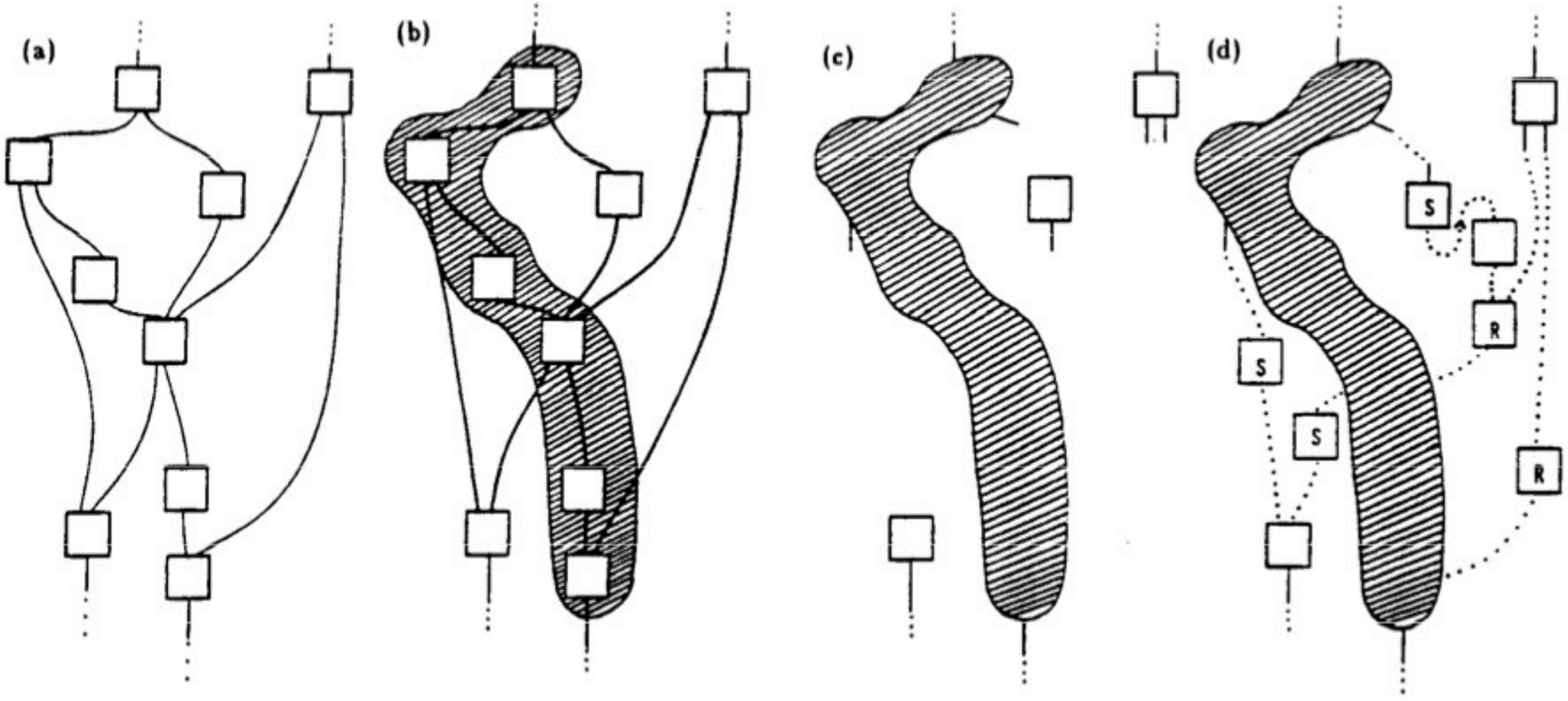
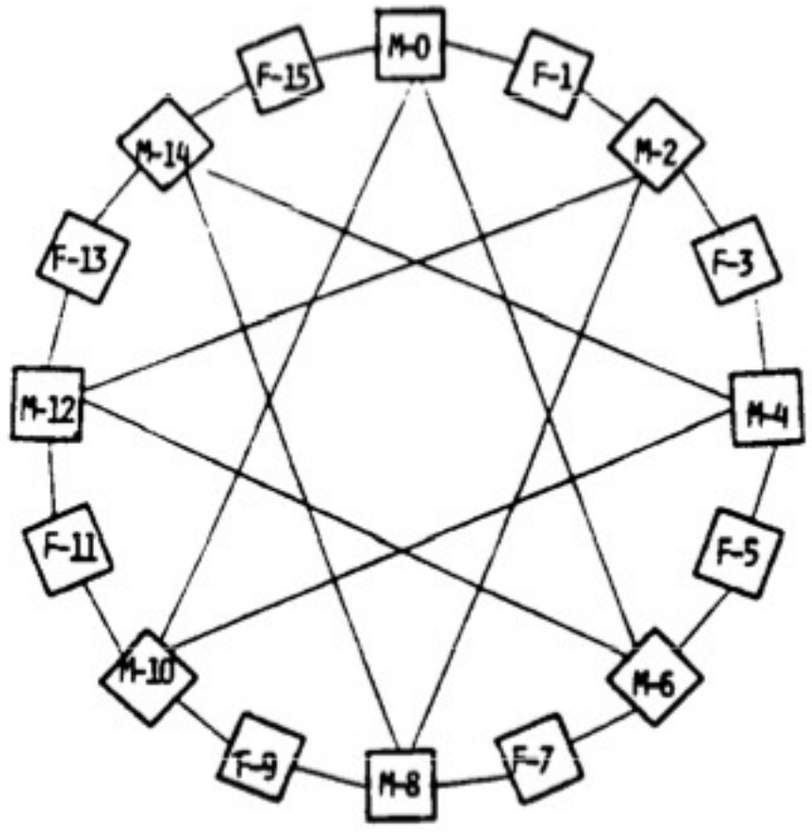
548

