Compiling with Time and Space Constraints

Jens Palsberg

Purdue University

Department of Computer Science

Secure Software Systems Group

http://www.cs.purdue.edu/people/palsberg

Supported by NSF CAREER and ITR awards, and DARPA.

Acknowledgments

Ph.D. students: Dennis Brylow, Ma Di, **Mayur Naik**, Krishna Nandivada, Tian Zhao.

Undergraduate students: James Rose, Vidyut Samanta, Matthew Wallace.

Visitor: Niels Damgaard (University of Aarhus, Denmark).

Secure Software Systems Group

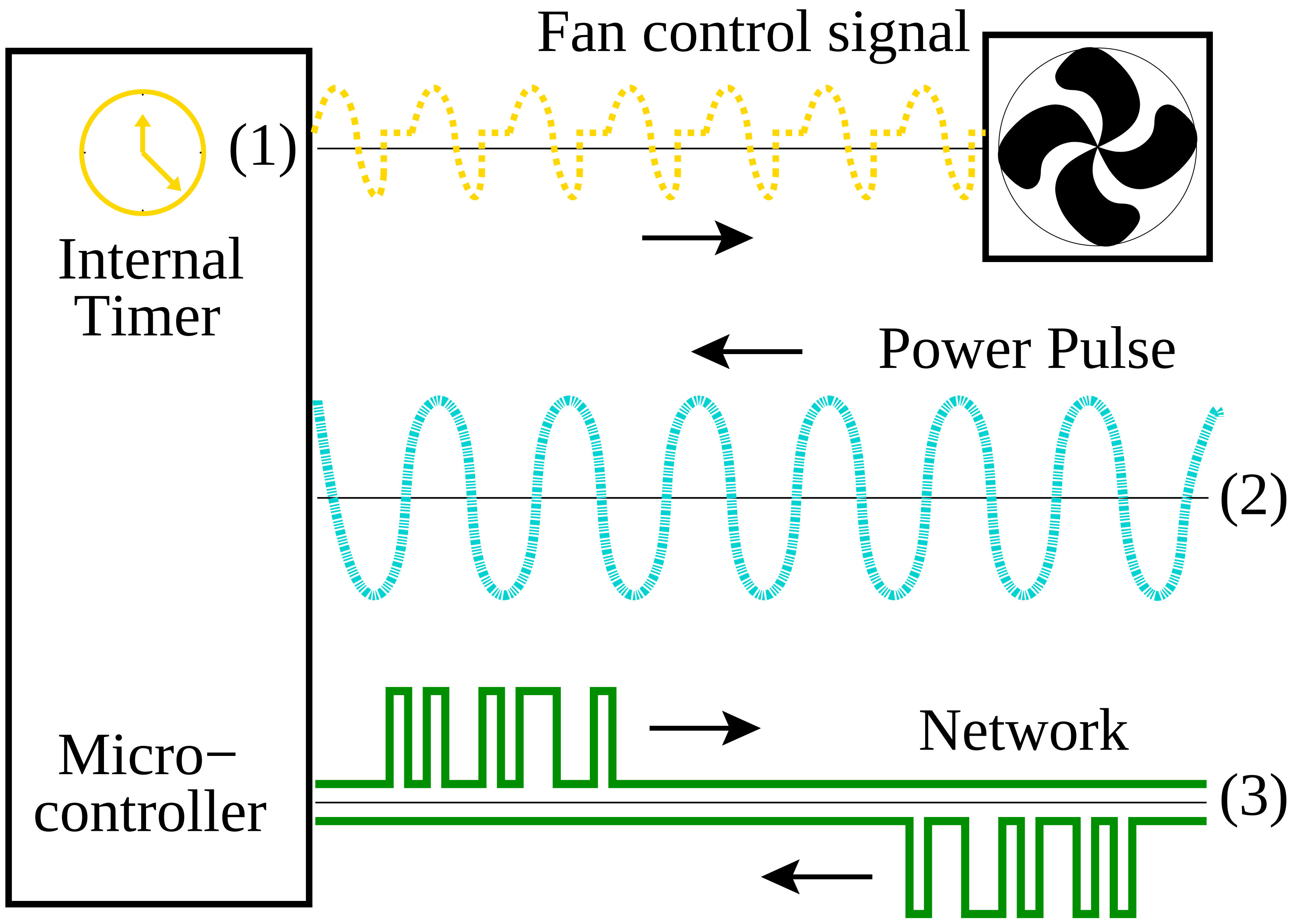
Department of Computer Science

Faculty: Antony Hosking, Jens Palsberg, Jan Vitek

Students: 15 Ph.D., 2 M.S., 8 undergraduate

Current Support:

NSF	DARPA	Lockheed Martin
– 2 CAREER awards	CERIAS	Microsoft
– 2 ITR awards	IBM	Motorola
– regular awards	Intel	Sun Microsystems


Fan control signal
(1)
Internal Timer
Power Pulse
(2)
Network
(3)
Micro–controller

