



Precision-Timed (PRET) Machines

Stephen A. Edwards

Columbia University

Joint work with Edward A. Lee,
University of California, Berkeley

A Major Historical Event

In 1980, Patterson and Ditzel did not invent reduced instruction set computers (RISC machines).

D. A. Patterson and D. R. Ditzel, “The case for the reduced instruction set computer,” ACM SIGARCH Computer Architecture News, 8(6):25-33, Oct. 1980.

Another Major Historical Event

In 2006, Lee and Edwards did not invent reduced precision-timed computers (PRET machines).

S. A. Edwards and E. A. Lee, “The Case for the Precision Timed (PRET) Machine,” EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2006-149, November 17, 2006.

<http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-149.html>

The World as We Know It

We do not consider how fast a processor runs when we evaluate whether it is “correct.”



Salvador Dali, *The Persistence of Memory*, 1931. (detail)

This Is Sometimes Useful For

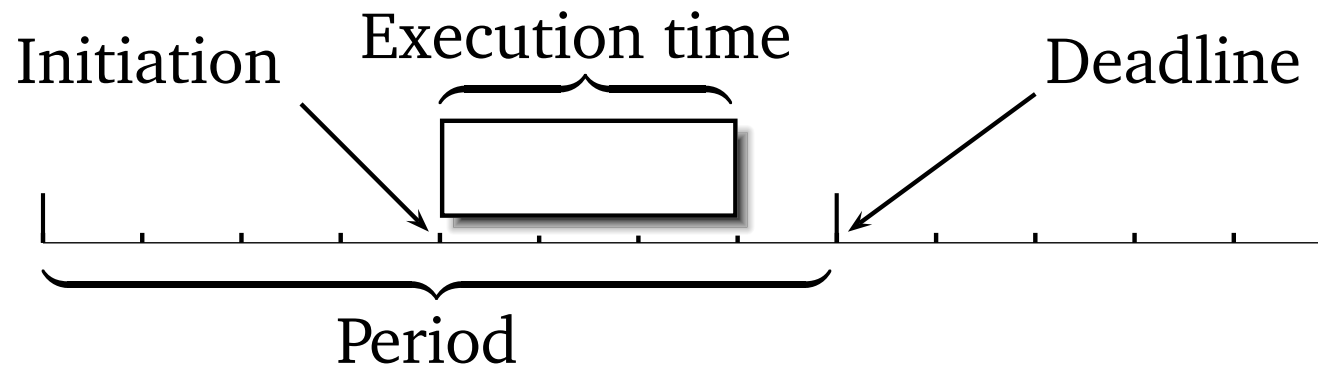
- Programming languages
- Virtual memory
- Caches
- Dynamic dispatch
- Speculative execution
- Power management (voltage scaling)
- Memory management (garbage collection)
- Just-in-time (JIT) compilation
- Multitasking (threads and processes)
- Component technologies (OO design)
- Networking (TCP)

But Time Sometimes Matters



Kevin Harvick winning the Daytona 500 by 20 ms, February 2007. (Source: Reuters)

Isn't Real-Time Scheduling Solved?

