Wavescalar Assembly: Dataflow

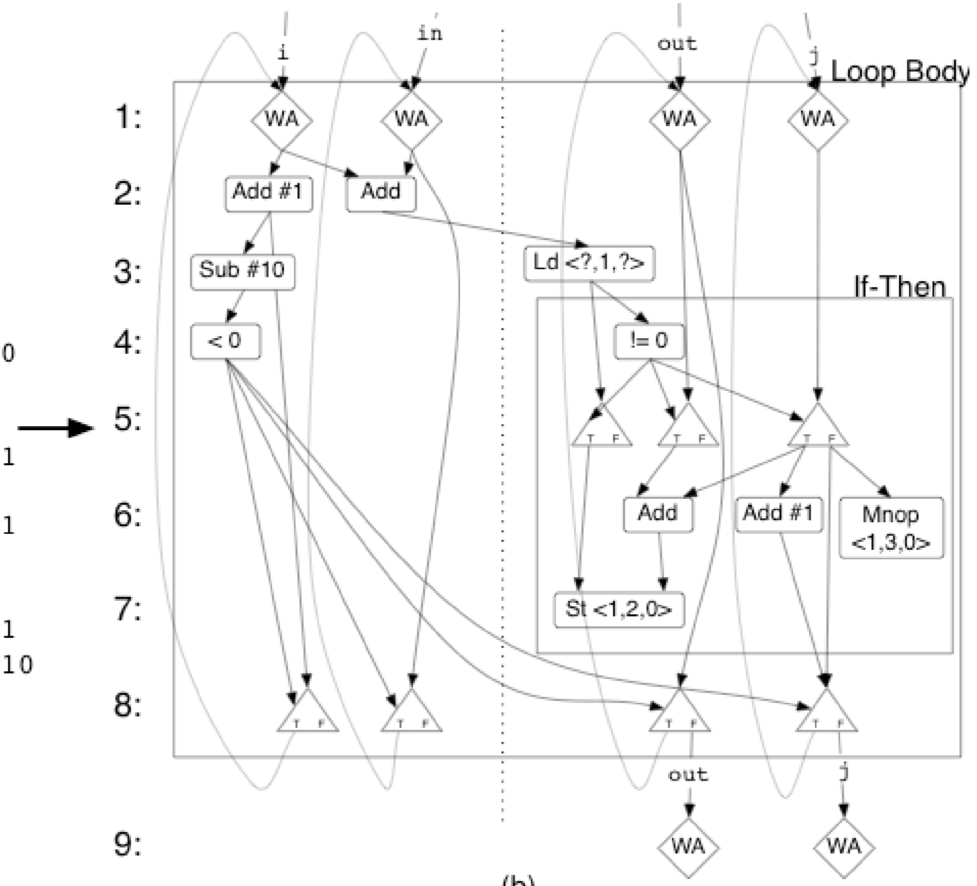
```
function s(char in[10], char out[10]) {
     i = 0;
     j = 0;
     do {
          int t = in[i];
          if (t) {
               out[j] = t;
               j++;
          }
          i++;
     } while (i < 10);
     // no more uses of i
     // no more uses of in
}
```

```
;; r0 = i
;; r1 = j
;; r2 = in
;; r3 = out
;; r4 = t

loop:
add  r6, r2, r0
ld   r4, r6(0)
bne  r4, L1
add  r6, r3, r1
st   r4, r6(0)
addi r1, r1, #1

L1:
addi r0, r0, #1
subi r7, r0, #10
blt  r7, loop
```

(a)



(b)

Wavescalar Assembly: Format

- Wavescalar is an extension of the Alpha ISA
  - RISC (more or less)
  - “Register to register” becomes “PE to PE”
  - Tagged-tokens

- Instructions have a basic format

  operand {outputs}, {inputA}, {inputB}, {inputC}
  - Each port may hold a list of inputs or outputs
  - Some instructions have less inputs
  - The curly braces are optional

