Raw Programming 101

Volker Strumpen

strumpen@lcs.mit.edu

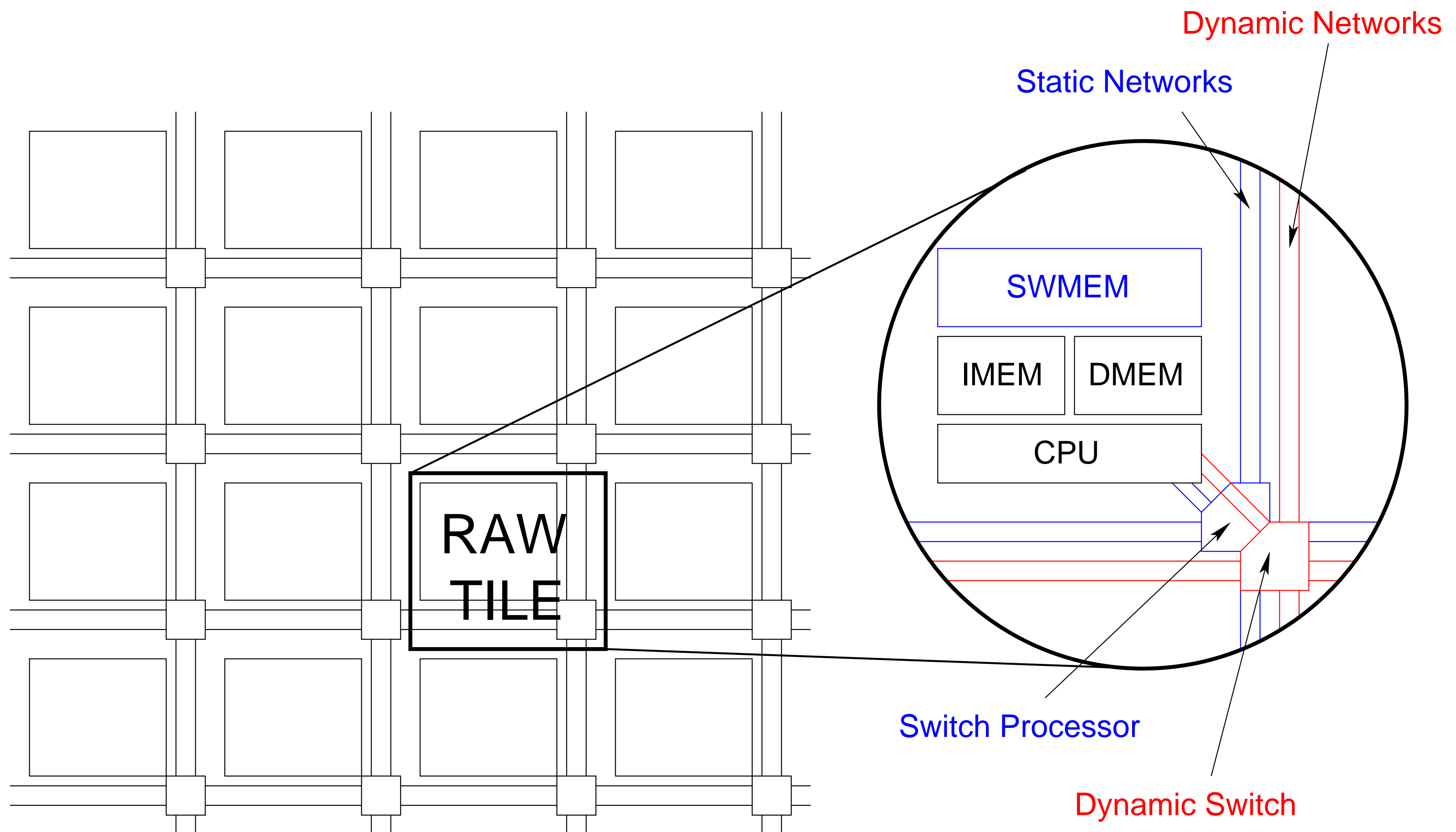
MIT Lab for Computer Science, Mar'03

Outline

1. Architectural Overview
   (a) The Raw Architecture
   (b) The Programmer's Perspective of Raw
2. Assembly Programming
   (a) $c = a + b$
   (b) Streaming Data
3. Inlining Assembly in C

Raw: A Tiled Architecture

We use Raw tiles to assemble 2-dimensional computational fabrics such as a 4x4 chip and multi-chip grids.

