What Do We Do With $10^{12}$ Transistors?
The Case for Precision Timing

Stephen A. Edwards
Columbia University
What Not To Do

• Not just a single CPU
  Processor architects have already given up trying to figure out how to waste that many transistors

• Not just one big memory
  Von Neumann Bottleneck
  $10^{12}$ bits vs. a 1 GHz clock: minutes
What Not To Do

- Not “Internet-on-a-chip” (TCP/IP over Ethernet)
  On-chip communication more reliable
  No on-chip backhoes to worry about
  We are not good at programming these anyway

- Not just an FPGA
  Non-software systems disappeared in the early 1980s
  Every interesting system has lots of software
What We Probably Will Do

An FPGA-like mesh of computational elements floating in a sea of communication.
What Sort Of Processor?

Hypothesis: it should be a precision-timed “PRET” processor
Embedded Systems Dominate

- In 2004, 97% of the 6.5 billion processors shipped went into embedded system.

- In 2004, 674 million cell phones sold, 3.3 billion total subscribers  
  2004 world population: 6.4 billion

- 100 processors in a typical automobile
Embedded Application Areas

Hard real-time systems

- Avionics
- Automotive
- Multimedia
- Consumer Electronics
We do not consider how fast a processor runs when we evaluate whether it is “correct.”

This Is Sometimes Useful For

- Programming languages
- Virtual memory
- Caches
- Dynamic dispatch
- Speculative execution
- Power management (voltage scaling)
- Memory management (garbage collection)
- Just-in-time (JIT) compilation
- Multitasking (threads and processes)
- Component technologies (OO design)
- Networking (TCP)
But Time Sometimes Matters

Kevin Harvick winning the Daytona 500 by 20 ms, February 2007. (Source: Reuters)
Isn’t Real-Time Scheduling Solved?

Fixed-priority (RMA): schedulable if $< 69\%$ utilization

Variable-priority (EDF): schedulable if $< 100\%$ utilization

Hinges on knowing task execution times
Worst-Case Execution Time

Virtually impossible to compute on modern processors.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Nearby instructions</th>
<th>Distant instructions</th>
<th>Memory layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipelines</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Caches</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Processors are Actually Chaotic


Sprott, *Strange Attractors*, Herring Figure 5–13.
State-of-the-art WCET

- Motorola ColdFire
- Two coupled pipelines (7-stage)
- Shared instruction & data cache
- Artificial example from Airbus
- Twelve independent tasks
- Simple control structures
- Cache/Pipeline interaction leads to large integer linear programming problem

C. Ferdinand et al., “Reliable and precise WCET determination for a real-life processor,” EMSOFT 2001
The Problem

Digital hardware provides extremely precise timing

20.000 MHz (± 100 ppm)

and modern architectural complexity discards it.
Our Vision: PRET Machines

PREcision-Timed processors: Performance & Predicability

(Image: John Harrison’s H4, first clock to solve longitude problem)
Caches and Memory Hierarchy?

Our goal: a predictable memory hierarchy
Use software-managed scratchpads with compiler support

Well-studied:
Panda et al. [EDAC 1997],
Kandemir et al. [DAC 2001, 2002],
Banakar et al. [CODES 2002],
Angiolini et al. [CASES 2003, 2004],
Udaykumaran et al. [CASES 2003],
Verma et al. [DATE 2004],
Francesco et al. [DAC 2004],
Dominguez et al. [JES 2005],
Li et al. [PACT 2005],
Egger et al. [Emsoft 2006],
Janapsatya et al. [ASPDAC 2006].
Pipelines?

Use thread-interleaved pipelines to avoid hazards

An old idea (60s): one thread per pipeline stage

Like Simultaneous Multi-threading, but it works

Interrupts?

One processor per interrupt source

Use polling; more predictable

I/O processors have a long history anyway

Really a way to share the processor resource across I/O sources

 Isn’t this wickedly inefficient?
Go Ahead: Leave Processors Idle

Modern processors do this at the functional unit level. Schuette and Shen (MICRO 1991) found for their VLIW,

<table>
<thead>
<tr>
<th>Unit</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Fetch Unit</td>
<td>12–44%</td>
</tr>
<tr>
<td>Floating-point Fetch Unit</td>
<td>7–23%</td>
</tr>
<tr>
<td>Integer Registers</td>
<td>4–37%</td>
</tr>
<tr>
<td>Floating-point Registers</td>
<td>8–25%</td>
</tr>
<tr>
<td>Shared Registers</td>
<td>1–65%</td>
</tr>
<tr>
<td>Integer Bus</td>
<td>1–22%</td>
</tr>
<tr>
<td>Floating-point Bus</td>
<td>4–25%</td>
</tr>
<tr>
<td>Shared Bus</td>
<td>2–5%</td>
</tr>
<tr>
<td>Address Bus</td>
<td>2–37%</td>
</tr>
</tbody>
</table>

This is actually a good thing for power
Communication?