Referring to Arcs

- Named arcs
  - You have infinite “registers”

```
ldq a, addr, 0
ldq b, addr, 8
addq c, a, b
```

- Use labels
  - The linker resolves symbols (if possible)

```
L0:
        ldq { }, addr, 0
        ldq ^L1:2, addr, 8
L1:
        addq c, ^L0:0, { }
```

Wavescalar Assembly: Instructions

Alpha-based

- Computation
- Memory
  - Ordered interface
  - Unordered

Wavescalar Specific

- Control
  - Branches/Joins
- Tag management
  - Wavescalar is dynamic dataflow
- Synchronization

For a list of all instructions and formats, run:
lc-devel/src/drip/printInsts

Alpha-based Instructions

http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15740-f98/public/doc/alpha-guide.pdf

- Arithmetic
  - add, sub, mul, div, …
  - Long word (32 bit) arithmetic

    addl {outputs}, {inputAs}, {inputBs}
  - Quad word (64 bit) arithmetic

    addq {outputs}, {inputAs}, {inputBs}
- Comparison
  - cmple, cmpeq, …
- Logical
  - and, bis, xor, …

Using Immediates

- Almost all instructions have immediate forms
  - AddI, sll_I, s4subq_I, …

  Addi {outputs}, {inputs}, immediate

- Otherwise, create a constant and send it
  - cnst creates an immediate when a trigger is received

  cnst {outputs}, {triggers}, immediate

Accessing Memory

Ordered

- ldq, stq, mnop, …
- The system manages dependences
  - Store buffer
- Memory operations are tagged
  - Wave-ordered memory

Unordered

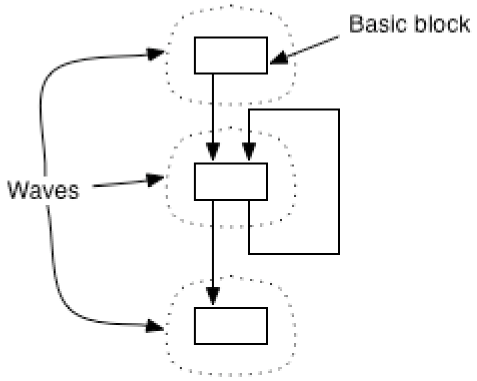
- ldq_U, stq_U
- The programmer manages dependences
  - Dataflow firing rule
- Stores have an output arc
  - Reports when store completes

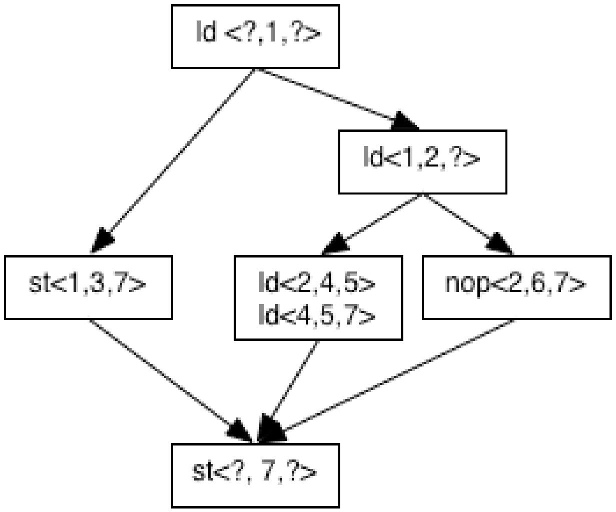
Wave-Ordered Memory

- Programs are partitioned into DAGs (“waves”)
- Memory operations are given “sequence numbers”
  - <previous, current, next>.ripple