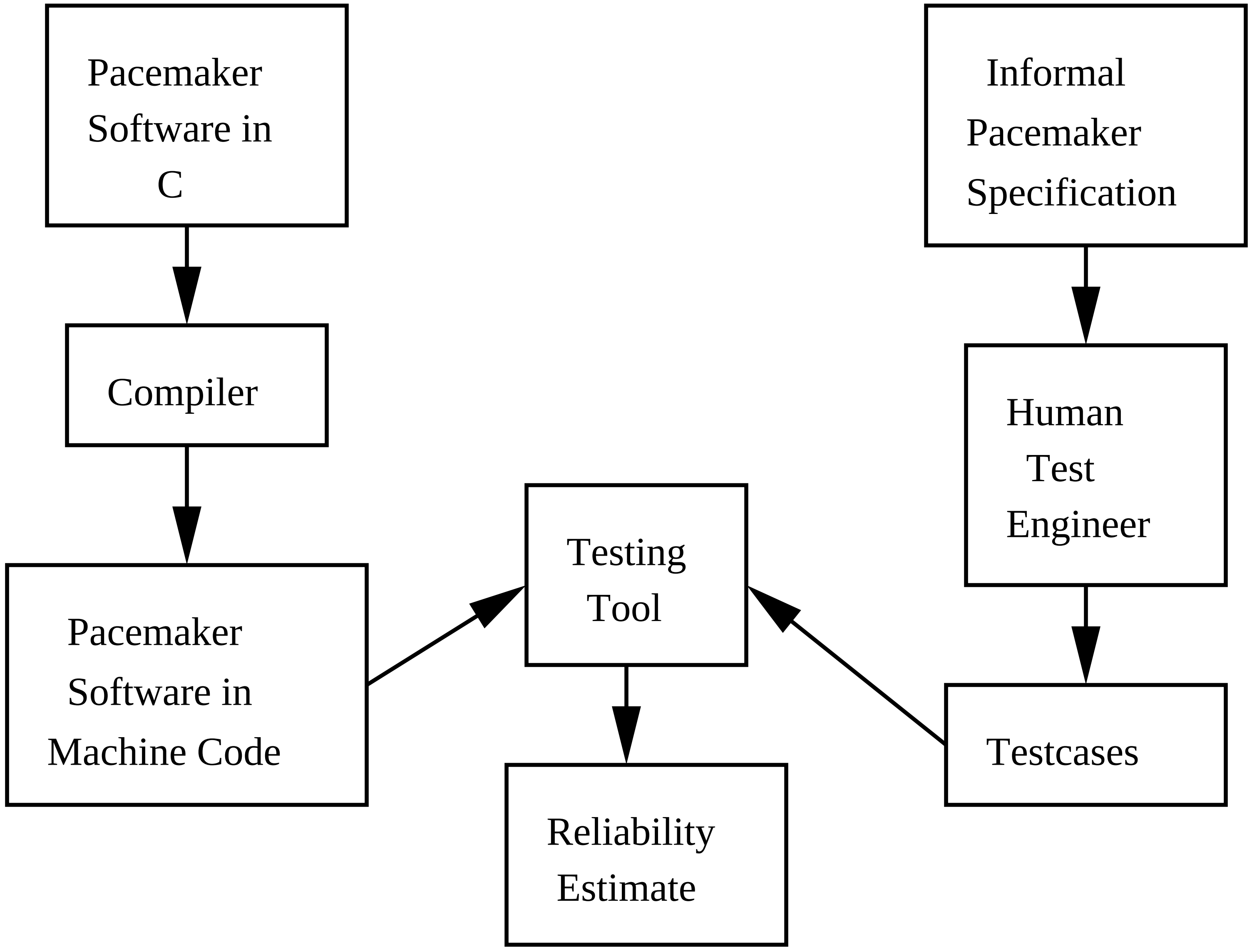
Example Program in Z86 Assembly Language

```
; Constant Pool (Symbol Table); Bit Flags for IMR and IRQ.
IRQ0 .EQU  #00000001b
; Bit Flags for external devices on Port 0 and Port 3.
DEV2 .EQU  #00010000b

; Interrupt Vectors.
     .ORG  %00h
     .WORD #HANDLER   ; Device 0

; Main Program Code.
     .ORG  0Ch
  INIT:               ; Initialization section.
0C   LD   SPL, #0F0h ; Initialize Stack Pointer.
0F   LD   RP,  #10h  ; Work in register bank 1.
12   LD   P2M, #00h  ; Set Port 2 lines to all outputs.
15   LD   IRQ, #00h  ; Clear IRQ.
18   LD   IMR, #IRQ0
1B   EI               ; Enable Interrupt 0.
```

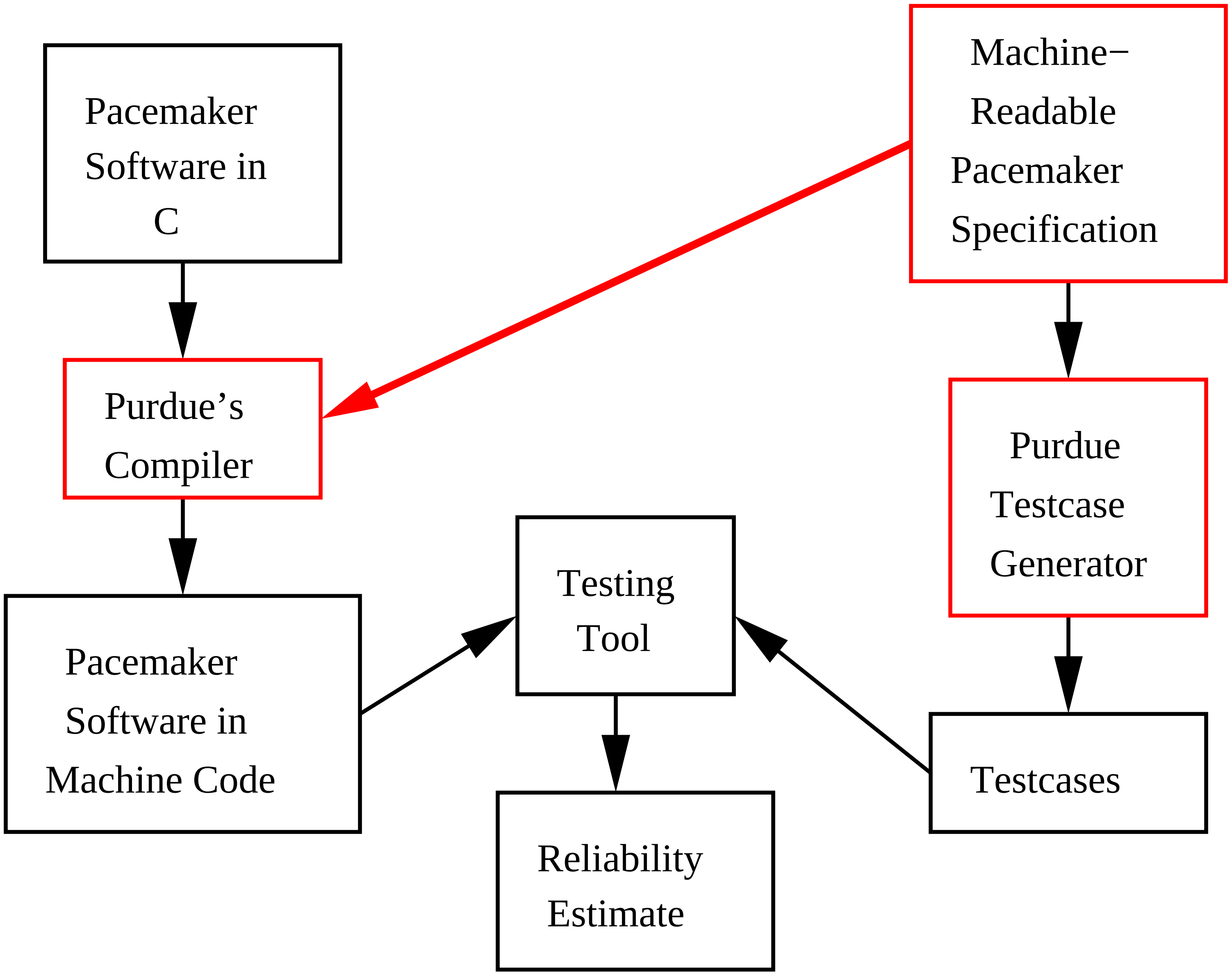
Example Program in Z86 Assembly Language

```
  START:                ; Start of main program loop.
1C    DJNZ r2,   START ; If our counter expires,
1E    LD   r1,   P3    ; send this sensor's reading
20    CALL SEND        ; to the output device.
23    JP   START

  SEND:                ; Send Data to Device 2.
26    PUSH IMR         ; Remember what IMR was.
  DELAY:
28    DI               ; Musn't be interrupted during pulse.
29    LD   P0,   #DEV2 ; Select control line for Device 2.
2C    DJNZ r3,   DELAY ; Short delay.
2E    CLR  P0
30    POP  IMR         ; Reactivate interrupts.
32    RET

  HANDLER:             ; Interrupt for Device 0.
33    LD   r2,   #00h  ; Reset counter in main loop.
35    CALL SEND
38    IRET             ; Interrupt Handler is done.
  .END
```