Use time-triggered busses (statically scheduled, periodic)
Examples: FlexRay, TTP, ATM

Source: TZM
Shared Resources?

Like communication, scheduled, periodic access sharing

First Ferris Wheel, 1893 World’s Columbian Exposition, Chicago
The Parallax Propeller Chip

- Power Up Detector (~10 ms)
- Reset Delay (~50 ms)
- Clock PLL
  - 1x, 2x, 4x, 8x, 16x (16x must be 64-128 MHz)
- RC Oscillator
  - 12 MHz / 20 KHz
- Crystal Oscillator
  - DC - 80 MHz
  - (4 - 8 MHz with Clock PLL)
- Brown Out Detector
- BOEn
- RESn
- PLLENA
- Clock Selector (MUX)
- Clock PLL
- SOFTRES
- OSCENA
- OSCMODE
- VDD
- VSS
- System Counter
- Data Bus
- Address Bus
- Clock Counter
- Pin Inputs
- Pin Outputs
- I/O Pins
- A0
- A1
- A2
- A3
- A4
- A5
- A6
- A7
- A8
- A9
- A10
- A11
- A12
- A13
- A14
- A15
- A16
- A17
- A18
- A19
- A20
- A21
- A22
- A23
- A24
- A25
- A26
- A27
- A28
- A29
- A30
- A31
- Hub
- Cog
- Pin Directions
- Pin Outputs
- System Counter
- Configuration Register
- 8192 x 32 RAM
- 8192 x 32 ROM
- Bus Clock (+2)
- Cog Clocks (+1)
- Cog Enables
- Semaphores (8)
The Parallax Propeller Chip

- 80 MHz low-power (<300mW) full-custom IC
- 8 32-bit, 20 MIPS “cog” processors each w/ 2K RAM
- 32K + 32K of round-robin-shared RAM and ROM
- On reset, program in main RAM copied to local ones
- Most instructions take 4 cycles
- To access main memory, wait for turn (7–22 cycles)
- No interrupts: devote one or more cogs to I/O
- ROM includes font with symbols for transistors, timing diagrams (!)
Operating System?

Process scheduling not necessary

Resource allocation largely static

Hardware abstraction layer (device drivers, etc.) useful
An Example: An ISA with Timing

MIPS-like processor with 16-bit data path as proof of concept for ISAs with timing

One additional “deadline” instruction:

```
dead timer, timeout
```

Wait until `timer` expires, then immediately reload it with `timeout`.

# Programmer’s Model

## General-purpose Registers

| $0$ $(= 0)$ |
| $1$ |
| $2$ |
| $\vdots$ |
| $13$ |
| $14$ |
| $15$ |

## Timers

| $t0$ |
| $t1$ |
| $t2$ |
| $t3$ |

## Program counter

| $pc$ |
### Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Mnemonic</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td><code>add</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td>Addi</td>
<td><code>addi</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td>And</td>
<td><code>and</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td>Andi</td>
<td><code>andi</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td>Be</td>
<td><code>be</code></td>
<td>Rd, Rs, offset</td>
</tr>
<tr>
<td>Bne</td>
<td><code>bne</code></td>
<td>Rd, Rs, offset</td>
</tr>
<tr>
<td>J</td>
<td><code>j</code></td>
<td>target</td>
</tr>
<tr>
<td>Lb</td>
<td><code>lb</code></td>
<td>Rd, (Rt + Rs)</td>
</tr>
<tr>
<td>Lbi</td>
<td><code>lbi</code></td>
<td>Rd, (Rs + offset)</td>
</tr>
<tr>
<td>Mov</td>
<td><code>mov</code></td>
<td>Rd, Rs</td>
</tr>
<tr>
<td>Movi</td>
<td><code>movi</code></td>
<td>Rd, imm16</td>
</tr>
<tr>
<td>Nand</td>
<td><code>nand</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td>Nandi</td>
<td><code>nandi</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td>Nor</td>
<td><code>nor</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td>Nori</td>
<td><code>nori</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td>Or</td>
<td><code>or</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td>Ori</td>
<td><code>ori</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td>Sb</td>
<td><code>sb</code></td>
<td>Rd, (Rt + Rs)</td>
</tr>
<tr>
<td>Sbi</td>
<td><code>sbi</code></td>
<td>Rd, (Rs + offset)</td>
</tr>
<tr>
<td>Sll</td>
<td><code>sll</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td>Slli</td>
<td><code>slli</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td>Srl</td>
<td><code>srl</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td>Srli</td>
<td><code>srli</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td>Sub</td>
<td><code>sub</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td>Subi</td>
<td><code>subi</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td>Dead</td>
<td><code>dead</code></td>
<td>T, Rs</td>
</tr>
<tr>
<td>Deadi</td>
<td><code>deadi</code></td>
<td>T, imm16</td>
</tr>
<tr>
<td>Xnor</td>
<td><code>xnor</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td>Xnorri</td>
<td><code>xnori</code></td>
<td>Rd, Rs, imm16</td>
</tr>
<tr>
<td>Xor</td>
<td><code>xor</code></td>
<td>Rd, Rs, Rt</td>
</tr>
<tr>
<td>Xori</td>
<td><code>xori</code></td>
<td>Rd, Rs, imm16</td>
</tr>
</tbody>
</table>
Architecture