ld {outputs}, {address}, immediate <p, c, n>.r

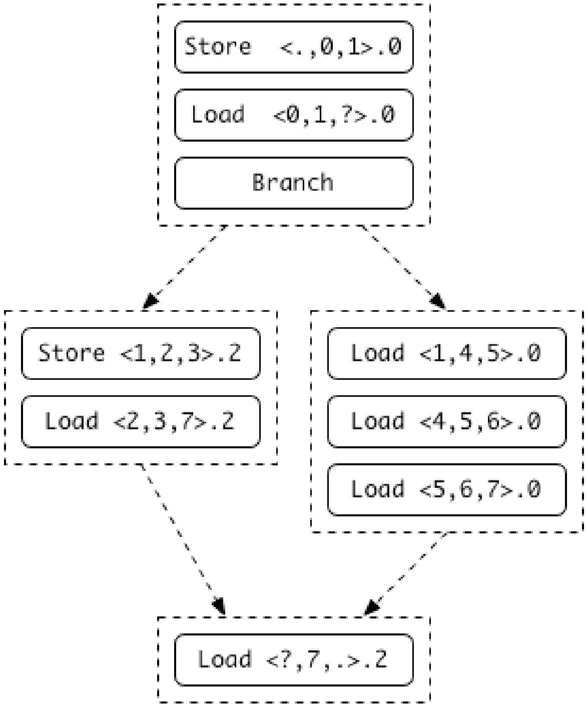
- No-ops may be required to totally order operations





Ripples

- A sequence of loads need not be ordered
  - The hazards are RAW, WAR, WAW
- Fully ordering loads decreases parallelism
- Add a “ripple number”
  - The previous store’s sequence number



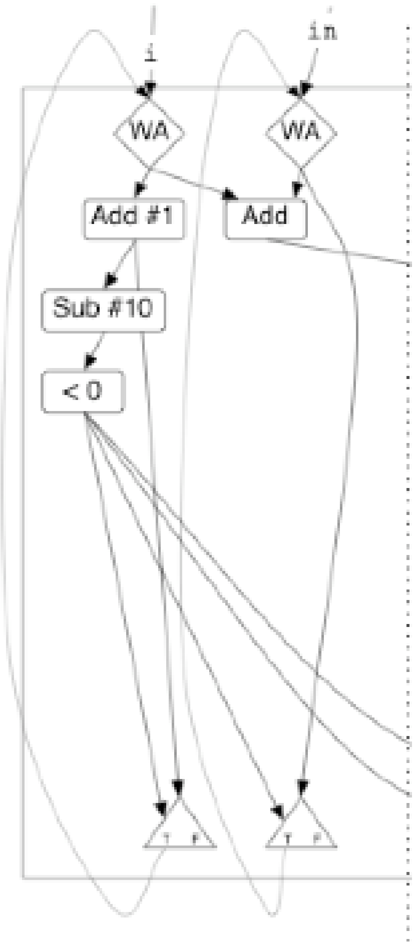
Tagged Tokens

- Wavescalar is a tagged-token architecture
  - Each token has two components
    - A value
    - A tag
  - Each tag has two components
    - A thread number
    - A wave number
- Tags allow re-entrant code
  - The dataflow firing rule is modified

  An instruction executes when all of its operands *for a given thread and wave* have arrived.