Current Practice



Our Goal


Pacemaker Software in C
Machine−Readable Pacemaker Specification
Purdue's Compiler
Purdue Testcase Generator
Testing Tool
Pacemaker Software in Machine Code
Testcases
Reliability Estimate

Embedded Software of the Future

A machine readable specification and an implementation:

Resoure Constraints:

– Available code space: 512 KB

– Maximum stack size: 800 bytes

– Maximum time to handle event 1: 400 μs

– Minimum battery life time: 2 years

Source Code:

// in a high-level language such as C

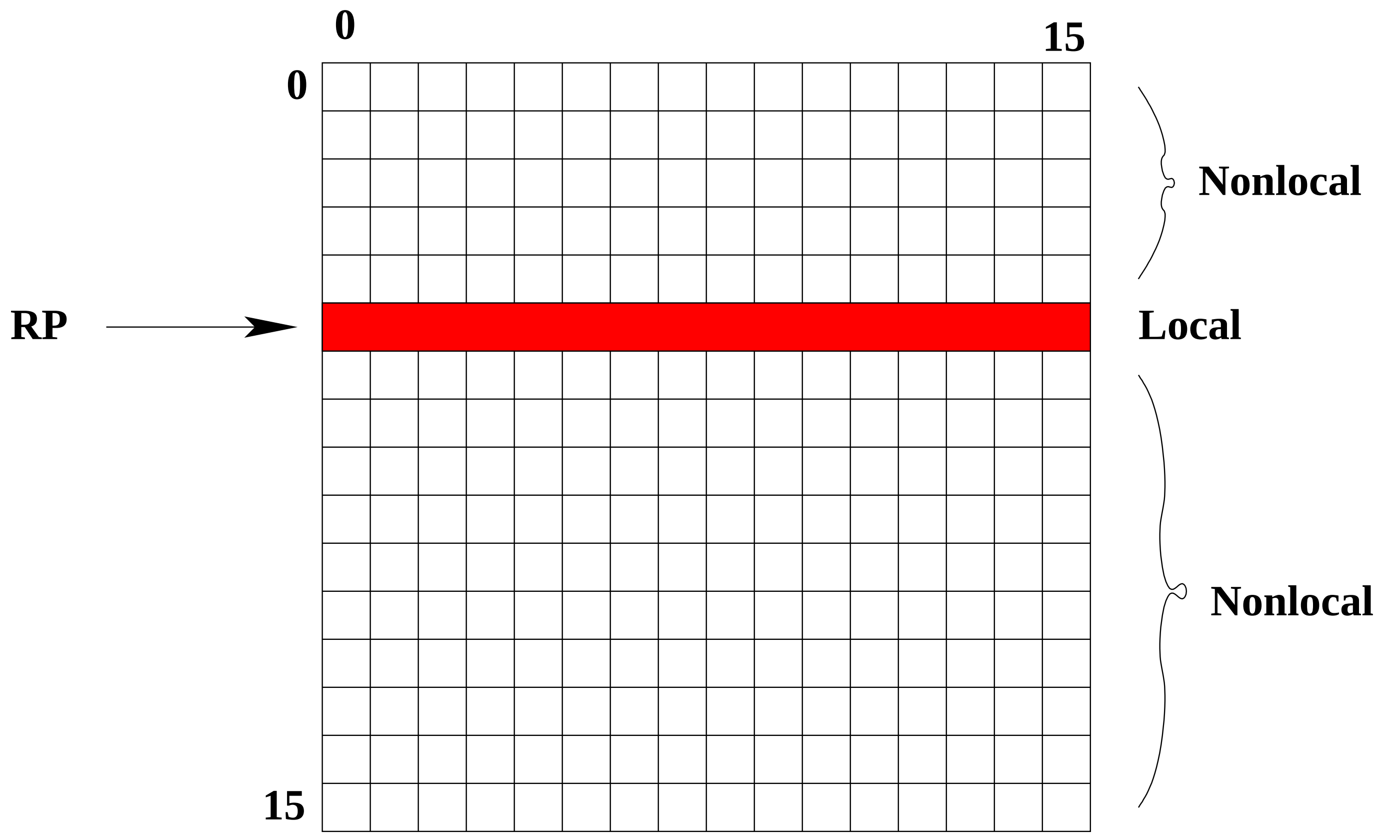
Can be compiled by a resource-aware compiler.