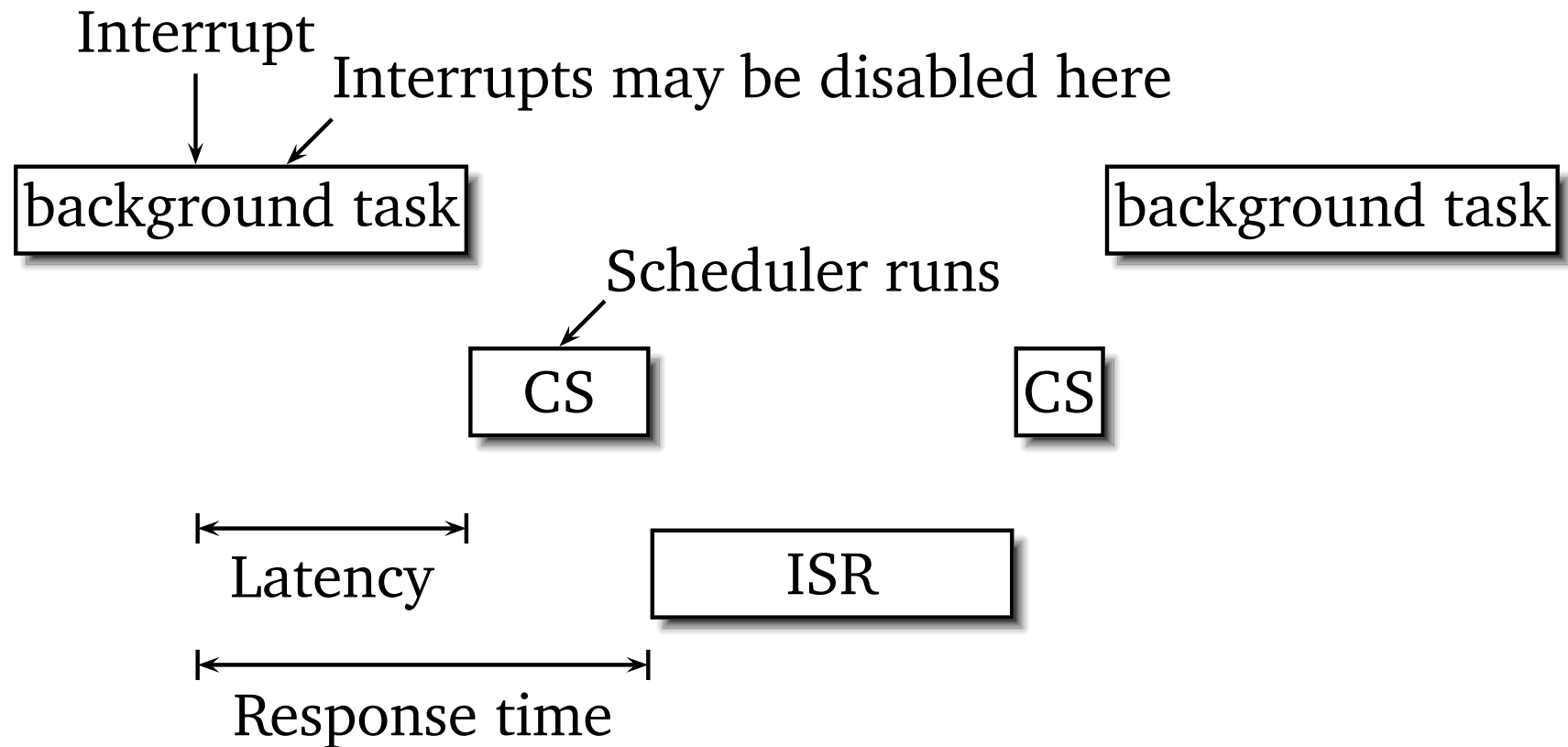


Fixed-priority (RMA): schedulable if $< 69\%$ utilization

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

Hinges on knowing task execution times

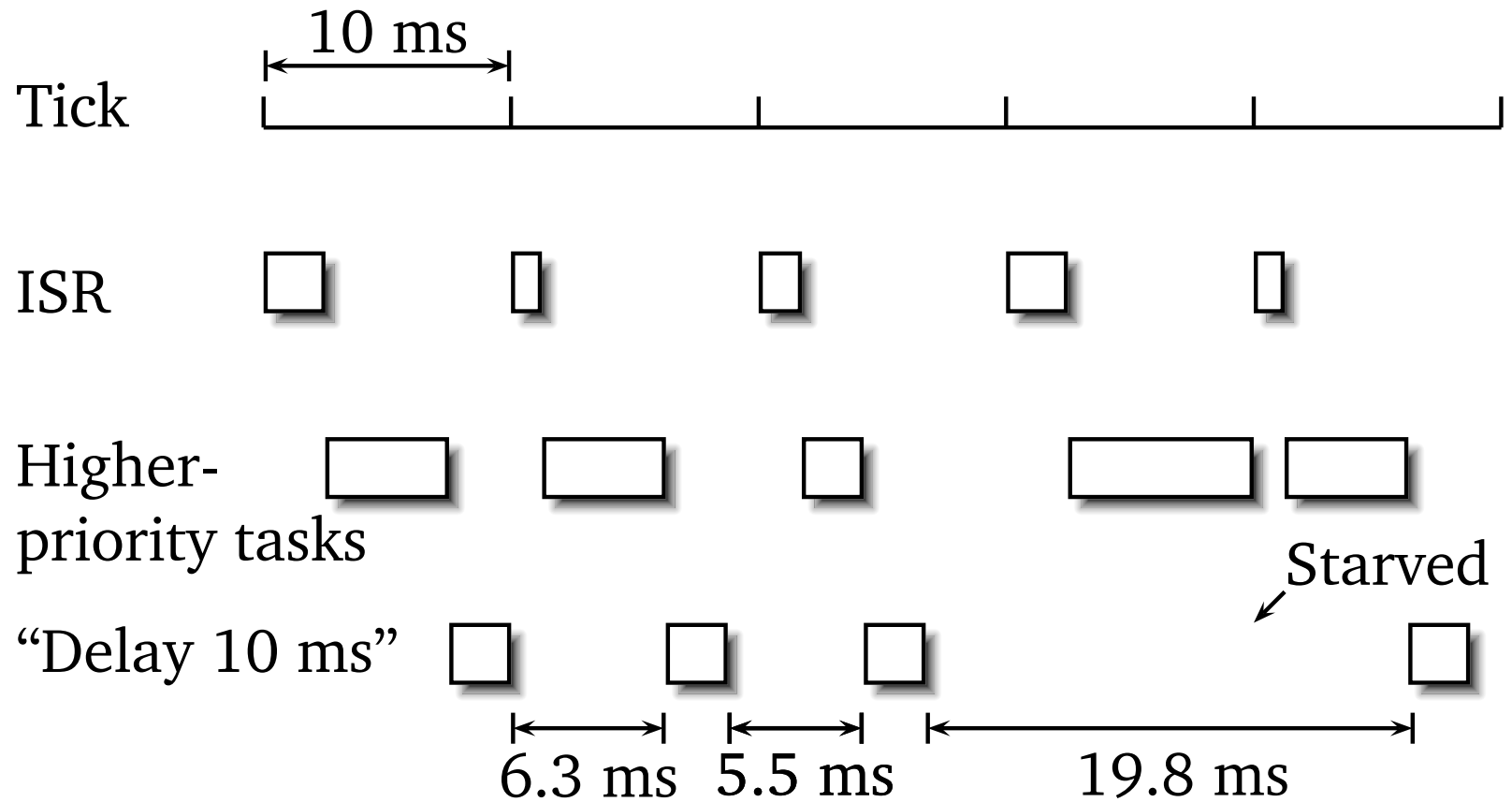
Interrupt Latency and Response



Need longest interrupt-disabled time + scheduling time

After Labrosse, *MicroC/OS-II: The Real-Time Kernel*, 1999.

Jitter from Delaying for One Tick



ISR time + other task time + our delay

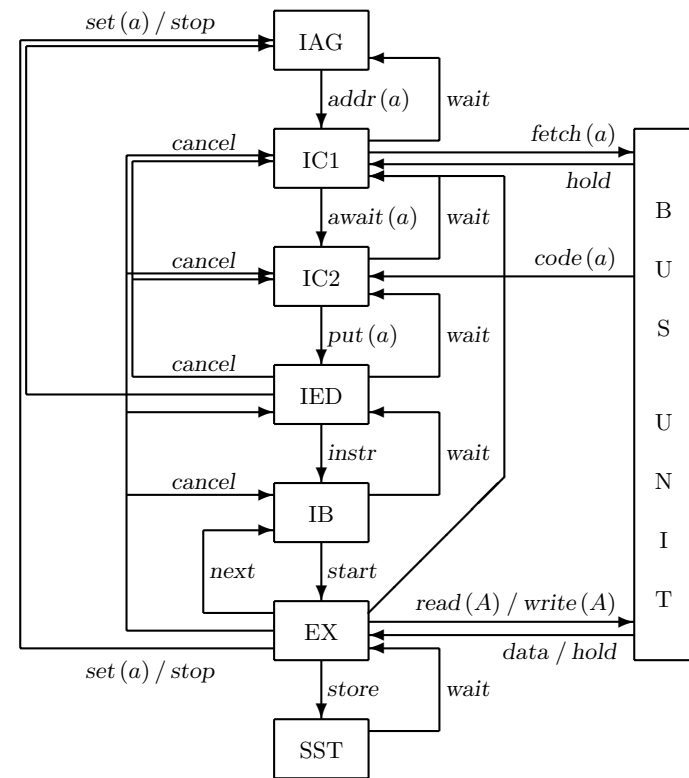
Worst-Case Execution Time

Virtually impossible to compute on modern processors.

Feature	Nearby instructions	Distant instructions	Memory layout
Pipelines	✓		
Branch Prediction	✓	✓	
Caches	✓	✓	✓

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

Certification in Avionics

- Rather expensive
- Software is *not* certified
- Entire system is certified
- Slight change, e.g., in the microprocessor, requires recertification
- Solution: stockpile parts; trust nobody



(Source: NASA)