Program Memory: 128 × 32

Instruction Decoder

Data Memory: 16K × 8

Pixel
## Behavior of Dead

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
<th>$t0</th>
<th>$t1</th>
<th>$t2</th>
<th>$t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>deadi $t0, 8</td>
<td>8</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td>&quot;</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td>&quot;</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&quot;</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>add $r1, $r2, $r3</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>deadi $t0, 10</td>
<td>10</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>&quot;</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>&quot;</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>add $r1, $r2, $r3</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8 cycles
Case Study: Video

80 × 30 text-mode display, 25 MHz pixel clock

Need 40 ns precision

Shift register in hardware; everything else in software
Case Study: Video

movi $2, 0 ; reset line address
row:
  movi $7, 0 ; reset line in char
line:
  deadi $t1, 96 ; h. sync period
  movi $14, HS+HB
  ori $3, $7, FONT
  deadi $t1, 48 ; font base address
  deadi $t1, 640 ; active video period
  mov $1, 0 ; column number
char:
  lb $5, ($2+$1) ; load character
  shli $5, $5, 4 ; *16 = lines/char
  deadi $t0, 8 ; wait for next character
  lb $14, ($5+$3) ; fetch and emit pixels
  addi $1, $1, 1 ; next column
  bne $1, $11, char
  deadi $t1, 16 ; front porch period
  movi $14, HB
  addi $7, $7, 1 ; next row in char
  bne $7, $13, line ; repeat until bottom
  addi $2, $2, 80 ; next line
  bne $2, $12, row ; until at end

Two nested loops:
  - Active line
  - Character

Two timers:
  - $t1 for line timing
  - $t0 for character

78 lines of assembly replaces 450 lines of VHDL (1/5th)
Case Study: Serial Receiver

Sampling rate under software control

Standard algorithm:

1. Find falling edge of start bit
2. Wait half a bit time
3. Sample
4. Wait full bit time
5. Repeat 3. and 4.

movi $3, 0x0400 ; final bit mask (10 bits)
movi $5, 651 ; half bit time for 9600 baud
shli $6, $5, 1 ; calculate full bit time

wait_for_start:
  bne $15, $0, wait_for_start

wait $t1, $5 ; sample at center of bit
movi $14, 0 ; clear received byte
movi $2, 1 ; received bit mask
movi $4, 0 ; clear parity
dead $t1, $6 ; skip start bit

receive_bit:
  dead $t1, $6 ; wait until center of next bit
  mov $1, $15 ; sample
  xor $4, $4, $1 ; update parity
  and $1, $1, $2 ; mask the received bit
  or $14, $14, $1 ; accumulate result
  shli $2, $2, 1 ; advance to next bit
  bne $2, $3, receive_bit

check_parity:
  be $4, $0, detect_baud_rate
  andi $14, $14, 0xff ; discard parity and stop bits
Implementation

Synthesized on an Altera Cyclone II FPGA (DE2 board)

Coded in VHDL

Runs at 50 MHz

Unpipelined

Uses on-chip memory
Our Vision: PRET Machines

Predictable performance, not just good average case

<table>
<thead>
<tr>
<th>Current</th>
<th>Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caches</td>
<td>Scratchpads</td>
</tr>
<tr>
<td>Pipelines</td>
<td>Thread-interleaved pipelines</td>
</tr>
<tr>
<td>Function-only ISAs</td>
<td>ISAs with timing</td>
</tr>
<tr>
<td>Function-only languages</td>
<td>Languages with timing</td>
</tr>
<tr>
<td>Best-effort communication</td>
<td>Fixed-latency communication</td>
</tr>
<tr>
<td>Time-sharing</td>
<td>Multiple independent processors</td>
</tr>
</tbody>
</table>
Final Provocative Hypothesis

PRET will help parallel general-purpose applications by making their behavior reproducible.

Data races, non-atomic updates still a danger, but at least they can be reproduced.