The generated assembly code can be verified by a model checker.

Good data layout can significantly reduce code size

Authors	Architecture	Good data layout increases the opportunities for using:
• Liao, Devadas, Keutzer, Tjiang, and Wang (1996) • Leupers and Marwedel (1996) • Rao and Pande (1999)	contemporary digital signal processor	auto-incr./auto-decr. addressing modes
• Sudarsanam and Malik (2000)	two memory units	parallel data access modes
• Sjödin and von Platen (2001)	multiple address spaces	pointer addressing modes
• Park, Lee, and Moon (2001)	two register banks	RP-relative addressing
• Our work (2002)	multiple register banks	RP-relative addressing

Data Locality is good for Time and Space



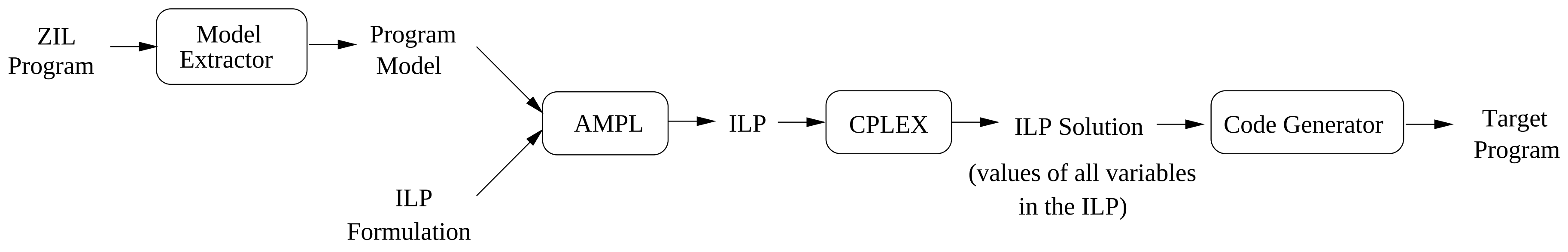
ADD r1, r2	if r1 and r2 are local:	2 bytes;	6 cycles;
	if not:	3 bytes;	10 cycles.

Our Infrastructure

$$\begin{tikzcd}
\text{C} \arrow[dr] & & \text{Java} \arrow[dl] \\
& \text{ZIL} \arrow[d, "\text{space–aware compiler}"] & \\
& \text{Z86} &
\end{tikzcd}$$

Size (in bytes) of:	`serial`	`cturk`
handwritten Z86 code	415	1789
Z86 code generated using our compiler	382	1811

Our Space-aware Compiler



– can generate **set RP** instructions anywhere
– handles interrupts
– does whole-program register allocation
– can generate code for saving **RP** on the stack:

push RP
set RP . . .
. . .
pop RP

Related Work on ILP-based Compilation

Authors	Task	Architecture
• Avissar, Barua, and Stewart (2001)	data layout	heterogeneous memory modules
• Appel and George (2001)	register allocation	Pentium
• Kong and Wilken (1998)	register allocation	irregular register architectures (IA-32)
• Stoutchinin (1997) • Ruttenberg, Gao, Stoutchinin, Lichtenstein (1996)	register allocation + software pipelining	MIPS R8000
• Goodwin and Wilken (1996)	register allocation	uniform register architectures

A ZIL Program

```
Main {
  int x0, x1, x2, x3



  LD x0, 00h
  LD x1, 01h
  CALL Foo
  LD x2, 02h
  LD x3, 03h
}
```

```
Foo() {
  int y0, y1, y2, y3



  LD y0, 00h
  LD y1, 02h

  LD y2, 02h
  LD y3, 03h

  RET
}
```

```
Handler() {
  int z0, z1, z2, z3



  LD z0, 00h
  LD z1, 01h

  LD z2, 02h
  LD z3, 03h

  IRET
}
```

From ZIL to Z86 by Hand

```
Main {
  int x0, x1, x2, x3


  SRP 0
  LD x0, 00h
  LD x1, 01h
  CALL Foo
  LD x2, 02h
  LD x3, 03h
}
```

```
Foo() {
  int y0, y1, y2, y3


  SRP 1
  LD y0, 00h
  LD y1, 02h

  LD y2, 02h
  LD y3, 03h
  SRP 0
  RET
}
```

```
Handler() {
  int z0, z1, z2, z3

  PUSH RP
  SRP 2
  LD z0, 00h
  LD z1, 01h

  LD z2, 02h
  LD z3, 03h
  POP RP
  IRET
}
```

