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?
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

$\text{cEi}$ $\text{cSi}$ $\text{cWi}$ $\text{cNi}$ $\text{cWo}$ $\text{cSo}$ $\text{cEo}$ $\text{cNo}$ $\text{cWo2}$ $\text{cSo2}$ $\text{cEo2}$ $\text{cNo2}$ $\text{cNi2}$ $\text{cSi2}$ $\text{cWi2}$

(from processor) $\text{csto}$ $(to\ processor)$ $\text{csti}$ $\text{csti2}$

$\text{swi}$ $\text{swi2}$

$0$ $1$ $2$ $3$
### Raw Assembly

<table>
<thead>
<tr>
<th>Processor</th>
<th>Instruction 1</th>
<th>Instruction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-tile</td>
<td>lw $csto,0(&amp;a)</td>
<td>nop route $csto-&gt;$cEo</td>
</tr>
<tr>
<td>switch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b-tile</td>
<td>move $csto,$8</td>
<td>nop route $csto-&gt;$cNo2</td>
</tr>
<tr>
<td>switch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c-tile</td>
<td>sw $csti,0(&amp;c)</td>
<td>nop route $cWi-&gt;$csti</td>
</tr>
<tr>
<td>switch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-tile</td>
<td>addu $csto,$csti,$csti2</td>
<td>nop route $csto-&gt;$cEo, $cWi-&gt;$csti, $cSi2-&gt;$csti2</td>
</tr>
<tr>
<td>switch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Switch Assembly

Deadlocks!

Correct solution requires two switch instructions:

```assembly
nop  route $csto->$cEo, $cWi->$csti, $cSi2->$csti2
```
Streaming on Raw

Element-wise sum of two vectors:

```plaintext
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:

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

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:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu $csto,$csti,$csti2</td>
<td>nop route $cWi-&gt;$csti, $cSi2-&gt;$csti2</td>
</tr>
<tr>
<td>addu $csto,$csti,$csti2</td>
<td>nop route $csto-&gt;$cEo, $cWi-&gt;$csti, $cSi2-&gt;$csti2</td>
</tr>
<tr>
<td>addu $csto,$csti,$csti2</td>
<td>nop route $csto-&gt;$cEo, $cWi-&gt;$csti, $cSi2-&gt;$csti2</td>
</tr>
<tr>
<td>addu $csto,$csti,$csti2</td>
<td>nop route $csto-&gt;$cEo, $cWi-&gt;$csti, $cSi2-&gt;$csti2</td>
</tr>
<tr>
<td>addu $csto,$csti,$csti2</td>
<td>nop route $csto-&gt;$cEo, $cWi-&gt;$csti, $cSi2-&gt;$csti2</td>
</tr>
<tr>
<td>addu $csto,$csti,$csti2</td>
<td>nop route $csto-&gt;$cEo, $cWi-&gt;$csti, $cSi2-&gt;$csti2</td>
</tr>
<tr>
<td>addu $csto,$csti,$csti2</td>
<td>nop route $csto-&gt;$cEo, $cWi-&gt;$csti, $cSi2-&gt;$csti2</td>
</tr>
<tr>
<td>addu $csto,$csti,$csti2</td>
<td>nop route $csto-&gt;$cEo</td>
</tr>
<tr>
<td>addu $csto,$csti,$csti2</td>
<td>nop route $csto-&gt;$cEo</td>
</tr>
</tbody>
</table>
Abstract Adder Pipeline

5-stage Pipeline
**Execution Chart of Adder Pipeline**

1. $cWi\rightarrow$costi, $cSi2\rightarrow$costi2
2. $costo\rightarrow$cEo, $cWi\rightarrow$costi, $cSi2\rightarrow$costi2
3. stalled on $costo$
4. executes
5. $costo\rightarrow$cEo, $cWi\rightarrow$costi, $cSi2\rightarrow$costi2
6. stalled on $costo$
7. executes
8. $costo\rightarrow$cEo, $cWi\rightarrow$costi, $cSi2\rightarrow$costi2
## Streaming Cont’d

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

<table>
<thead>
<tr>
<th>Processor</th>
<th>Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addu $csto,$csti,$csti2</code></td>
<td><code>nop</code> route <code>$cWi-&gt;$csti, $cSi2-&gt;$csti2</code></td>
</tr>
<tr>
<td><code>addu $csto,$csti,$csti2</code></td>
<td><code>nop</code> route <code>$cWi-&gt;$csti, $cSi2-&gt;$csti2</code></td>
</tr>
<tr>
<td><code>addu $csto,$csti,$csti2</code></td>
<td><code>nop</code> route <code>$cWi-&gt;$csti, $cSi2-&gt;$csti2</code></td>
</tr>
<tr>
<td><code>addu $csto,$csti,$csti2</code></td>
<td><code>nop</code> route <code>$csto-&gt;$cEo, $cWi-&gt;$csti, $cSi2-&gt;$csti2</code></td>
</tr>
<tr>
<td><code>addu $csto,$csti,$csti2</code></td>
<td><code>nop</code> route <code>$csto-&gt;$cEo, $cWi-&gt;$csti, $cSi2-&gt;$csti2</code></td>
</tr>
<tr>
<td><code>addu $csto,$csti,$csti2</code></td>
<td><code>NOP</code> route <code>$csto-&gt;$cEo, $cWi-&gt;$csti, $cSi2-&gt;$csti2</code></td>
</tr>
<tr>
<td><code>addu $csto,$csti,$csti2</code></td>
<td><code>NOP</code> route <code>$csto-&gt;$cEo, $cWi-&gt;$csti, $cSi2-&gt;$csti2</code></td>
</tr>
<tr>
<td><code>addu $csto,$csti,$csti2</code></td>
<td><code>NOP</code> route <code>$csto-&gt;$cEo</code></td>
</tr>
<tr>
<td><code>addu $csto,$csti,$csti2</code></td>
<td><code>NOP</code> route <code>$csto-&gt;$cEo</code></td>
</tr>
<tr>
<td><code>addu $csto,$csti,$csti2</code></td>
<td><code>NOP</code> route <code>$csto-&gt;$cEo</code></td>
</tr>
</tbody>
</table>
Execution Chart of Adder Pipeline

1. \( c_{Wi} \rightarrow c_{sti}, \ c_{Si2} \rightarrow c_{sti2} \)
2. \( c_{Wi} \rightarrow c_{sti}, \ c_{Si2} \rightarrow c_{sti2} \)
3. \( c_{Wi} \rightarrow c_{sti}, \ c_{Si2} \rightarrow c_{sti2} \)
4. \( c_{sto} \rightarrow c_{Eo}, \ c_{Wi} \rightarrow c_{sti}, \ c_{Si2} \rightarrow c_{sti2} \)
5. \( c_{sts} \rightarrow c_{Eo}, \ c_{Wi} \rightarrow c_{sti}, \ c_{Si2} \rightarrow c_{sti2} \)
6. \( c_{sto} \rightarrow c_{Eo}, \ c_{Wi} \rightarrow c_{sti}, \ c_{Si2} \rightarrow c_{sti2} \)
7. \( c_{sto} \rightarrow c_{Eo} \)
8. \( c_{sto} \rightarrow c_{Eo} \)
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”

```assembly
.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, $cWi->$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