OUTLINE OF THE TALK

I. INTRODUCTION

II. DSP HARDWARE TO REDUCE LOOP OVERHEAD

III. COMPILER SUPPORT

IV. RESULTS AND CONCLUSIONS

INTRODUCTION

(i) **Signal Processing Filters**
(e.g., FIR, IIR)

(ii) **Frequence Transformations**
(e.g., Fourier, Cosine)

(iii) **Image Processing Algorithms**
(e.g., Edge Manipulation)

Typical DSP Applications

INTRODUCTION (cont.)

Tight
Small
Loops

are quite COMMON!!

HARDWARE LOOPING SUPPORT

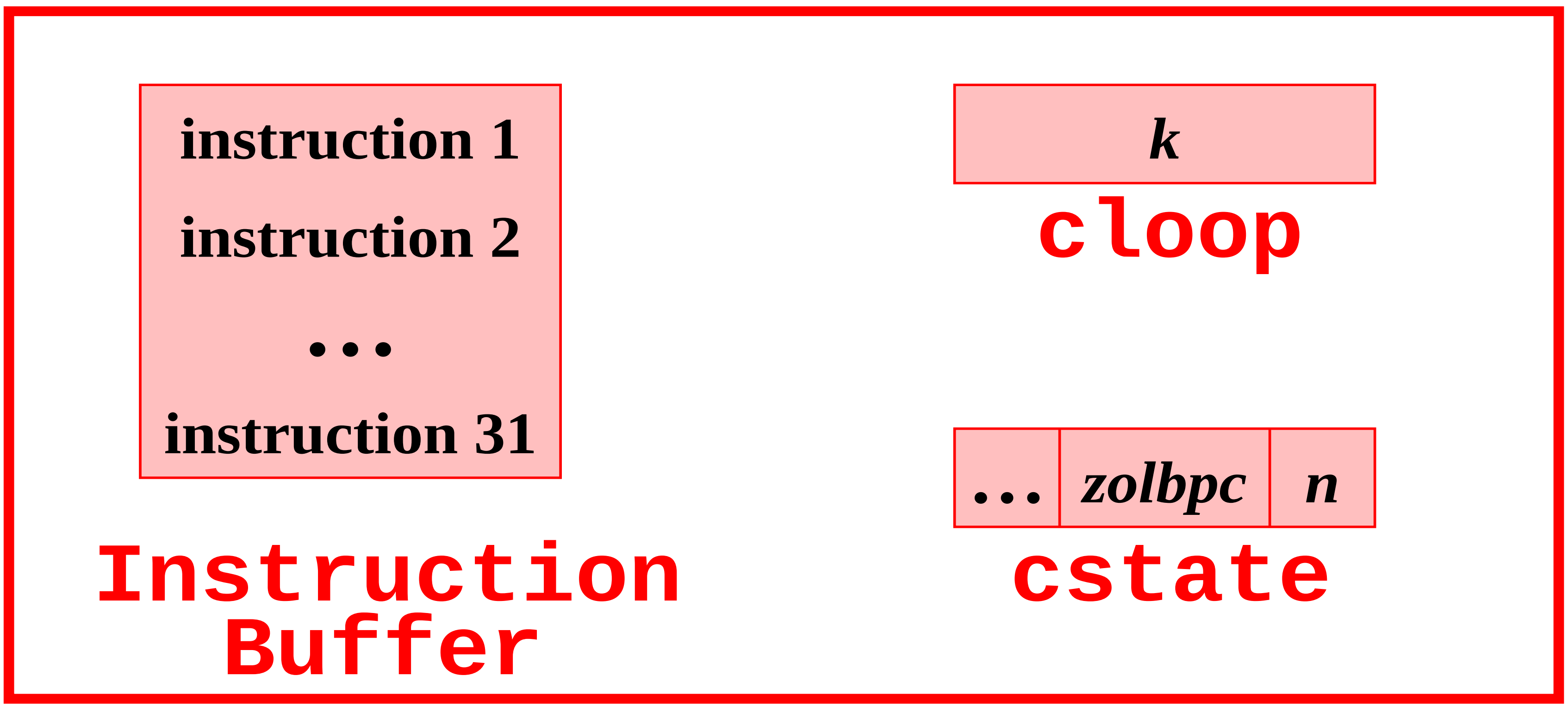
(1) Execute Fixed Set of Instructions Multiple Times