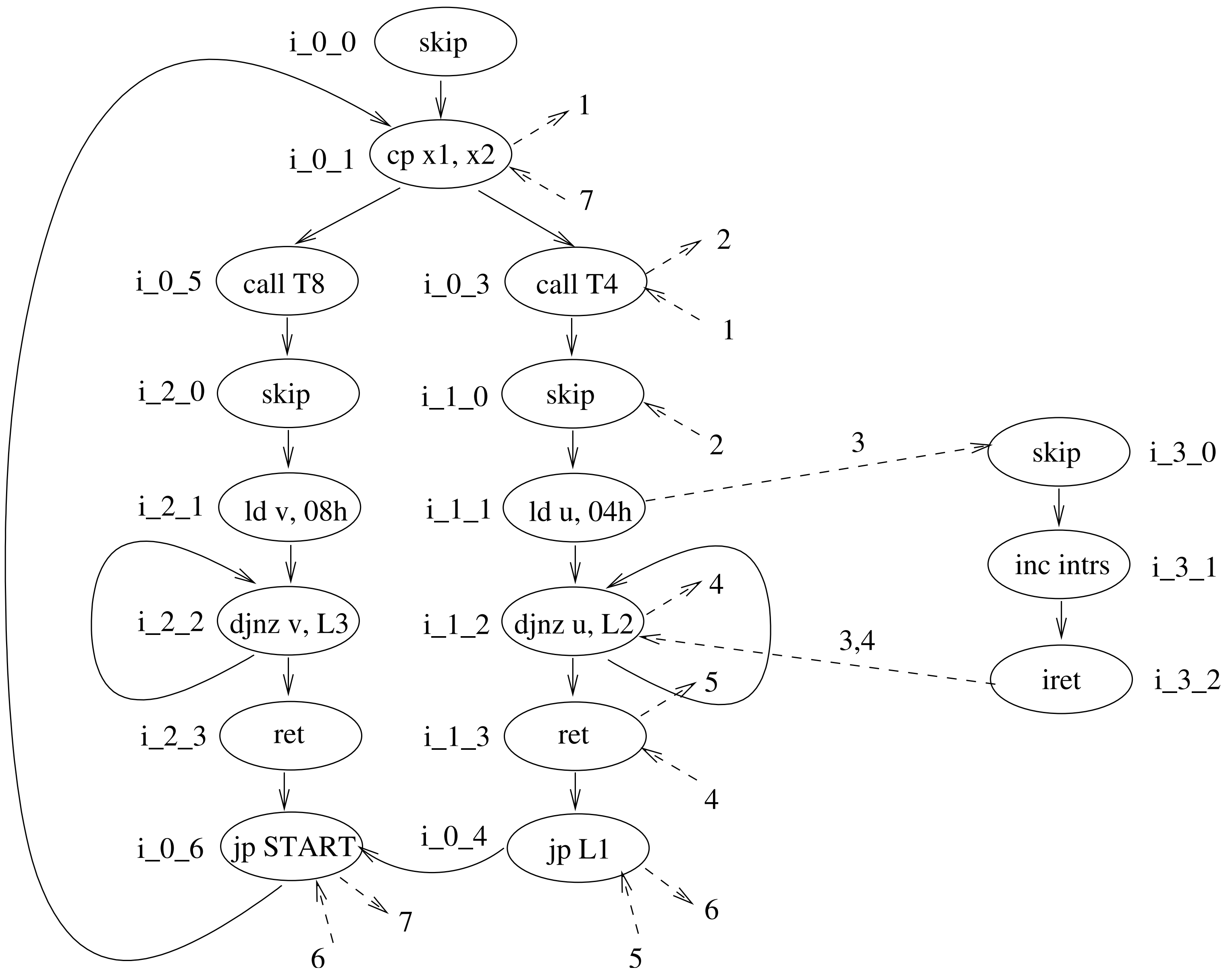
Compiling ZIL with our Space-aware Compiler

```
Main {                  Foo() {                Handler() {
  int x0, x1, x2, x3      int y0, y1, y2, y3     int z0, z1, z2, z3

                                                 PUSH RP
  SRP 0                                          SRP 2
  LD x0, 00h              LD y0, 00h             LD z0, 00h
  LD x1, 01h              LD y1, 02h             LD z1, 01h
  CALL Foo                SRP 1
  LD x2, 02h              LD y2, 02h             LD z2, 02h
  LD x3, 03h              LD y3, 03h             LD z3, 03h
}                                                POP RP
                          RET                    IRET
                        }                      }
```

Another ZIL Program

```
int intrs               PROCEDURES              HANDLERS

MAIN                    T4()                    INTR()
{                       {                       {
        int   x                 int   u                 inc   intrs
        int   y                 ld    u, 04h            iret
START:                  L2:     djnz  u, L2     }
        cp    x, y              ret
        jp    eq, L0    }
        call  T4
        jp    L1        T8()
L0:     call  T8        {
L1:     jp    START             int   v
                                ld    v, 08h
}                       L3:     djnz  v, L3
                                ret
                        }
```

Control-flow Graph for the ZIL Program



ILP Formulation in AMPL format (Excerpt)

```
# Set Declarations

set Banks := { 0, ..., 12, 15 };   # 13, 14 reserved for stack
set Instr_abs;
set Var;
set Bin2Instr within (Instr_abs × Var × Var);
set DjnzInstr within (Instr_abs × Var);

# Variables

var r { Var × Banks } binary;
var RPVal { Instr_abs × Banks } binary;
var Bin2Cost { Bin2Instr } binary;
var SetRP { Instr_abs } binary;

# Objective Function

minimize SPACE_COST:

    sum { (i, v1, v2) in Bin2Instr } Bin2Cost[i, v1, v2] +
    sum { i in Instr_abs } 2 * SetRP[i] + ...;

# Constraints

subject to VAR_IN_SINGLE_BANK { v in Var }:
    sum { b in Banks } r[v, b] = 1;
subject to RP_UNIQUE { i in Instr_abs }:
    sum { b in Banks } RPVal[i, b] = 1;
subject to DJNZ_RESTRICTION { (i, v) in DjnzInstr }:
    r[v, b] = RPVal[i, b];
```