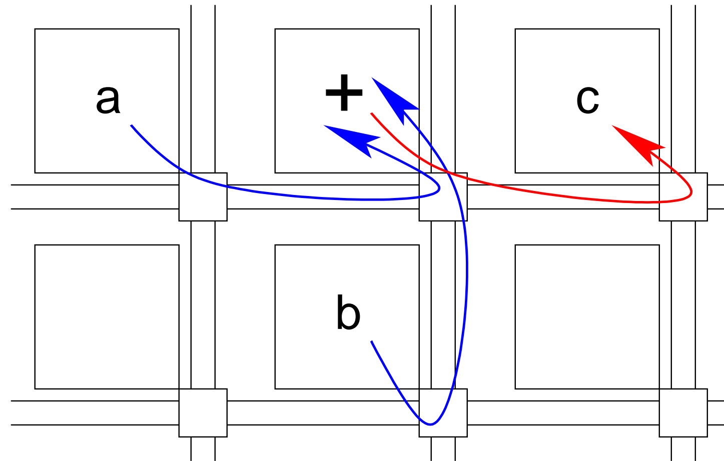


Features of MIT's 4x4 Raw Prototype

- 122 M Transistors on 0.15 μm ASIC (IBM) at 250 MHz
- 4x4 Tiles
  - Single-issue, in-order, 8-stage pipeline (MIPS R4000)
  - Independent Switch Processor (VLIW)
  - On-Chip-Memory: 2 MByte (32 K I, 32 K D, 64 K SW)
  - Peak Performance: 4 GIPS.
- Four Register-Mapped Networks
  - 32-bit wide, full-duplex networks
  - Aggregate On-Chip Bandwidth: 2 Tb/s
  - Next-Neighbor Latency: 3 clock cycles
  - I/O Bandwidth: 200 Gb/s (14 channels, 1080 pins)

Programming Challenge

Compute the sum $c = a + b$ across four tiles:

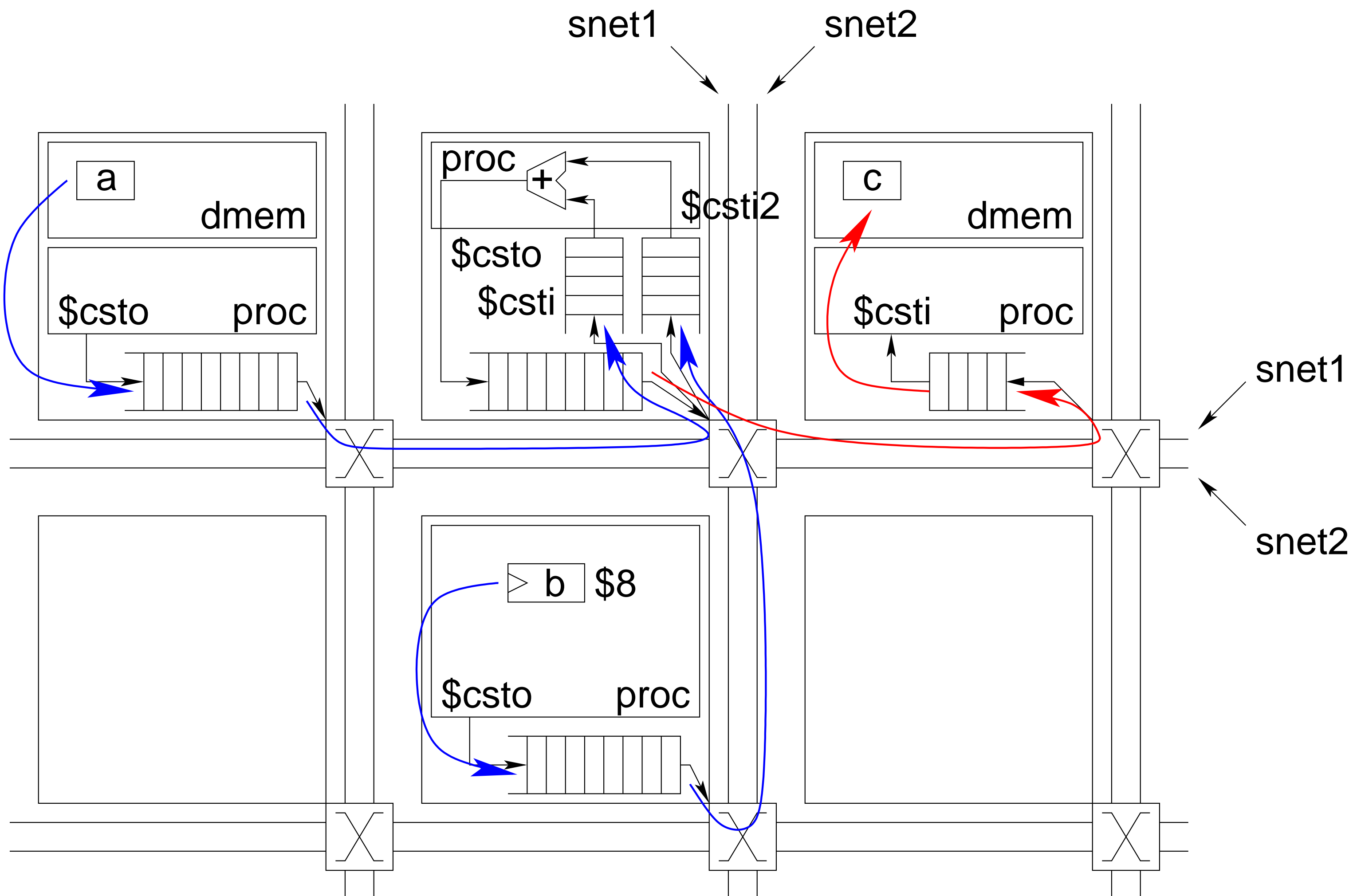


1. Where are the data paths?

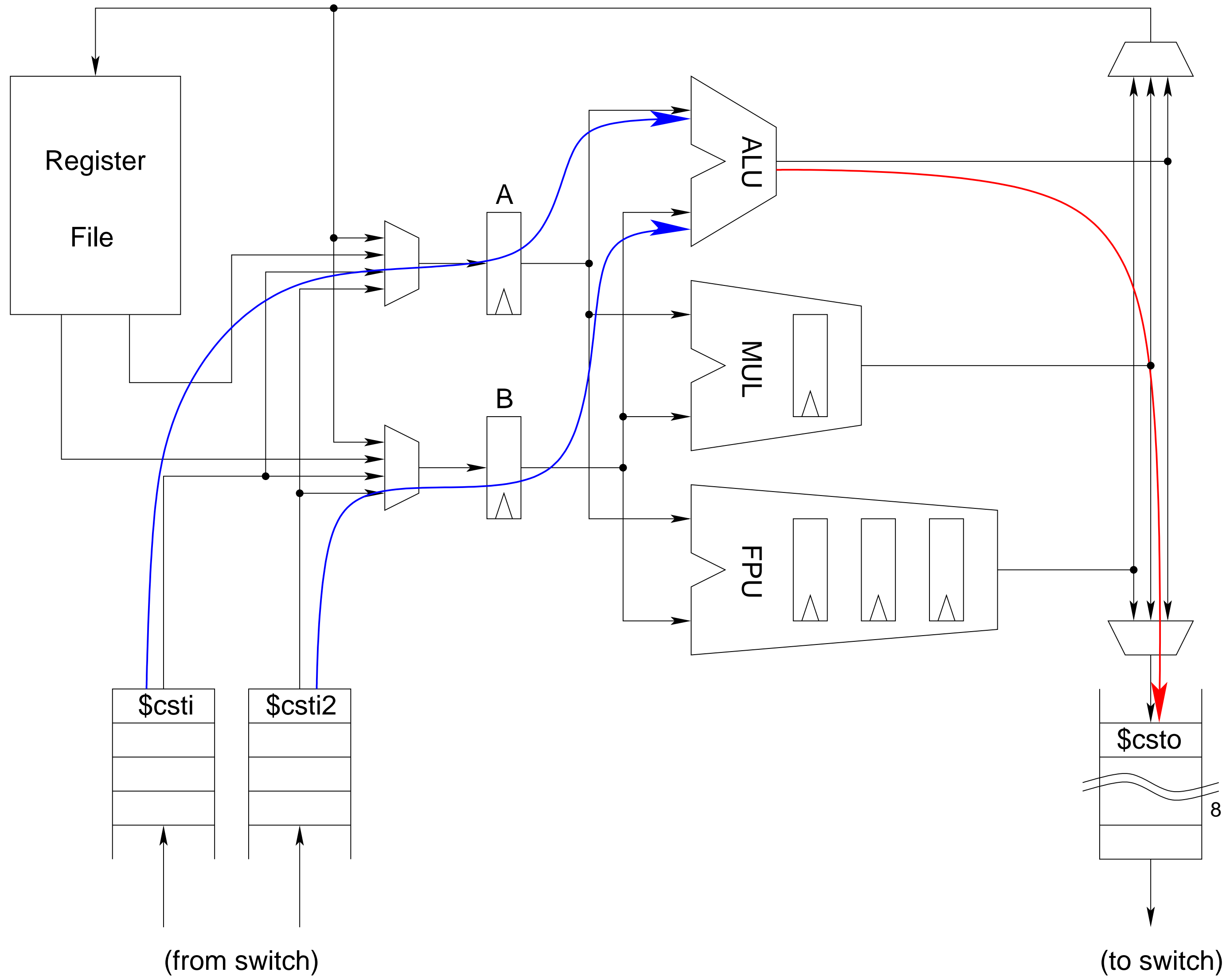
2. How do we program the processor and the switch?