The Problem

Digital hardware provides extremely precise timing



20.000 MHz (± 100 ppm)

and architectural complexity discards it.

Our Vision: PRET Machines

PREcision-Timed processors: Performance & Predicability



(Image: John Harrison's H4, first clock to solve longitude problem)

Our Vision: PRET Machines

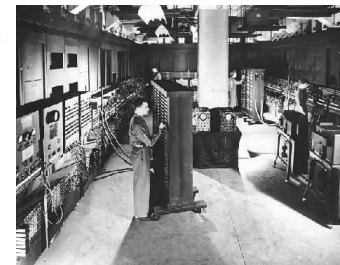
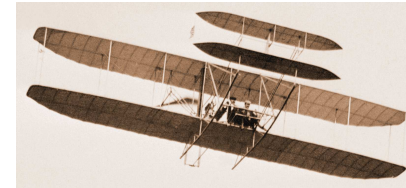
Predictable performance, not just good average case

Current	Alternative
Caches	Scratchpads
Pipelines	Thread-interleaved pipelines
Function-only ISAs	ISAs with timing
Function-only languages	Languages with timing
Best-effort communication	Fixed-latency communication
Time-sharing	Multiple independent processors

Application Areas

Hard real-time systems

- Avionics
- Automotive
- Multimedia
- Consumer Electronics
- Simple digital hardware



Basic Idea

Q: How do you make software run at a precise speed?



Basic Idea

Q: How do you make software run at a precise speed?

A: Give it access to a clock.