Re-entering a Wave

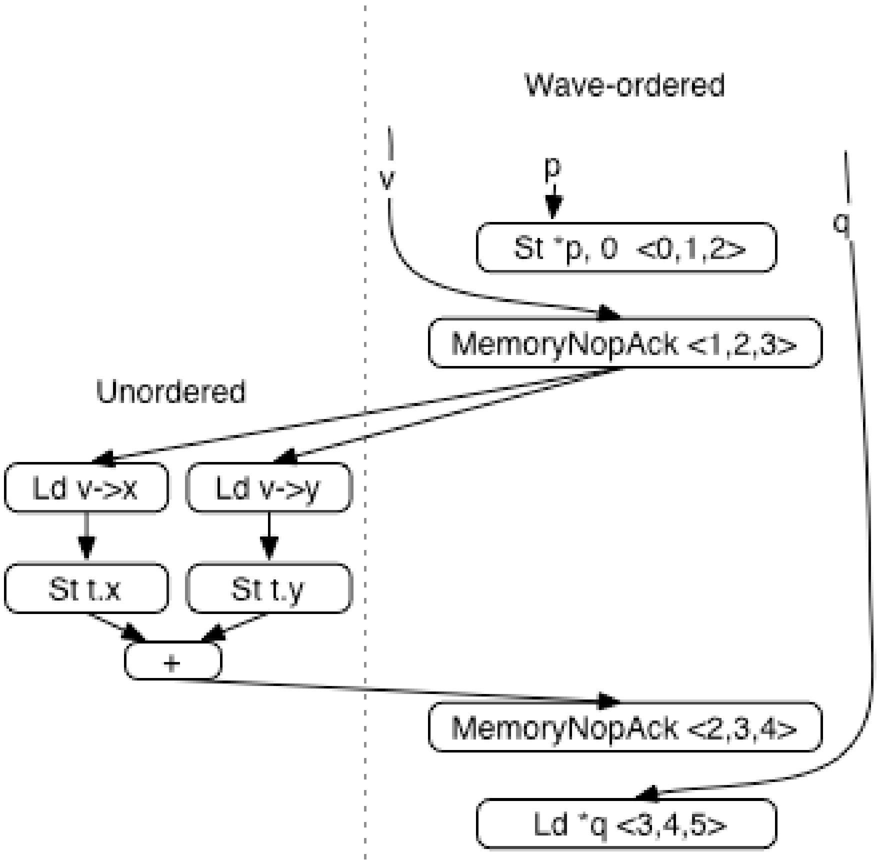
- Each dynamic wave is assigned a wave number
- Tokens entering a wave are tagged with that wave number
  - Wave advance (wa)
    - Increments the wave number on a token
  - Canonical wave advance (cwa)
    - Increments the wave number
    - Creates a new memory ordering for that wave
- Multiple memory orderings can exist…but talk to us first



Ordered and Unordered

```
struct {
    int x,y;
} point;

foo(point *v, int *p, int *q)
{
    point t;
    *p = 0;
    t.x = v->x;
    t.y = v->y;
    return *q;
}
```