Data Paths: Zoom 1

Stateful hardware: local data memory (a,c), registers (b), and both static networks (snet1,snet2).

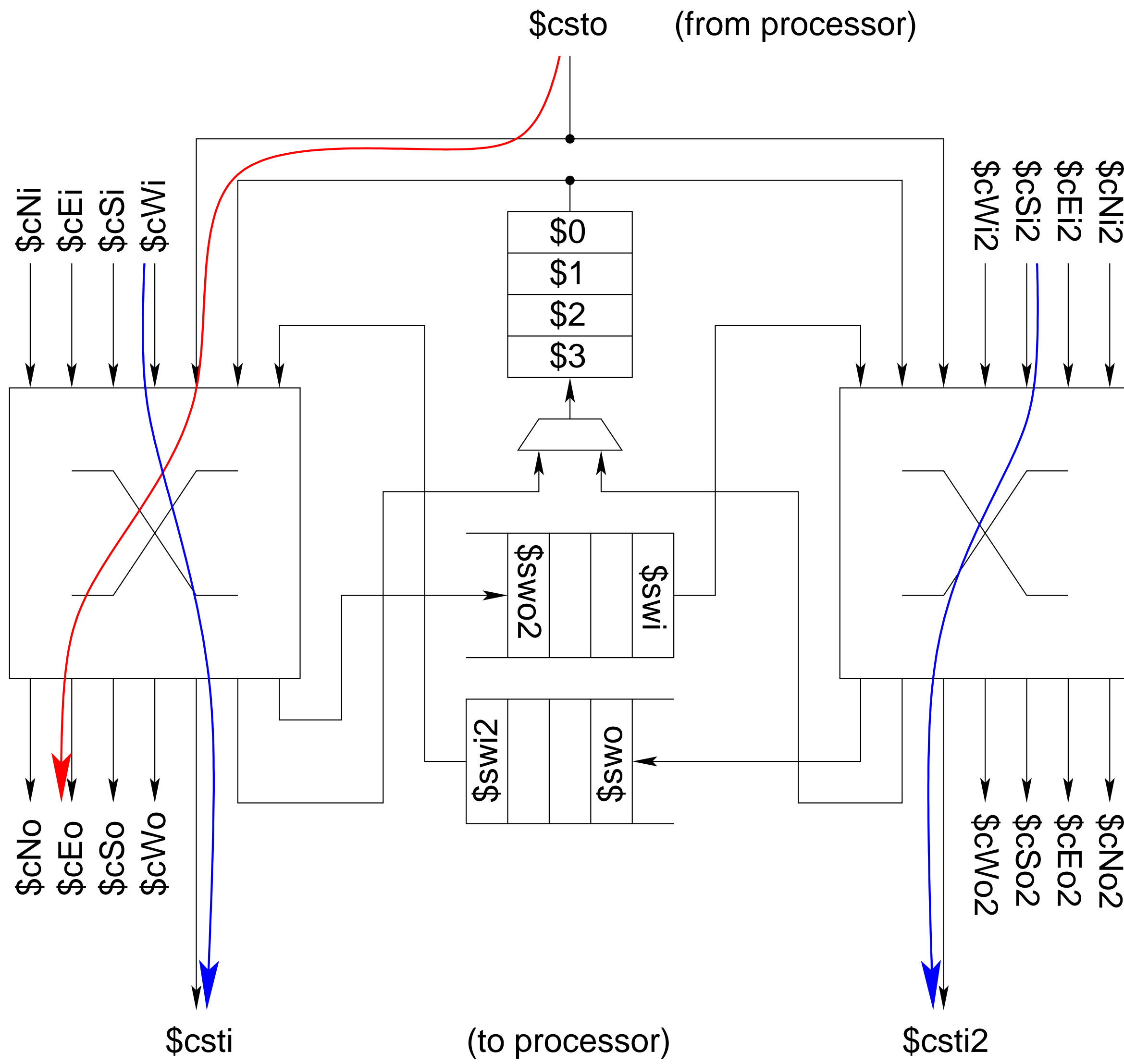


Zoom 2: Processor Datapaths



Note: Divider datapath omitted

Zoom 2: Switch Datapaths



Raw Assembly

a-tile processor: `lw $csto,0(&a)`

switch: `nop  route $csto->$cEo`

b-tile processor: `move $csto,$8`

switch: `nop  route $csto->$cNo2`

c-tile processor: `sw $csti,0(&c)`

switch: `nop  route $cWi->$csti`

+-tile processor: `addu $csto,$csti,$csti2`

switch:

```
nop  route $csto->$cEo, $cWi->$csti, $cSi2->$csti2
```

Switch Assembly

```
nop  route $csto->$cEo, $cWi->$csti, $cSi2->$csti2
```

Deadlocks!

Correct solution requires two switch instructions:

```
