

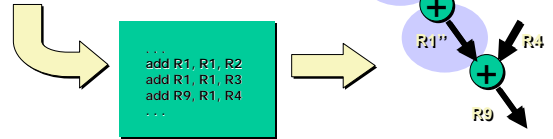
Compilers vs. Architecture

Instruction Scheduling

Four way addition example

- In a dyadic instruction set, performing an operation on more than two sources requires temporary values.

```
int sum( int a, int b, int c, int d)
{
    return a + b + c + d;
}
```



Instruction scheduling

- compiler -> instruction list
- machine logic ->
 - fetch, decode, read args, execute, write result...
 - but not necessarily single instruction at a time (ILP)
- degree of parallelism depends on
 - data-dependencies
 - resource constraints

Compiler approach

- consider resource requirements for each instruction, and make sure another instruction is not scheduled while resources are unavailable
- consider data-dependencies, and make sure instructions are not scheduled before data becomes available
- easier said than done...

Scheduling without resource bounds

- we look at loops here
 - find groups of instructions which can be coscheduled, and rewrite loop so that these are body of loop...
 - this is important...

Example

```
for i <- 1 to N
  a <- j ⊕ V[i-1]
  b <- a ⊕ f
  c <- e ⊕ j
  d <- f ⊕ c
  e <- b ⊕ d
  f <- U[i]
g: V[i] <- b
h: W[i] <- d
  j <- X[i]
```

- we want to find data dependencies here...

- need to find largest pattern to represent loop
- loop length depends on longest data dependency graph
- may include instructions executing different iterations in 'higher-level' loop
 - obviously, will need move instr, i++, loop bounds check, etc...
 - can include cycle count in model

Resource-bounded planning

- pattern, if existing depends on
 - data-dependency graph
 - number of instruction requesting certain resource
 - number of 'copies' we have of that resource
 - cyclecount (time spent at resource)
- any algorithm should
 - consider instruction priorities
 - keep track of resource assignment
 - iterate through list of possible instructions by moving them back and forth schedule and option list.

Conditionals

- For balanced short branches – can compute both and have a conditional move

```
for i <- 1 to N
  x <- M[i]          x <- M[i]
  if x > 0           u' <- A[i]
    u <- A[i]       u <- b
  else               if x > 0
    u <- b           u <- u'
```

- waste of cycle for long, unbalanced branches
- not an option for instructions with side effects
- compiler can use trace (profile) info for scheduling

- Compiler scheduling:

- static, but have info for longest data dependency chains, cannot predict/model everything a priori

- Hardware out-of-order exec:

- dynamic, uses `local` info, but handles odd cases...

Branch prediction

- Important: if wrong branch pipeline has to be emptied
- Compiler prediction:
 - can use heuristics (~20%), or profiles (~9%)
- Hardware approach:
 - learn from experience
 - results in ~10% miss