One Usual Way: Timers

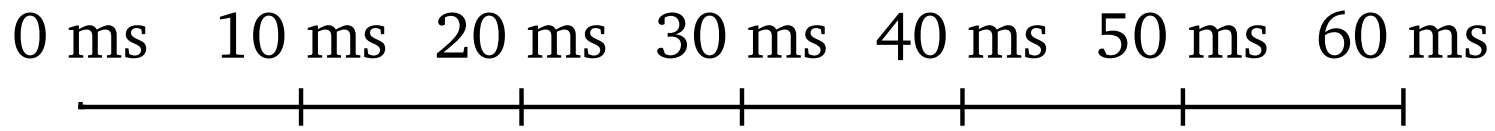
Period timer interrupt triggers scheduler

Large period reduces overhead

Linux uses a **10 ms** clock

Result: OS provides 10 ms resolution at best

Higher precision requires more overhead



Or NOPs/cycle counting

Code from Linux arch/i386/kernel/timers/timer_none.c

delay_none:

```
0:  push    %ebp
1:  mov     %esp,%ebp
3:  sub     $0x4,%esp
6:  mov     0x8(%ebp),%eax
9:  jmp     10
10: jmp     20
20: dec     %eax
21: jns     20
23: mov     %eax,-4(%ebp)
26: leave
27: ret
```

Tricky

Clock speed + cache behavior
+ branch behavior + ?

This example worries about
cache alignment

Very much an
assembly-language trick

1000s of lines of code in
Linux needed for busy wait

Related Work: Giotto

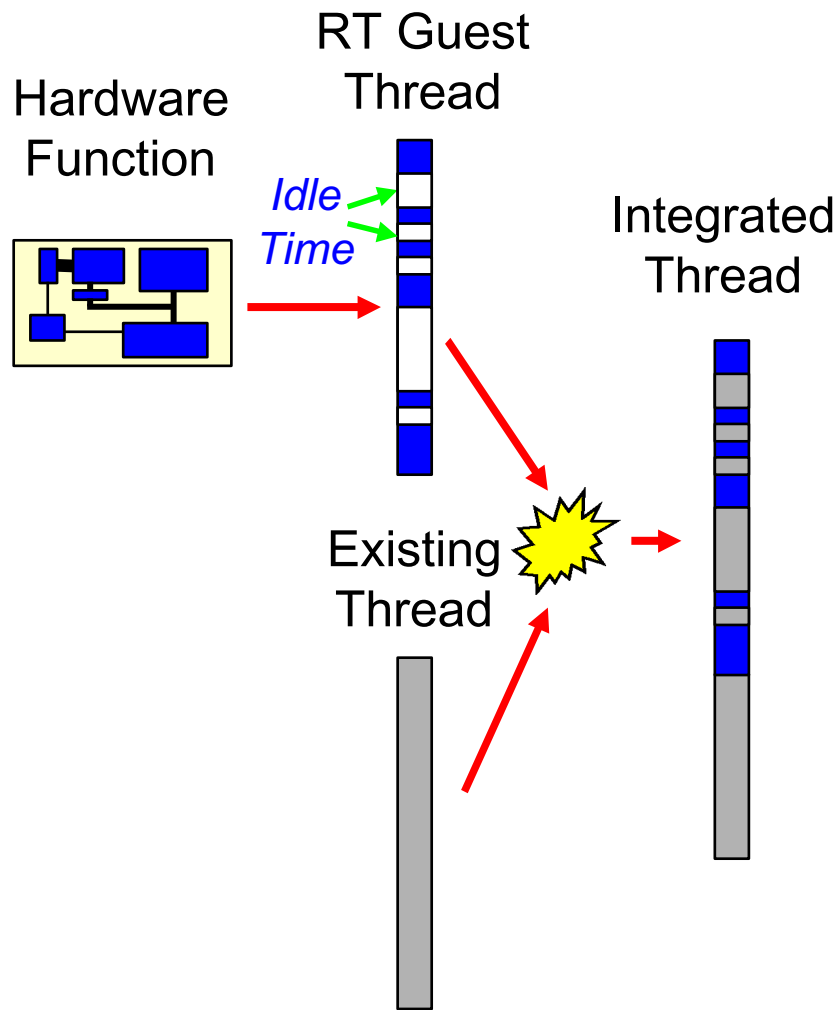
Giotto [Henzinger, Horowitz, Kirsch: Proc. IEEE 2003]

The RTOS style: specify a collection of tasks and modes.
Compiler produces schedule (task priorities).

Precision limited by periodic timer interrupt.

```
mode forward() period 200 {
  actfreq 1 do leftJet(leftMotor);
  actfreq 1 do rightJet(rightMotor);
  exitfreq 1 do point(goPoint);
  exitfreq 1 do idle(goIdle);
  exitfreq 1 do rotate(goRotate);
  taskfreq 2 do errorTask(getPos);
  taskfreq 1 do forwardTask(getErr);
}
```

Related Work: STI



Software Thread Integration [Dean: RTSS 1998]

Insert code for a non-real-time thread into a real-time thread.