nop  route $cWi->$csti, $cSi2->$csti2
nop  route $csto->$cEo
```

Note: Switch instructions block until all operands are ready!

Streaming on Raw

Element-wise sum of two vectors:

```
for (i=0; i<N; i++)
  c[i] = a[i] + b[i];
```

With two switch instructions per addition, the upper bound for efficiency is to 50 %!

Loop overhead is the other source of inefficiency:

```
for (i=0; i<6; i++) {
  __asm__("addu $csto,$csti,$csti2");
}
```

```
        li   $2, 5
$L0:    addu $csto, $csti, $csti2
        addu $2, $2, -1
        bgez $2, $L0
```

Due to index decrement and branching, the upper bound for efficiency is 33 %! (25 % if ub is not constant)

Streaming Cont'd

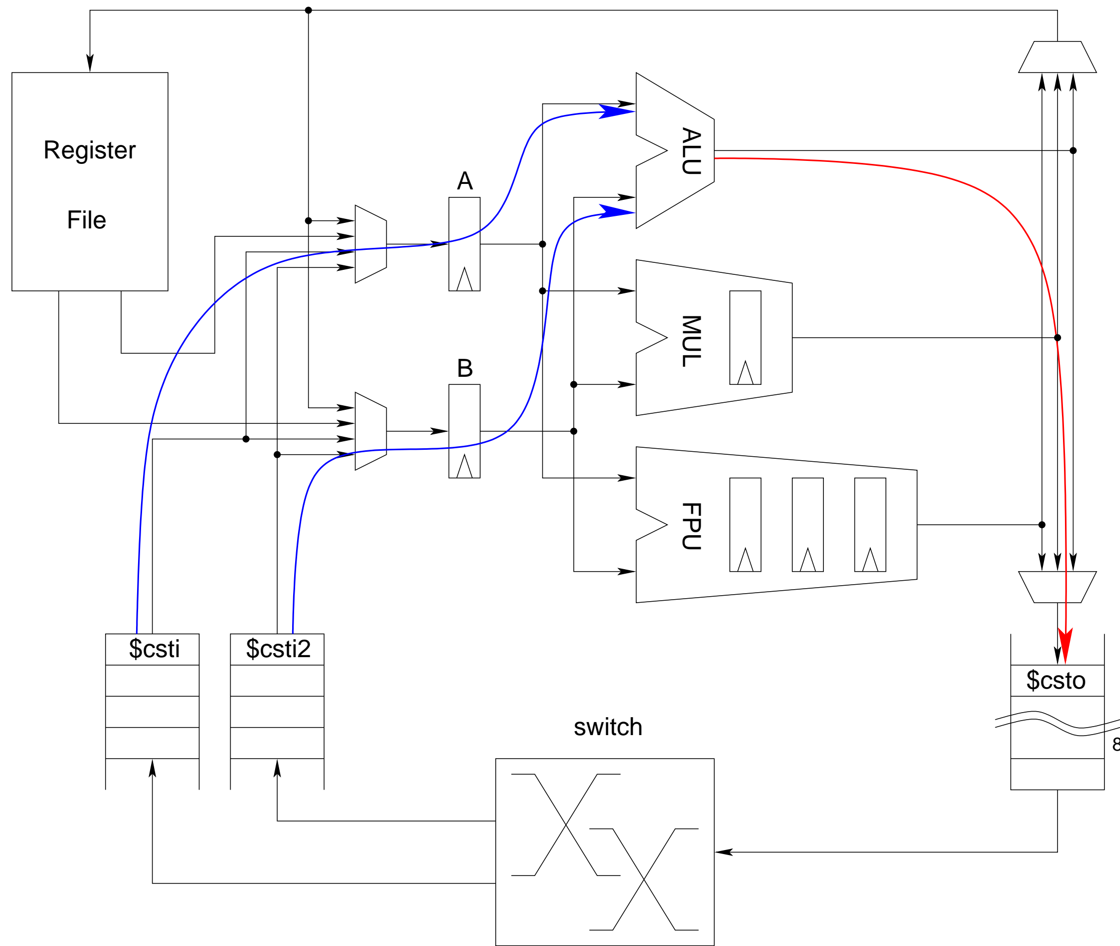
Key Techniques for Efficiency:

1. **pipeline startup/drainage (avoid deadlock)**
2. **loop unrolling (avoid loop overhead)**

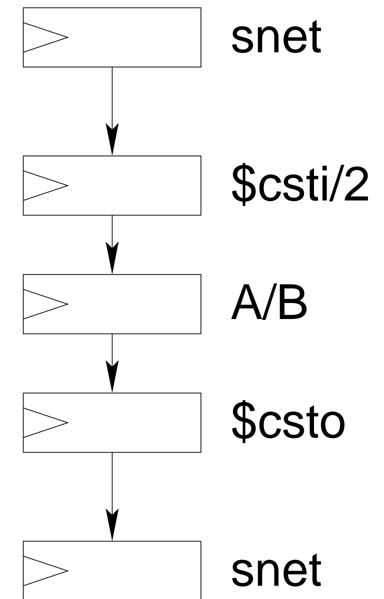
Completely unrolled loop with one startup step:

Processor	Switch
	`nop route $cWi->$csti, $cSi2->$csti2`
`addu $csto,$csti,$csti2`	`nop route $csto->$cEo, $cWi->$csti, $cSi2->$csti2`
`addu $csto,$csti,$csti2`	`nop route $csto->$cEo, $cWi->$csti, $cSi2->$csti2`
`addu $csto,$csti,$csti2`	`nop route $csto->$cEo, $cWi->$csti, $cSi2->$csti2`
`addu $csto,$csti,$csti2`	`nop route $csto->$cEo, $cWi->$csti, $cSi2->$csti2`
`addu $csto,$csti,$csti2`	`nop route $csto->$cEo, $cWi->$csti, $cSi2->$csti2`
`addu $csto,$csti,$csti2`	`nop route $csto->$cEo`