Lecture 6


(a)
(b)
(c)
(d)
S
S
S
R
R


M-0
F-1
M-2
F-3
M-4
F-5
M-6
F-7
M-8
F-9
M-10
F-11
M-12
F-13
M-14
F-15

Characteristics of VLIW

- executes a large number of simple instructions at once
- each operation takes a statically predictable amount of time

Why VLIW?

- Lower power per computation
- More general than a vector machine
- Compiler can express explicit parallelism
- Compiler can view more than an OOO inflight instruction limit.
- Easier to extract large amounts of ILP compared to OOO.

Why not VLIW?

- If life doesn't go as you statically predicted it, you are in trouble
  - (modify bundles at runtime maybe?)
- (limited) backward compatibility
- Not good at general purpose code
  - relies too much on datalevel parallelism

Branching

- Multiple tests per instruction bundle
- Branches have a precedence to them
  - add; add; mul; beq; blt; branch
- Compute branch predicate early
- Compiler inserts fixedup code on early trace exits

Memory

- Alias problem "memory disambiguation"
- Each unit had a local bank + slower general-purpose any-bank port

Your questions

- Are GPUs the primary use for VILW today? Or is it also used in specialized architectures for scientific computing?
- Is Itanium surviving? Are the speedups too difficult to get? (need to have good alias and memory bank prediction?)
- There are OoO engine, vector, VLIW, vector-threaded. Today we put vector units on normal cores. Is there still execution model datapoints to explore between VLIW (static ILP) and OoO (dynamic ILP)?
- How does VLIW cores look in a multi-core environment? (memory ordering issues?)
- The VLIW inst is up to 5 alu and 1 FC. Is this ILP easy to extract from small pieces of code for this domain?
- How long does the compiler take to generate code for this machine?
- How good was the memory and loop prediction?
- How much can modern hardware techniques reduce the complexity required of the compiler?
- Are any of the trace scheduling ideas used by modern (general use) compilers?
- How well did the ELI actually perform?
- How much speed up does VLIW actually by over increased pipelining and fine-grained control?
- Why not implement as a vector-based processor?
  Why is register size greater than size of L1 and L2 caches?
- Does each SIMD have a branch predictor? If not, where is branch prediction done?
- Is VLIW in some form then here to stay?
- Fisher (1983) talks about adding special edges to the dependency graph to prevent live variables from being clobbered but it is unclear to me why this works.
- Another idea would seem to be to convert instructions to VLIW microcode at runtime. Is this what is meant by horizontal microcode?
- A question which is raised in Fisher (1998): Is out of order execution really worth it or is it better to let the compiler perform the parallelization (VLIW/trace scheduling)?
- Why not extend trace scheduling to adapt dynamically as it runs based on what branches are taken?
- It seems like a fundamental limitation of trace scheduling is that beyond a certain number of blocks any trace will be unlikely. Are there any good ways around this?
- With (n+1)-way jumps based on common suffix, doesn't that mean you'll have lots of instruction cache misses? (Can't place both branches relatively close to test.) Does that matter? Is it made up for by more ILP?

Your questions

- Fisher's retrospective asked the question: Does region scheduling really work well enough to justify it? He answered sometimes, but when exactly is it worth it?
- What is a vector machine?
- How does the jumping mechanism perform n+1 tests in a single cycle?
- Instruction Cache and Constant Cache has a unlimited program size. How does this work? Do we really need it?
- How realistic is it to pack a full workload onto a VLIW processor?
- How long are instructions to the individual spus?
- 5 AU instructions doesn't seem particularly 'VLIW'. What prevents the use of larger groupings of units in the SPU and longer instruction words?
- Why do GPUs require custom programming patterns (like operation kernels), rather than transforming code in the compiler like the trace-scheduling bulldog design?
- How much larger are programs compiled for VLIW? (Looks like the compiler may need to include parts of the program multiple times, in different contexts.)
- Is there an efficient way to cache instructions with many possible jump locations per instruction?
- Does the memory bank prediction scheme work well in pra
- Predication seems to be very effective in eliminating the branches. Do they consider predication in VLIWs?
- Trace scheduling seems like a good approach for JITing. Do JIT compilers suse this technique?
- How do they balance the execution latencies of parallel yet different operations like add and mult
- Amdahl's law considering the memory operations will back lash, doesn't it?
- Huge register file that could be a bottleneck.

- What is the fundamental difference between stream processors and VLIWs?
- Is SMT being used mostly for hiding memory latency or lack of parallelism is a major factor?
- How restrictive is the programming model for GPUs?
- Does this depend on dynamic compiler analysis (based on run-time traces) as opposed to static analysis?
    - If so, how well can we do with just static analysis
- At what branch prediction accuracy and recovery overhead does VLIW not benefit us
    - What are we at right now?