Pad the rest with NOPs

Often creates code explosion

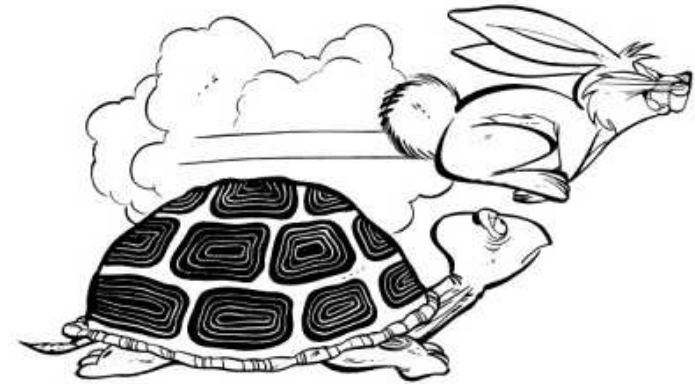
Requires PRET processors; he uses AVRs

Related Work: VISA

VISA [Meuller et al.: ISCA 2003]

Run two processors:

- Slow and predictable
- Fast and unpredictable



Start tasks on both.

If fast completes first, use extra time.

If fast misses a checkpoint, switch over to slow.

A First Attempt

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

One additional “deadline” instruction:

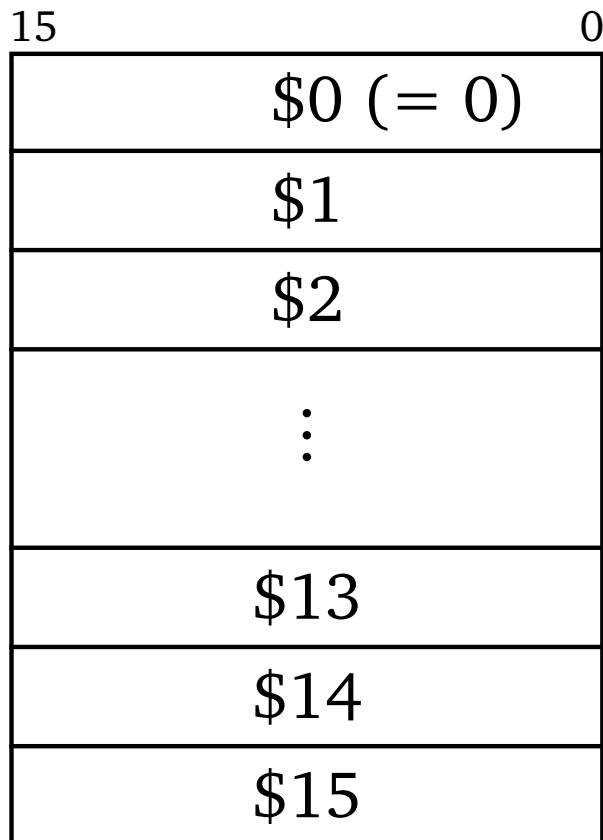
dead timer, timeout

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

Nicholas Ip and Stephen A. Edwards, “A Processor Extension for Cycle-Accurate Real-Time Software,” Proceedings of EUC, Seoul, Korea, August 2006.