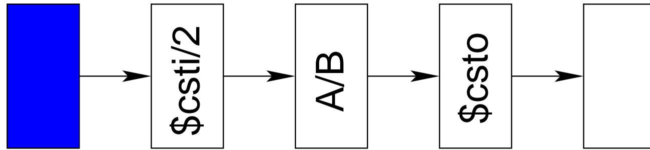
Abstract Adder Pipeline



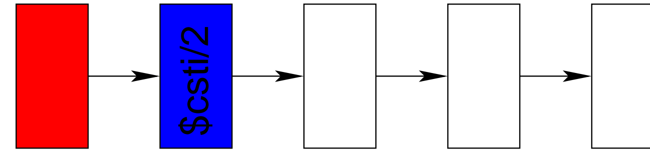
5-stage Pipeline



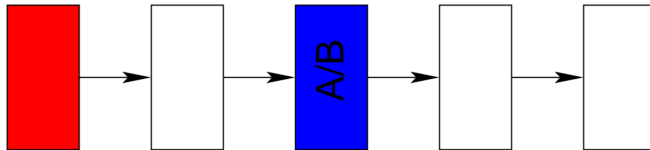
Execution Chart of Adder Pipeline



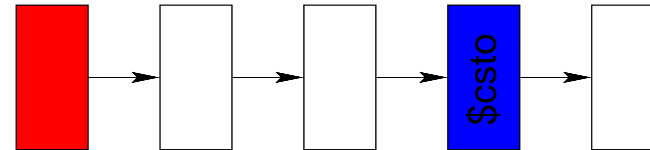
(1) $cWi–>$csti, $cSi2–>$csti2



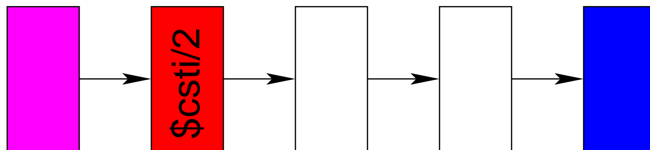
(2) $csto–>$cEo, $cWi–>$csti, $cSi2–>$csti2



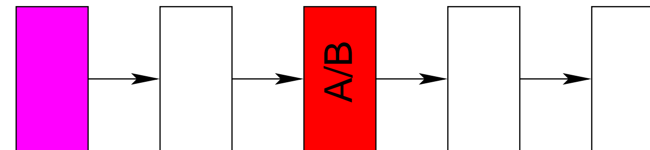
(3) stalled on $csto



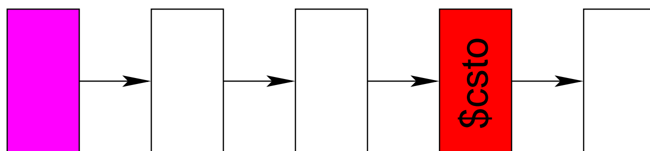
(4) executes



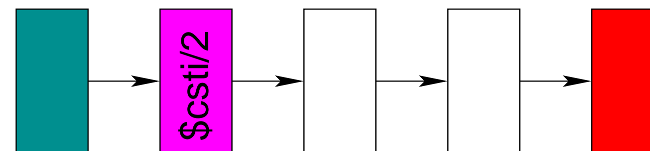
(5) $csto–>$cEo, $cWi–>$csti, $cSi2–>$csti2



(6) stalled on $csto



(7) executes



(8) $csto–>$cEo, $cWi–>$csti, $cSi2–>$csti2

Streaming Cont'd

Magic number of startup steps for 100 % efficiency: 3 steps

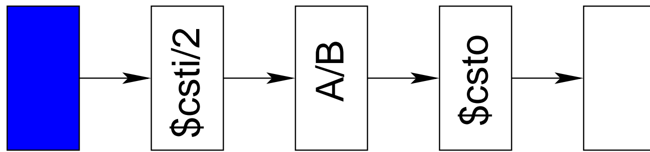
Processor

```
addu $csto,$csti,$csti2
addu $csto,$csti,$csti2
addu $csto,$csti,$csti2
addu $csto,$csti,$csti2
addu $csto,$csti,$csti2
addu $csto,$csti,$csti2
```

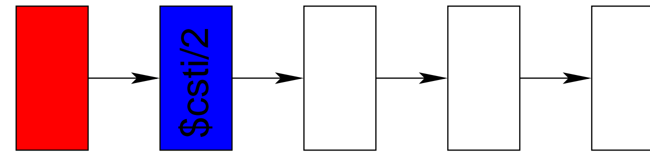
Switch

```
nop  route $cWi->$csti, $cSi2->$csti2
nop  route $cWi->$csti, $cSi2->$csti2
nop  route $cWi->$csti, $cSi2->$csti2
nop  route $csto->$cEo, $cWi->$csti, $cSi2->$csti2
nop  route $csto->$cEo, $cWi->$csti, $cSi2->$csti2
nop  route $csto->$cEo, $cWi->$csti, $cSi2->$csti2
nop  route $csto->$cEo
nop  route $csto->$cEo
nop  route $csto->$cEo
```

