SCOPES 2003

Tailoring Software Pipelining For Effective Exploitation Of Zero Overhead Loop Buffer

Gang-Ryung Uh
CS Department
Boise State University

Outline

1. Low-power DSP 16000 and ZOLB
2. Compiler Mission
3. Conventional Approach
4. Alternatative approach
5. Intermediate Results
6. Conclusion

Signal Processing Algorithm

FIR $y_k = \sum b_n x_{k-n}$, for $n = 0, \ldots, N$

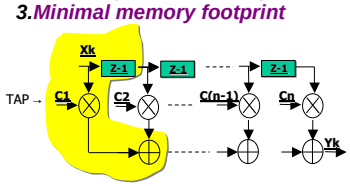
FFT $y_k = \sum w^{jk} x_j$, for $j = 0, \ldots, N-1$, where $w = e^{-2\pi i/N}$

2D DCT $F(u,v) = 1/N \sum c_m c_n e_{mn}(u,v) \cos[(2m+1)u\pi/2N] \cos[(2n+1)v\pi/2N]$
, for $m, n = 0, \ldots, N-1$

I. ***Heavy arithmetic*** computations
II. Can be easily programmed into
Tight Small Loops

DSP (Digital Signal Processor)

- Programmable processor for mathematical operations to manipulate signals with
  1. *High performance,*
  2. *Minimal power consumption*
  3. *Minimal memory footprint*



Finite Impulse Response (FIR)

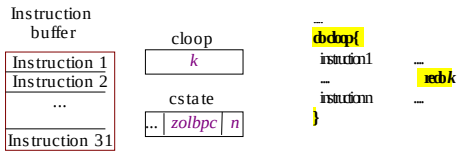
At the least, compute one *Tap* in a *Single Cycle*

Lucent DSP16000 Architecture Features

1. *Havard Architecture*
2. *Separate AGU from DALU for rich addressing modes*
3. *Zero-wait State High Speed Memory*

Lucent DSP16000 Architecture Features (cont)

4. *Compiler (Programmer) Controlled On-Core Instruction Cache – ZOLB (Zero Overhead Loop Buffer) to support high performace with minimal power dissipation*



Lucent DSP16000 Instruction Set Design

In order to achieve performance & higher code density

A0 = A0 + P0 P0 = Xh*Yh P1 = Xl * Yl Y = *R0++ X = *PT0++

16 bit word instruction

- Permissible order of operations is very limited
- The register usage is restricted to only a few different registers

Compiler Mission!

Where are the compound/complex instructions?

```
// EDN Benchmarks
fir(const short array1[ ],
    const short coeff[],
    short output[])
{
   int i,j,sum;

   for(i=0;i < N-ORDER;i++){
      sum=0;
      for(j=0; j < ORDER; j++){
            sum += array1[i+j]*coeff[j];
         }
      output[i]=sum>>15;
   }
}
```

```
A2=0
j=a4

do 50 {
/* inst 1 */   xh = *(r0 + j)
/* inst 2 */   yh = *r3++
/* inst 3 */   r4 = j
/* inst 4 */   p0 = xh*yh  p1 = xl*yl
/* inst 5 */   a2 = a2+p0
/* inst 6 */   j = r4+1
}
```

A0 = A0 + P0 P0 = Xh*Yh P1 = Xl * Yl Y = *R0++ X = *PT0++

Experience with Iterative Modulo Scheduling Techniques

*EDN Benchmark: **FIR** Filter*

```
fir(const short array1[ ], const short coeff[], short output[])
{
   int i,j,sum;

   for(i=0;i < N-ORDER;i++){
      sum=0;
      for(j=0; j < ORDER; j++){
            sum += array1[i+j]*coeff[j];
         }
      output[i]=sum>>15;
   }
}
```

```
a2=0
j=a4

do 50 {
/* inst 1 */   xh = *(r0 + j)
/* inst 2 */   yh = *r3++
/* inst 3 */   r4 = j
/* inst 4 */   p0 = xh*yh  p1 = xl*yl
/* inst 5 */   a2 = a2+p0
/* inst 6 */   j = r4+1
}
```