ILP for the ZIL program

minimize

$$\mathtt{Bin2Cost\_i\_0\_1} + 2*\mathtt{SetRP\_i\_0\_0} + 2*\mathtt{SetRP\_i\_0\_1} + \ldots$$

subject to

$$\begin{aligned}
&\mathtt{r\_u\_0} + \mathtt{r\_u\_1} + \ldots + \mathtt{r\_u\_15} = 1\\
&\mathtt{r\_v\_1} + \mathtt{r\_v\_1} + \ldots + \mathtt{r\_v\_15} = 1\\
&\ldots\\
&\mathtt{RPVal\_i\_0\_0\_0} + \mathtt{RPVal\_i\_0\_0\_1} + \ldots + \mathtt{RPVal\_i\_0\_0\_15} = 1\\
&\mathtt{RPVal\_i\_0\_1\_0} + \mathtt{RPVal\_i\_0\_1\_1} + \ldots + \mathtt{RPVal\_i\_0\_1\_15} = 1\\
&\ldots\\
&\mathtt{r\_u\_0} = \mathtt{RPVal\_i\_1\_2\_0}\\
&\mathtt{r\_u\_1} = \mathtt{RPVal\_i\_1\_2\_1}\\
&\ldots\\
&\mathtt{r\_u\_15} = \mathtt{RPVal\_i\_1\_2\_15}\\
&\mathtt{r\_u\_0} = \mathtt{RPVal\_i\_2\_2\_0}\\
&\mathtt{r\_v\_1} = \mathtt{RPVal\_i\_2\_2\_1}\\
&\ldots\\
&\mathtt{r\_v\_15} = \mathtt{RPVal\_i\_2\_2\_15}
\end{aligned}$$

Target Z86 code for the ZIL program

```
MAIN
{
        ; x, y are allotted
        ; regs 0, 1 in bank 1

START:
        srp   1
        cp    r0, r1  ; 6b saved
        jp    eq, L0
        call  T4
        jp    L1
L0:     call  T8
L1:     jp    START
}
```

```
PROCEDURES

T4()
{
        ; u is allotted register
        ; 0 in bank 2

        srp   2
        ld    r0, 04h  ; 6b saved
L2:     djnz  r0, L2
        ret
}

T8()
{
        ; v is allotted register
        ; 1 in bank 2

        srp   2
        ld    r1, 08h  ; 6b saved
L3:     djnz  r1, L3
        ret
}
```

```
HANDLERS

INTR()
{
        ; intrs is allotted register
        ; 0 in bank 3

        push  RP
        srp   3
        inc   r0  ; 6b saved
        pop   RP
        iret
}
```

ILP Constraints

$$\begin{aligned}
V &= 0 \\
V &= 1 \\
V &= V' \\
V &\leq V' \\
V_1 + \ldots + V_n &= 1 \\
V_1 + \ldots + V_n &\leq C \\
V &\leq V' + V''
\end{aligned}$$

where $V, V', V'', V_1, \ldots, V_n$ are variables that range over $\{0, 1\}$.

Solvability is NP-complete.

A Cheap ILP-based Approach

Approach: Just one **set RP**, at the start of the program.

Variables: Bin2Cost, InCurrBank.

Idea: InCurrBank_v = 1 if we should store v in the bank to which RP points.

$$\sum_{v \in \texttt{Var}} \text{InCurrBank}_v \leq 16$$
$$\forall v_1 \in \texttt{PDV0}.\ \ \forall v_2 \in \texttt{PDV15}.\ \text{InCurrBank}_{v_1} + \text{InCurrBank}_{v_2} \leq 1$$
$$\forall v_1 \in \texttt{PDV0}.\ \ \forall v_2 \in \texttt{PDV0}.\ \ \text{InCurrBank}_{v_1} = \text{InCurrBank}_{v_2}$$
$$\forall v_1 \in \texttt{PDV15}.\ \forall v_2 \in \texttt{PDV15}.\ \text{InCurrBank}_{v_1} = \text{InCurrBank}_{v_2}$$
$$\forall (i,v) \in \texttt{DjnzInstr}.\ \text{InCurrBank}_v = 1$$
$$\forall (i,v_1,v_2) \in \texttt{Bin2Instr}.\ \text{Bin2Cost}_i + \text{InCurrBank}_{v_1} \geq 1$$
$$\forall (i,v_1,v_2) \in \texttt{Bin2Instr}.\ \text{Bin2Cost}_i + \text{InCurrBank}_{v_2} \geq 1$$

minimize: $$\sum_{(i,v_1,v_2) \in \texttt{Bin2Instr}} \text{Bin2Cost}_i \; - \sum_{(i,v) \in \texttt{Bin1OrIncrInstr}} \text{InCurrBank}_v$$

Benchmark Characteristics

Number of:	`ex`	`serial`	`cturk`
Lines of ZIL (nodes in CFG)	14	181	850
Lines of ZIL after abstraction (nodes in CFG_{abs})	17	53	304
Edges in CFG_{abs}	42	193	1287
Instructions in `Bin2Instr`	1	9	147
Instructions in `Bin1Instr`	2	41	126
Instructions in `IncrInstr`	1	2	20
Instructions in `DjnzInstr`	2	0	10
User-defined variables	5	9	55
Procedures (excluding MAIN)	2	6	37
Interrupt handlers	1	2	2

Experimental Results: Timing

Number of:	technique	`ex`	`serial`	`cturk`
Integer variables	Cheap	23	33	187
	SetRP	401	1035	2343
	SetRP + PuPoRP	449	1275	2885
	SetRP + Full PuPoRP	617	2009	6909
Constraints	Cheap	225	239	525
	SetRP	656	3132	8900
	SetRP + PuPoRP	1052	6369	21426
	SetRP + Full PuPoRP	1722	9953	35961
Seconds to solve the ILP	Cheap	0.00	0.01	0.10
	SetRP	0.04	0.27	856
	SetRP + PuPoRP	0.07	0.92	2478
	SetRP + Full PuPoRP	0.14	3.39	59351

Experimental Results: Target Code

Number of:	technique	`ex`	`serial`	`cturk`
srp introduced	Cheap	1	1	1
	SetRP	1	2	19
	SetRP + PuPoRP	1	2	21
	SetRP + Full PuPoRP	1	2	18
push/pop RP introduced	SetRP + PuPoRP	0	0	2
	SetRP + Full PuPoRP	0	0	2
instructions addressing only working registers	Cheap	4	29	50
	SetRP	3	30	131
	SetRP + PuPoRP	4	35	150
	SetRP + Full PuPoRP	4	35	150
	upper bound	4	52	293

Size (in bytes) of:	`serial`	`cturk`
handwritten Z86 code	415	1789
Z86 code generated using SetRP + PuPoRP	382	1811

Embedded Software of the Future

A machine readable specification and an implementation:

Resoure Constraints:

– Available code space: 512 KB

– Maximum stack size: 800 bytes

– Maximum time to handle event 1: 400 μs

– Minimum battery life time: 2 years

Source Code:

```
//   in a high-level language such as C
```

Can be compiled by a resource-aware compiler.