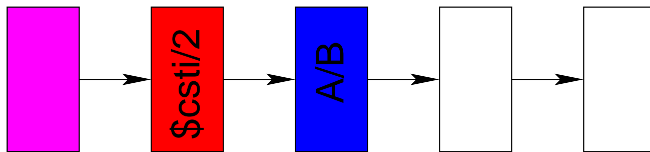
Execution Chart of Adder Pipeline



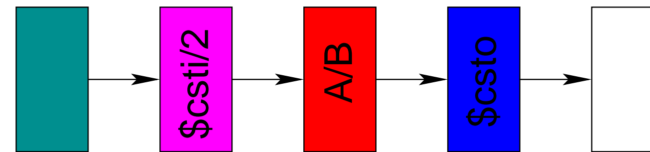
(1) $cWi–>$csti, $cSi2–>$csti2



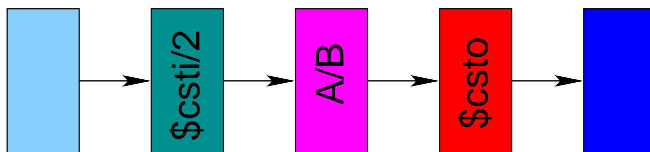
(2) $cWi–>$csti, $cSi2–>$csti2



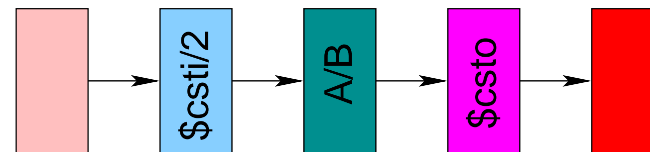
(3) $cWi–>$csti, $cSi2–>$csti2



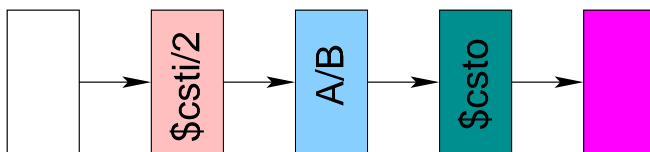
(4) $csto–>$cEo, $cWi–>$csti, $cSi2–>$csti2



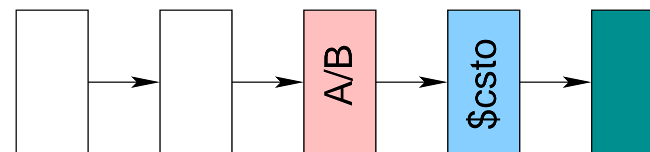
(5) $csto–>$cEo, $cWi–>$csti, $cSi2–>$csti2



(6) $csto–>$cEo, $cWi–>$csti, $cSi2–>$csti2



(7) $csto–>$cEo



(8) $csto–>$cEo

Inlining with rgcc

Consult gcc info topic: C extensions / extended Asm

Processor code: file “p.c”

```
extern void myroute(void);

void main(void)
{
  int i;

  __asm__ volatile("mtsr SW_PC,%0" : : "r" (myroute));
  for (i=0; i<8; i+=4) {
    __asm__ volatile("addu $csto,$csti,$csti2");
    __asm__ volatile("addu $csto,$csti,$csti2");
    __asm__ volatile("addu $csto,$csti,$csti2");
    __asm__ volatile("addu $csto,$csti,$csti2");
  }
}
```

Switch Assembly with rgcc

Switch code: file “sw.S”

```
.swtext
.global myroute

myroute:
   nop  $cWi->$csti, $cSi2->$csti2
   nop  $cWi->$csti, $cSi2->$csti2
   nop  $cWi->$csti, $cSi2->$csti2
   nop  route $csto->$cEo, $cWi->$csti, $cSi2->$csti2
   nop  route $csto->$cEo, $cWi->$csti, $cSi2->$csti2
   nop  route $csto->$cEo, $cWi->$csti, $cSi2->$csti2
   nop  route $csto->$cEo, $cWi->$csti, $cSi2->$csti2
   nop  route $csto->$cEo, $cWi->$csti, $cSi2->$csti2
   nop  route $csto->$cEo
   nop  route $csto->$cEo
   nop  route $csto->$cEo
   j .
```

Conclusion

1. Introduced Raw programming
2. Demonstrated potential for deadlock and inefficient programming
3. Illustrated how to achieve 100 % efficiency by
   (a) loop unrolling
   (b) pipeline startup/drainage
4. Showed how inlining in C supports assembly wimps