Step 1: Resource Inition Interval

MII = MAX(RecII, ResII)

ResII : Smallest Loop Initiation Interval to meet the system resource requirement

```
do 50 {
/* inst 1 */   xh = *(r0 + j)
/* inst 2 */   yh = *r3++
/* inst 3 */   r4 = j
/* inst 4 */   p0 = xh*yh  p1 = xl*yl
/* inst 5 */   a2 = a2+p0
/* inst 6 */   j = r4+1
}
```

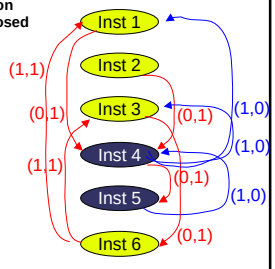
ResII: Resource Initiation Interval = 2

Step 2: Recurrence Initiation Interval

MII = MAX(RecII, ResII)

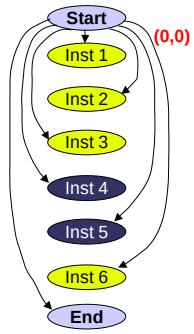
RecII : Smallest Integer Loop Initiation Interval to meet all the deadlines imposed by data dependence circuits.

```
do 50 {
/* inst 1 */   xh = *(r0 + j)
/* inst 2 */   yh = *r3++
/* inst 3 */   r4 = j
/* inst 4 */   p0 = xh*yh  p1 = xl*yl
/* inst 5 */   a2 = a2+p0
/* inst 6 */   j = r4+1
}
```



True Dependence
Output Dependence
Anti Dependence

Step 2: RecII (cont)



Adjacency Matrix

	Start	Inst-1	Inst-2	Inst-3	Inst-4	Inst-5	Inst-6	End
Start	X	(0,)	(0,)	(0,)	(0,)	(0,)	(0,)	(0,)
Inst-1	X	X	X	X	(0,)	X	X	(0,)
Inst-2	X	X	X	X	(0,)	X	X	(0,)
Inst-3	X	X	X	X	X	X	(0,)	(0,)
Inst-4	X	(0,)	((0,)	X	X	(0,)	X	(0,)
Inst-5	X	X	X	X	(0,)	X	X	(0,)
Inst-6	X	(1,)	X	(1,)	X	X	X	(0,)

Step 2: Compute MinDIST Matrix

Adjacency Matrix

	Start	Inst-1	Inst-2	Inst-3	Inst-4	Inst-5	Inst-6	End
Start	X	(0)	(0)	(0)	(0)	(0)	(0)	(0)
Inst-1	X	X	X	X	(0)	X	X	(0)
Inst-2	X	X	X	X	(0)	X	X	(0)
Inst-3	X	X	X	X	X	X	(0)	(0)
Inst-4	X	(1)	(1)	X	X	(0)	X	(0)
Inst-5	X	X	X	X	(1)	X	X	(0)
Inst-6	X	(1)	X	(1)	X	X	X	(0)

Floyd Algorithm: MinDist[i,i] ≤ 0
with II (Initiation Interval) 2

	Start	Inst-1	Inst-2	Inst-3	Inst-4	Inst-5	Inst-6	End
Start	X	0	0	0	1	2	1	2
Inst-1	X	**-1**	-1	X	1	2	X	2
Inst-2	X	-1	**-1**	X	1	2	X	2
Inst-3	X	0	-1	**0**	1	2	1	2
Inst-4	X	-2	-2	X	**-1**	1	X	1
Inst-5	X	-4	-4	X	-2	**-1**	X	0
Inst-6	X	-1	-2	-1	0	1	**0**	1
End	X	X	X	X	X	X	X	X