(2) Reduce Loop Branch Overhead

(3) Reduce Power Consumption

(4) Reduce Memory Bus Contention

HARDWARE LOOPING SUPPORT (cont.)



LUCENT DSP 16000
Zero Overhead Loop Buffer

ASSEMBLY SYNTAX FOR ACCESSING ZOLB (Zero Overhead Loop Buffer)

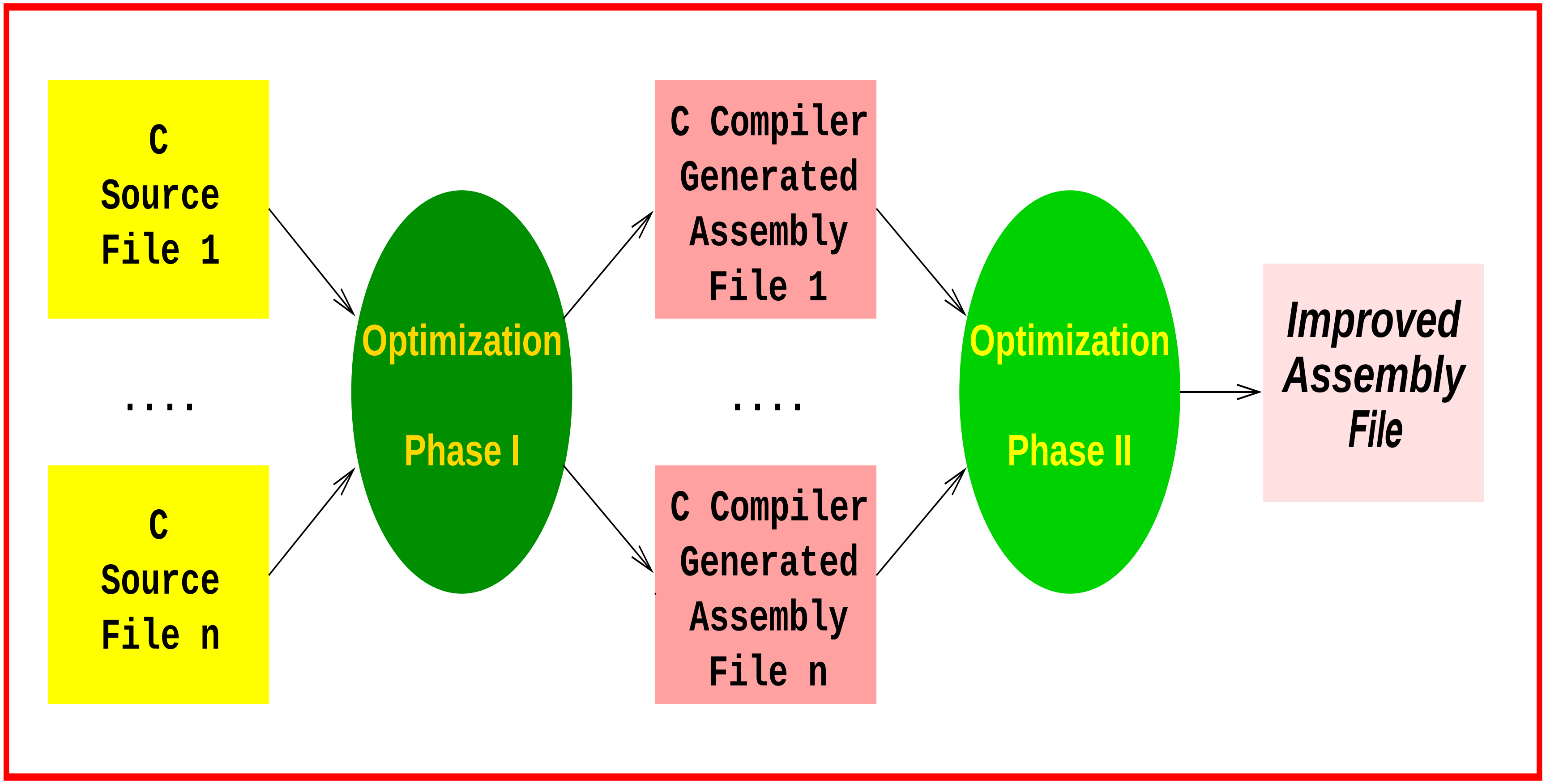
```
...
do k {
  instruction 1
  ...
  instruction n
}
...
```