Programmer's Model

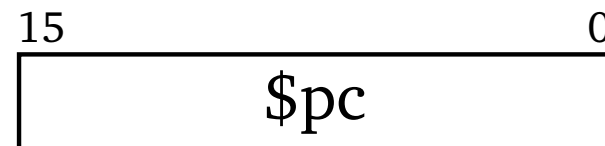
General-purpose Registers



Timers



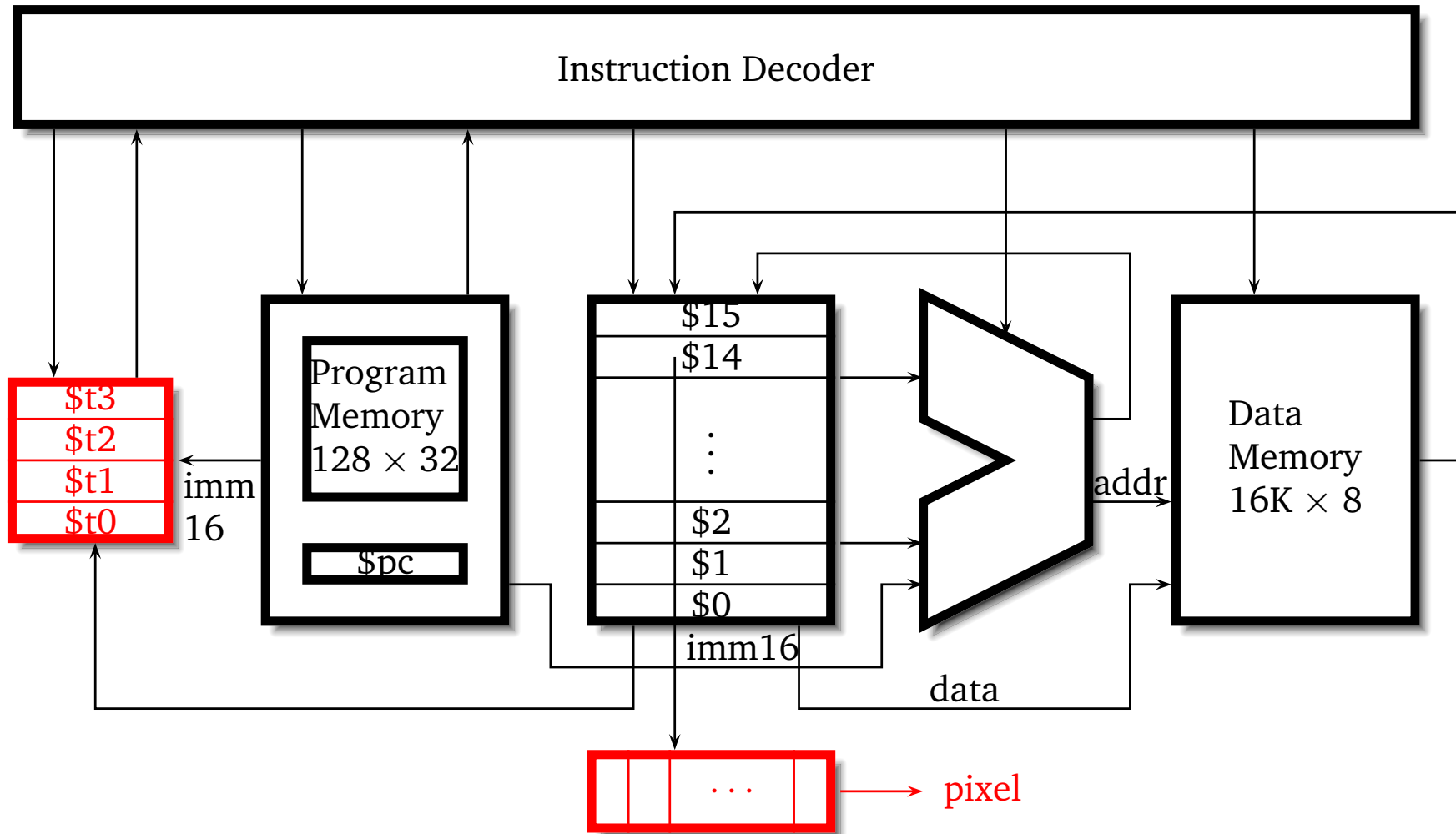
Program counter



Instructions

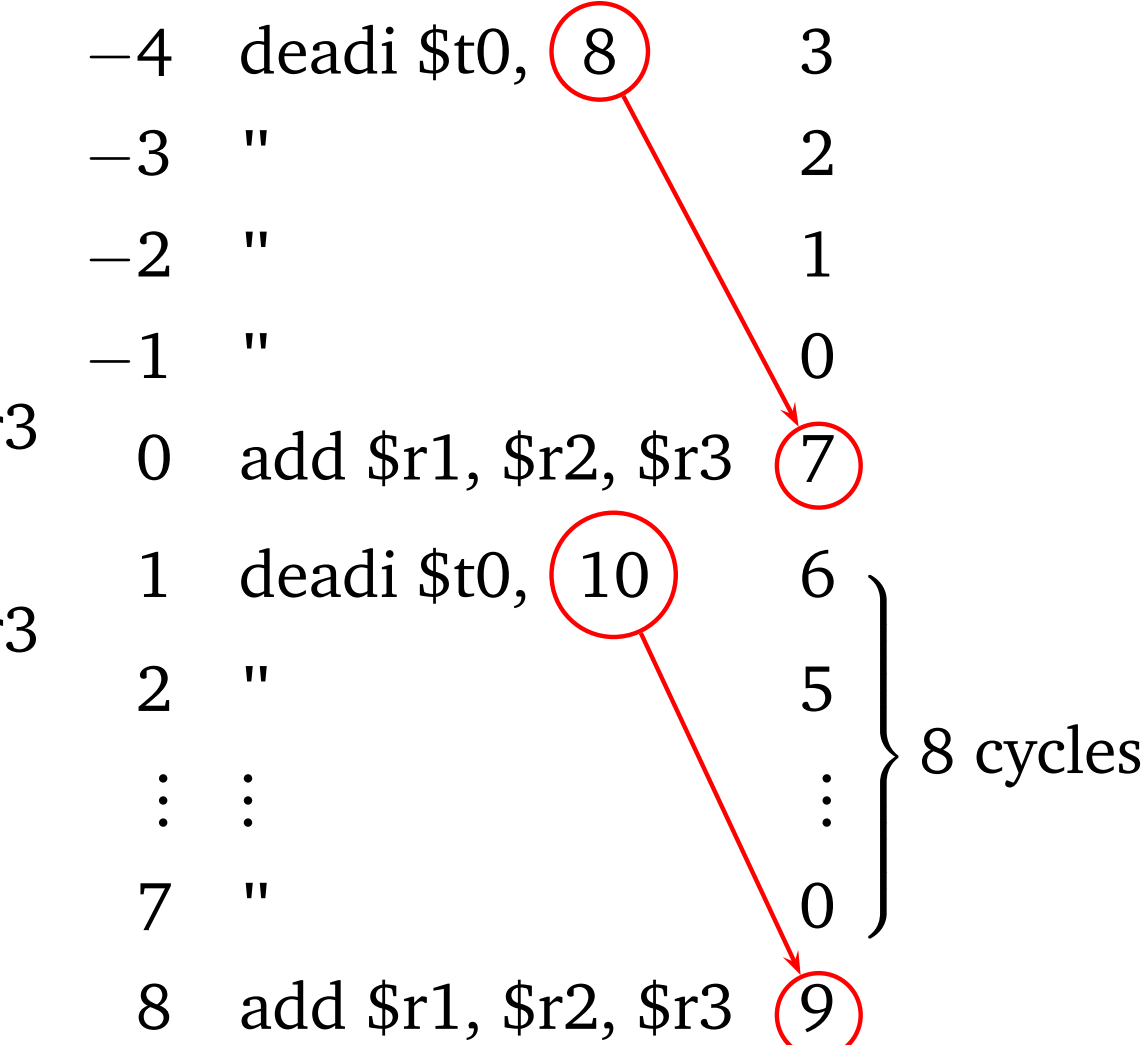
add	Rd, Rs, Rt	or	Rd, Rs, Rt
addi	Rd, Rs, imm16	ori	Rd, Rs, imm16
and	Rd, Rs, Rt	sb	Rd, (Rt + Rs)
andi	Rd, Rs, imm16	sbi	Rd, (Rs + offset)
be	Rd, Rs, offset	sll	Rd, Rs, Rt
bne	Rd, Rs, offset	slli	Rd, Rs, imm16
j	target	srl	Rd, Rs, Rt
lb	Rd, (Rt + Rs)	srli	Rd, Rs, imm16
lbi	Rd, (Rs + offset)	sub	Rd, Rs, Rt
mov	Rd, Rs	subi	Rd, Rs, imm16
movi	Rd, imm16	dead	T, Rs
nand	Rd, Rs, Rt	deadi	T, imm16
nandi	Rd, Rs, imm16	xnor	Rd, Rs, Rt