The generated assembly code can be verified by a model checker.

A Nasty Programming Error

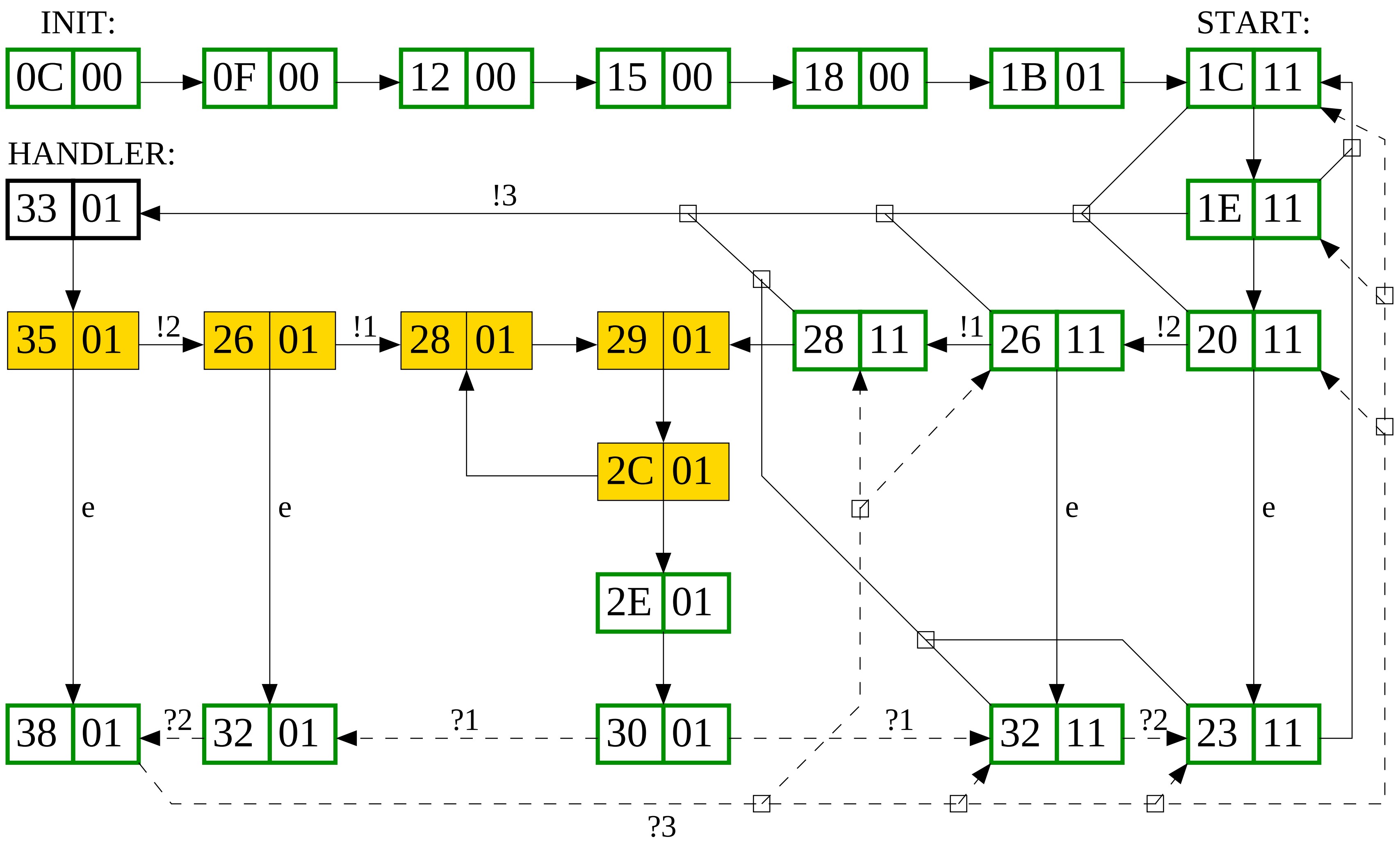
```
handler 1 {
 // do something
 enable-handling-of-interrupt-2
 // do something else
 iret
}
handler 2 {
 // do something
 enable-handling-of-interrupt-1
 // do something else
 iret
}
```