using the DO Instruction

```
...
redo k
...
```

using the REDO Instruction

COMPILER SUPPORT



Overview of DSP 16K C Compiler

COMPILER SUPPORT (cont.)

(1) Conditional Instructions

(2) Inlining

(3) Loop Splitting

(4) Dealing with an Unknown Number of Loop Iterations

More Loops in the ZOLB

CONDITIONAL INSTRUCTIONS

```
for (i = 0;
     i < 1000;
     i++)

   if (a[i] > 0)
      sum += a[i];
```

Original Source

```
    r0 = _a
    a1 = -9999
L5: a0 = *r0
    a0 = a0
    if gt goto L4
    a2 = a2 + a0
L4: r0 = r0 + 2
    a1 = a1 + 1
    if le goto L5
```

without Conditional

```
    r0 = _a
    a1 = -9999
L5: a0 = *r0
    a0 = a0
    if le a2 = a2 + a0
    r0 = r0 + 2
    a1 = a1 + 1
    if le goto L5
```

with Conditional

INLINING

```
int abs(int v)
{
   if (v < 0)
      v = -v;
   return v;
}

...
sum = 0;
for (i = 0;
     i< 10000; i++)
   sum += abs(a[i]);
```

Source Code

```
_abs: a0 = a0
      if lt a0 = -a0
      return
      ...
      r4 = _a
      a5 = 0
      a4 = -9999
L5:   a0 = *r4++
      call _abs
      a5 = a5 + a0
      a4 = a4 + 1
      if le goto L5
```

Before Inlining

```
      r4 = _a
      a5 = 0
      a4 = -9999
L5:   a0 = *r4++
      a0 = a0
      if lt a0 = -a0
      a5 = a5 + a0
      a4 = a4 + 1
      if le goto L5
```

After Inlining

UNKNOWN NUMBER OF LOOP ITERATIONS

```
sum = 0;
for (i = 0;
        a[i] != n;
        i++)
    sum += a[i]*2;
```

Source Code of Loop

UNKNOWN NUMBER OF LOOP ITERATIONS (cont.)

```
     r0 = _a
     a2 = 0
     r1 = _n
     a0 = *r0
     a1 = *r1
     a0 - a1
     if eq goto L3
L5:  a0 = *r0++
     a0 = a0 <<< 1
     a2 = a2 + a0
     a0 = *r0
     a0 - a1
     if ne goto L5
L3:
```

without Using the ZOLB

```
      ...
      if eq goto L3
      cloop = 65535
      a3 = 1
      do cloop {
        a4 = *(r0+2)
        a4 - a1
        if eq cloop = a3
        a0 = *r0++
        a0 = a0 <<< 1
        a2 = a2 + a0
      }
      goto L01
L02:  cloop = 65535
      redo cloop
L01:  a4 - a1
      if ne goto L02
L3:
```

after Using the ZOLB

COMPILER SUPPORT (cont.)