nop		xnori	Rd, Rs, imm16
nor	Rd, Rs, Rt	xor	Rd, Rs, Rt
nori	Rd, Rs, imm16	xori	Rd, Rs, imm16

Architecture



Behavior of *Dead*

	cycle	instruction	\$t0
	-4	deadi \$t0, 8	3
	-3	"	2
	-2	"	1
deadi	-1	"	0
add	0	add \$r1, \$r2, \$r3	7
deadi	1	deadi \$t0, 10	6
add	2	"	5
	⋮	⋮	⋮
	7	"	0
	8	add \$r1, \$r2, \$r3	9



8 cycles

Idioms: Straightline Code

deadi \$t0, 42

⋮

deadi \$t0, 58

⋮

deadi \$t0, 100

← First block will take
at least 42 cycles.

← Second block: at
least 58 cycles.

Idioms: Loops

L1:

⋮

deadi \$t0, 42 ← Put a deadline in a loop:

⋮

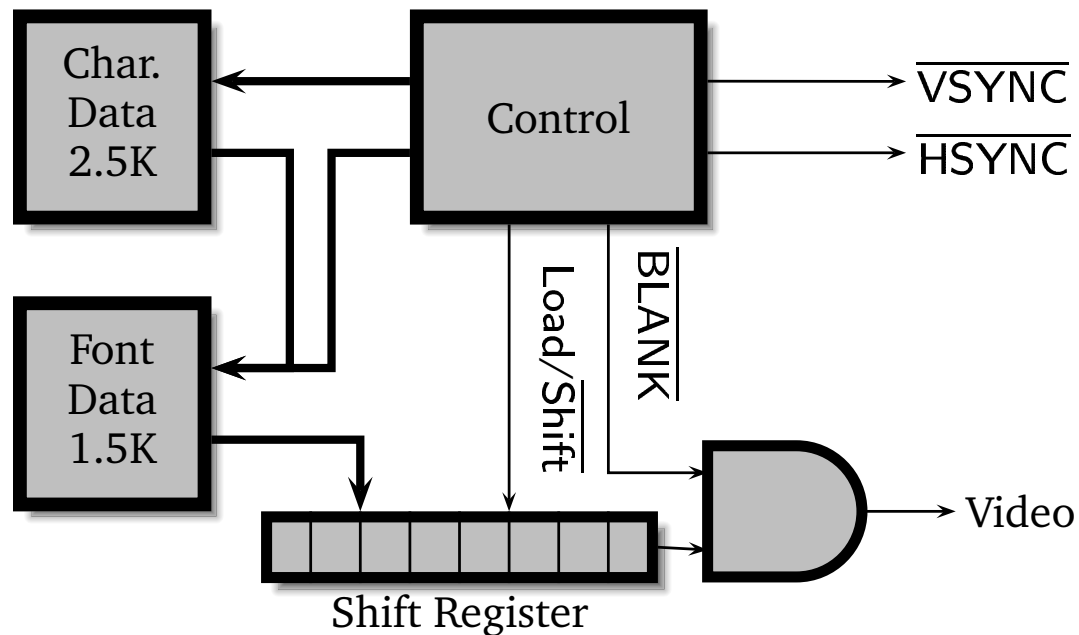
bne \$r1, \$r2, L1 Each iteration will take at least 42 cycles.