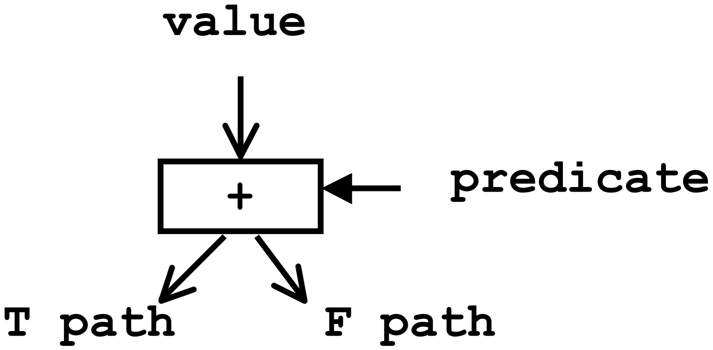


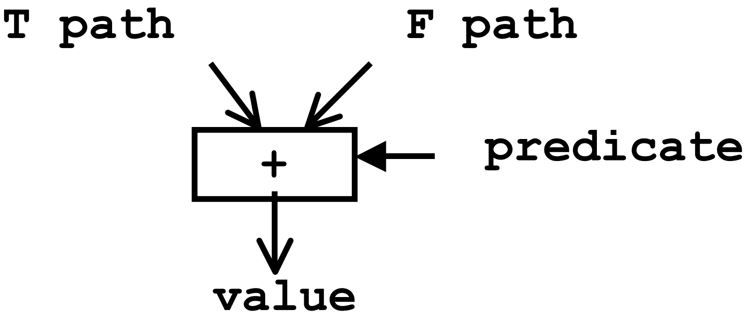
Control: Token Steering

- No branch instructions
- Two control instructions
  - rho (split): conditional

    rho {T-output}, {F-output}, {value}, {predicate}
  - phi (join): speculative

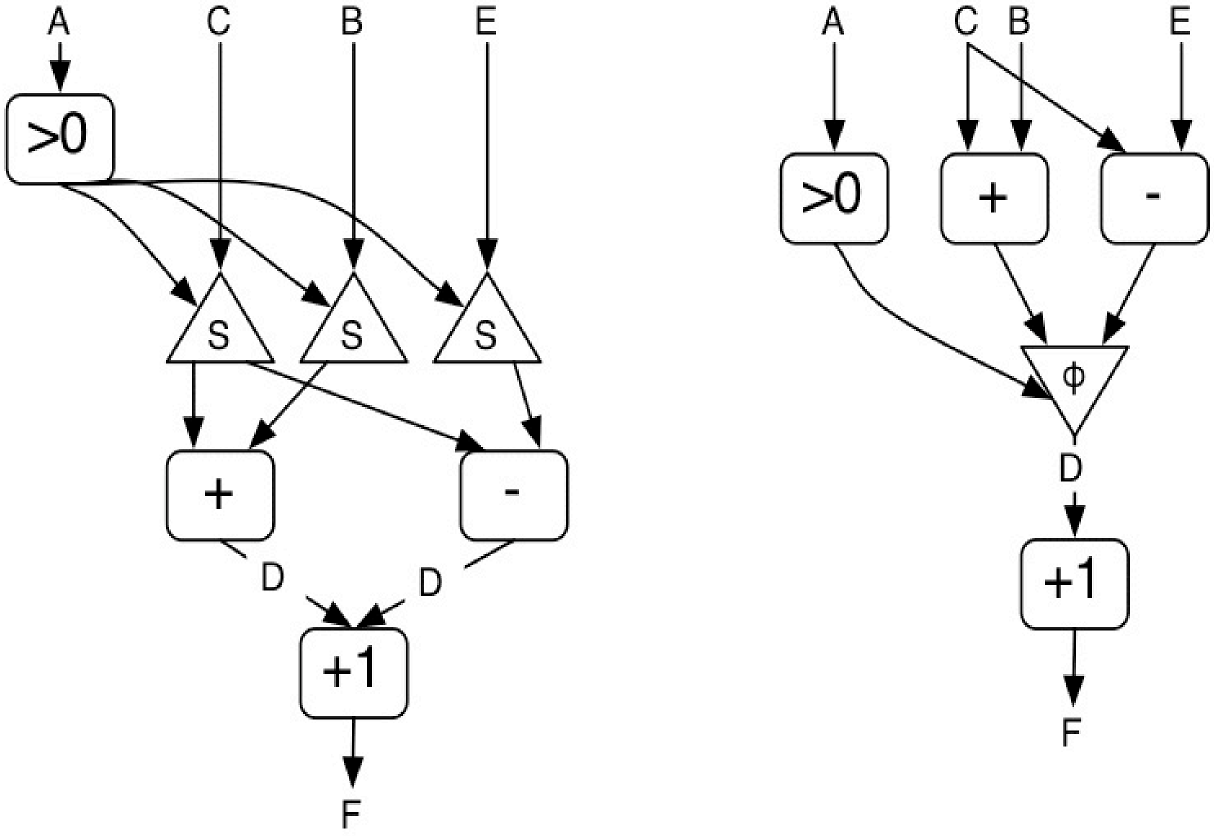
    phi {output}, {T-value}, {F-value}, {predicate}