(5) Extracting Increments of Basic Induction Variables

(6) Loop Collapsing

(7) Loop Interchange

Further Reducing Loop Overhead

EXTRACTING INCREMENT OF BASIC INDUCTION VARIABLES

```
cloop = 10000
r0 = _a
a2 = 0
do cloop {
 *r0++ = a2
 a1 = a1 + 1
}
```

after Using the ZOLB with a1 Live after the Loop

```
cloop = 10000
r0 = _a
a2 = 0
do cloop {
  *r0++ = a2
}
a1 = a1 + 10000
```

after Extracting the Assignment to a1

LOOP INTERCHANGE (cont.)

```
    r1 = _a
    a3 = 0
    a2 = -199
L5: r0 = r1
    a1 = -49
L9: *r0++ = a3
    a1 = a1 + 1
    if le goto L9
    r1 = r1 + 200
    a2 = a2 + 1
    if le goto L5
```

before Loop Interchange

```
    r1 = _a
    a3 = 0
    a2 = -49
L5: r0 = r1
    a1 = -199
    k = 200
L9: *r0++k = a3
    a1 = a1 + 1
    if le goto L9
    r1 = r1 + 2
    a2 = a2 + 1
    if le goto L5
```

after Loop Interchange

LOOP INTERCHANGE (cont.)

```
     r1 = _a
     a3 = 0
     a2 = -49
L5:  cloop = 200
     r0 = r1
     k = 200
     do cloop {
       *r0++k = a3
     }
     r1 = r1 + 2
     a2 = a2 + 1
     if le goto L5
```

DSP16000 Assembly after Using the ZOLB

LOOP INTERCHANGE (cont.)

```
     r1 = _a
     a3 = 0
     a2 = -49
L5:  cloop = 200
     r0 = r1
     k = 200
     do cloop {
       *r0++k = a3
     }
     r1 = r1 + 2
     a2 = a2 + 1
     if le goto L5
```

DSP16000 Assembly after Using the ZOLB

AVOIDING REDUNDANT LOAD of the ZOLB

```
extern int a[100];
extern int b[100];
extern float c[200];
extern float d[200];
...
for (i = 0;
     i < 100;
     i++)
  a[i] = b[i];
...
for (i = 0;
     i < 200;
     i++)
  c[i] = d[i];
```

Source Code of Two Different Loops

```
r1 = _a
r0 = _b
do 100 {
a0 = *r0++
*r1++ = a0
}
...
cloop = 200
r1 = _c
r0 = _d
do cloop {
a0 = *r0++
*r1++ = a0
}
```

DSP 16000 Assembly After Using the ZOLB

```
r1 = _a
r0 = _b
do 100 {
a0 = *r0++
*r1++ = a0
}
...
cloop = 200
r1 = _c
r0 = _d
redo cloop
```

DSP 16000 Assembly after Using REDO

$$\texttt{in[B]} = \begin{cases} \texttt{Null} & \texttt{if B == function entry} \\ \bigcup_{\texttt{P} \in \texttt{pred[B]}} \texttt{out[P]} & \texttt{otherwise} \end{cases} \quad (1)$$

$$\texttt{out[B]} = \begin{cases} \texttt{Null} & \texttt{if B contains a call} \\ \texttt{B} & \texttt{if B contains a ZOLB loop} \\ \texttt{in[B]} & \texttt{otherwise} \end{cases} \quad (2)$$

ORDER OF THE ANALYSIS AND TRANSFORMATION

(1) Build Call Graph

(2) Merge Blocks

(3) Find Loops

(4) Dataflow Analysis

(5) If Conversion

(6) Find Loop Invariants and Loop Induction Variables

(7) Calculate Loop Iterations

(8) Perform Inlining

(9) Memory Disambiguation

(10) Loop Splitting

(11) Loop Flattening

(12) Loop Interchange

(13) Place Loops in the ZOLB

(14) Basic Induction Variable Elimination

(15) Extract Basic Induction Variable Assignments

CONCLUSIONS

Improvements in

EXECUTION TIME

and

REDUCTIONS in CODE SIZE