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         ; font base address
    deadi   $t1, 48              ; back porch period
    movi    $14, HB
    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

```
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
```

Sampling rate under
software control

```
wait_for_start:
  bne $15, $0, wait_for_start
```

```
got_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
```

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.

Implementation

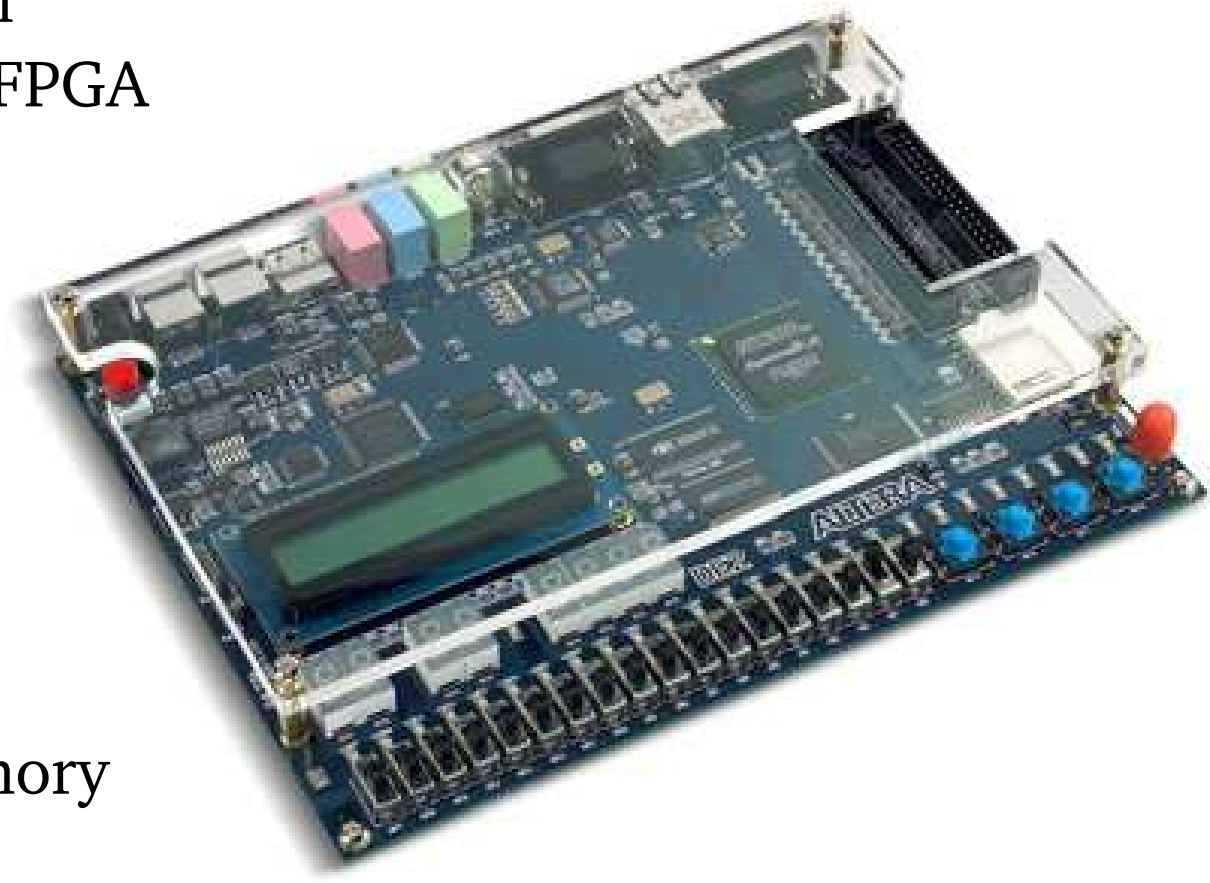
Synthesized on an
Altera Cyclone II FPGA
(DE2 board)

Coded in VHDL

Runs at 50 MHz

Unpipelined

Uses on-chip memory



Conclusions

- Embedded applications need timing control
- RTOSes on modern processors too unpredictable
- We need hardware support
- High-performance processors with predictable timing
- **Predictable performance our mantra**
- A first cut: MIPS-like processor with timers
- 50 MHz on an Altera Cyclone II FPGA
- *Dead* instruction waits for timeout, then reloads
- Video controller 1/5 the size of VHDL
- Serial controller even simpler