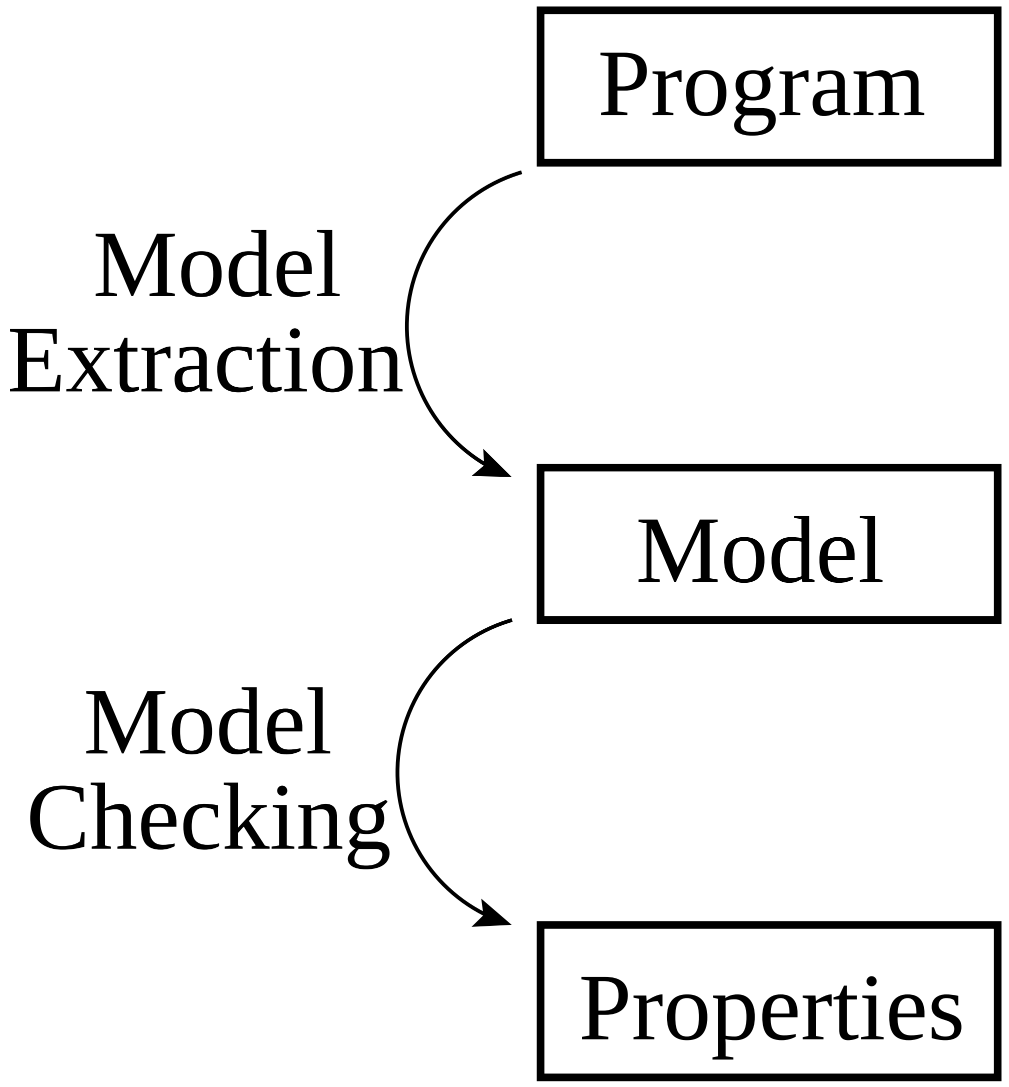
Interrupt Mask Register

Well-known product	Processor	interrupt sources	master bit
Microcontroller	Zilog Z86	6	yes
iPAQ Pocket PC	Intel strongARM, XScale	21	no
Palm	Motorola Dragonball (68K Family)	22	yes
Microcontroller	Intel MCS-51 Family (8051 etc)	6	yes

MCS–51 interrupt mask register:

EA	–	ET2	ES	ET1	EX1	ET0	EX0


INIT:
START:
HANDLER:
0C 00
0F 00
12 00
15 00
18 00
1B 01
1C 11
1E 11
33 01
!3
35 01
26 01
28 01
29 01
28 11
26 11
20 11
!2
!1
!1
!2
2C 01
2E 01
e
e
e
e
38 01
32 01
30 01
32 11
23 11
?2
?1
?1
?2
?3


Program
Model Extraction
Model
Model Checking
Properties

Stack-Size Analysis

Program	Lower	Upper	Time	Space
CTurk	17	18	4.11 s	31.6 MB
GTurk	16	17	4.31 s	32.2 MB
ZTurk	16	17	4.22 s	32.1 MB
DRop	12	14	4.14 s	31.1 MB
Rop	12	14	4.18 s	31.8 MB
Fan	11	N/A	N/A	N/A
Serial	10	10	3.87 s	31.0 MB
Example	37	37	3.21 s	34.9 MB

The lower bounds were found with a software simulator for Z86 assembly language that we wrote.

A Typed Interrupt Calculus

```
maximum stack size: 2

imr = imr or 111b

loop {
   skip
   imr = imr or 111b
}
```

```
handler 1 [ ( 111b -> 111b : 0 )
            ( 110b -> 110b : 0 )
          ] {
   skip
   iret
}
handler 2 [ ( 111b -> 111b : 1 )
          ] {
   skip
   imr = imr and 110b
   imr = imr or  100b
   iret
}
```

Theorem: A well-typed program cannot cause stack overflow.

The Interrupt Calculus

$$
\begin{array}{llll}
\text{(program)} & p & ::= & (m, \overline{h}) \\
\text{(main)} & m & ::= & \text{loop } s \mid s \; ; \; m \\
\text{(handler)} & h & ::= & \text{iret} \mid s \; ; \; h \\
\text{(statements)} & s & ::= & x = e \mid \text{imr} = \text{imr} \wedge imr \mid \text{imr} = \text{imr} \vee imr \mid \\
 & & & \text{if0 } x \text{ then } s_1 \text{ else } s_2 \mid s_1 \; ; \; s_2 \mid \text{skip} \\
\text{(expression)} & e & ::= & c \mid x \mid x + c \mid x_1 + x_2
\end{array}
$$

Conclusion

High-assurance embedded systems in high-level languages =

> machine-readable specifications +
> type systems +
> model checking +
> time-, space-, and power-aware compiler +
> automatic testcase generation.

Bibliography

http://www.cs.purdue.edu/homes/palsberg/paper/icse01.ps.gz
Static Checking of Interrupt-driven Software, with Dennis Brylow and Niels Damgaard. In Proceedings of ICSE'01, 23rd International Conference on Software Engineering, pages 47-56, Toronto, May 2001.

http://www.cs.purdue.edu/homes/palsberg/paper/naik-palsberg02.ps.gz
Compiling with Code-Size Constraints, with Mayur Naik. In Proceedings of LCTES'02, Languages, Compilers, and Tools for Embedded Systems joint with SCOPES'02, Software and Compilers for Embedded Systems, Berlin, Germany, June 2002.

http://www.cs.purdue.edu/homes/palsberg/paper/ftrtft02.ps.gz
A Typed Interrupt Calculus, with Di Ma. In FTRTFT'02, 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems, Oldenburg, Germany, September 2002.