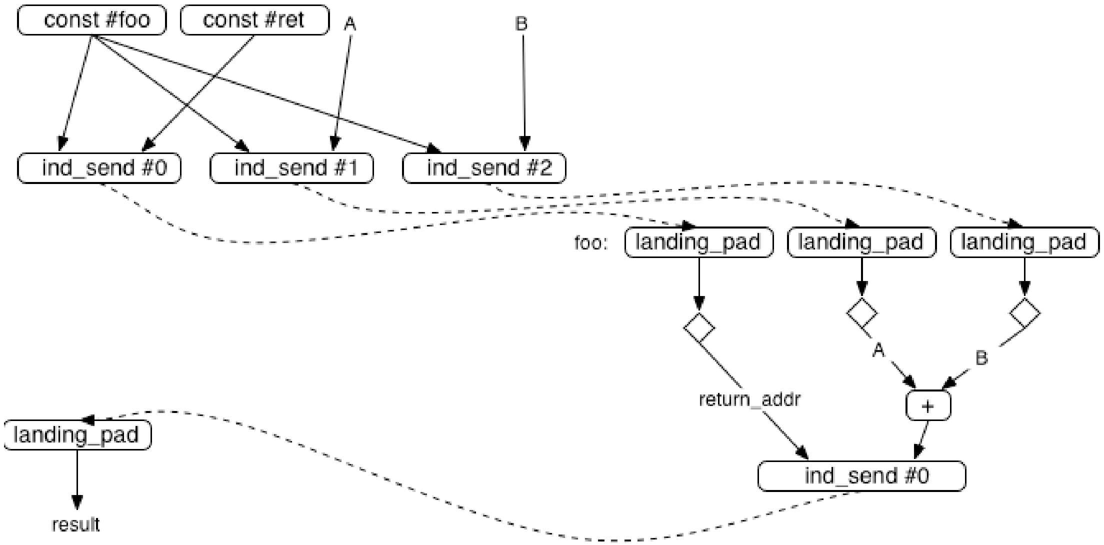
Steering Example

```
if (A > 0)
        D = C + B;
else
        D = C - E;
F = D + 1;
```



Control: Jumps

- Sometimes, destinations must be resolved dynamically
  - Indirect send, indirect receive
  - Dynamic resolution is fairly slow
- Macros will be provided for function calls and returns



Control: Wave Management

- Wave advance is an optimization
  - Only increments wave numbers
- Wave number manipulation is used to pass values around loops or complex control
  - Wave-to-data (wtd): outputs the wave number

    wtd {wave-as-output}, {input}
  - Data-to-wave (dtw): sets a wave number

    dtw {output}, {new-wave-input}, {value-input}

Control: Thread Management

- Values can be passed between threads by altering the tags
  - Thread-to-data (ttd): outputs the thread id

    ttd {thread-as-output}, {input}
  - Data-to-thread (dtt): sets the thread id

    dtt {output}, {new-thread-input}, {value-input}
  - dttw: sets the thread id and wave number

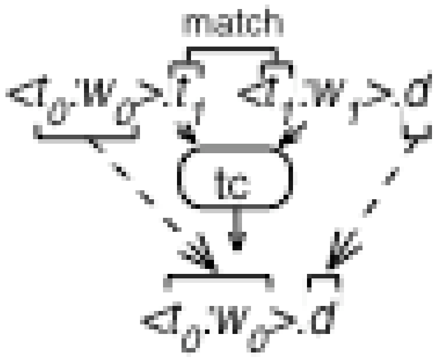
    dttw {output}, {thread}, {wave}, {value}

Concerns about Thread Management

- Sending values to a new thread is equivalent to an indirect send
  - Each thread has its own set of instructions
  - Destinations are resolved when the thread id is received
- Two kinds of threads exist
  - Light: unordered (or no) memory
    - Easy to create, requires very little support
  - Heavy: requires memory ordering support
    - If you want multiple memory orderings, talk to us first
- Thread ids should be unique across the system
  - Operating system concern

Synchronization

- For lightweight threads, lightweight synchronization is needed
  - Thread Coordinate (tc): implements a m-structure
- Requires a different firing rule


match
$<t_0:w_0>.t_1$
$<t_1:w_1>.d$
tc
$<t_0:w_0>.d$