functions - compilers often can’t see through calls

- Look at parallelism at different scales - vector, pipeline, process