Step 3: Slack Scheduling by computing Estart and Lstart

Floyd Algorithm: MinDist[i,i] ≤ 0
with II (Initiation Interval) 2

	Start	Inst-1	Inst-2	Inst-3	Inst-4	Inst-5	Inst-6	End
Start	X	0	0	0	1	2	1	2
Inst-1	X	**0**	-1	X	1	2	X	2
Inst-2	X	-1	**0**	X	1	2	X	2
Inst-3	X	0	-1	**0**	1	2	1	2
Inst-4	X	-2	-2	X	**0**	1	X	1
Inst-5	X	-4	-4	X	-2	**0**	X	0
Inst-6	X	-1	-2	-1	0	1	**0**	1
End	X	X	X	X	X	X	X	X

Legal Partial Schedule based on Estart and Lstart

Operation	Slack		Issue Time
	Estart	Lstart	
Inst-1	0	1	0
Inst-2	0	1	1
Inst-3	0	1	0
Inst-4	1	1	1
Inst-5	0	1	1
Inst-6	1	1	1

Why Modulo Scheduling is not suitable?

Legal Partial Schedule based on SLACK

Operation	Slack		Issue Time
	Estart	Lstart	
Inst-1	0	1	0
Inst-2	0	1	1
Inst-3	0	1	0
Inst-4	1	1	1
Inst-5	0	1	1
Inst-6	1	1	1

```
// inst-1 && inst-3
xh=*(r0+j)    r4=j
```

No Legal Encoding

```
// inst-2 && inst-4 && inst-5 && inst-6
yh=*r3++ p0=xh*yh p1=xl*yl a2=a2+p0 j=r4+1
```

Why Modulo Scheduling is not suitable?

Due to limited encoding space*, DSP16000 compound instructions that account for *{Inst-i, Inst-j, Inst-k}*,but there is NO legal encoding to capture any subset of *{Inst-i,Inst-j,Inst-k}

How to Overcome?

- Software pipelining optimization must be sensitive to ***Instruction Selection***
- This requires that the Instruction selection performs the following tasks in a demand driven manner
  - proactively perform *Register Renaming*
  - proactively introduce additional *micro-operations* on the fly

New Compiler Strategy

- Task 1: ***Partition*** a given loop body into instruction groups, $G_1, G_2, ..., G_n$, such that instructions in G_k can potentially be packed in G_{k-1} where $k = (n+1), (n-2), ..., n$
- Task 2: ***Restructure*** the loop such that its body consists of n instruction groups, where each group is selected to be executed in one loop iteration
- Task 3: Perform ***Instruction Selection*** among groups in the restructured loop body such that selected instructions can be combined to form instructions. If necessary, perform register renaming and passively introduce extra instruction(s) to form a potential set of parallel operations called LSPKGs for packing

Potentially Pipelineable

Instructions I_i and I_j are ***potentially pipelineable*** only when the following two conditions can be met.

1. There exists a compound/complex instruction template that can hold (may be more) both effects I_i and I_j in parallel. This implies that I_i and I_j can be potentially combined into a single complex instruction.

2. The Distance from I_i in the instruction Group G_k to I_j *can meet the Minimum Distance requirements* (***MinDist[i,j]***).

Another FIR Filter from a Customer

Instruction	DSP Code Segment
	do2{
I1	y=*r0++ x=*pt0++
I2	p0=xh*yh p1=xl*yl
I3	a0=a2
I4	a0=a0+p1
I5	*r2+=a0
I6	a0=a2
I7	a0=a0+p0
I8	*r2+=a0
	}

	Strt	Ins1	Ins2	Ins3	Ins4	Ins5	Ins6	Ins7	Ins8	Initl
Strt	(X)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)
Ins1	(X)	(X)	(1)	(X)	(X)	(X)	(X)	(1)	(X)	(1)
Ins2	(X)	(1)	(X)	(X)	(1)	(X)	(X)	(1)	(X)	(1)
Ins3	(X)	(X)	(X)	(X)	(1)	(1)	(1)	(1)	(1)	(1)
Ins4	(X)	(X)	(1)	(1)	(X)	(1)	(1)	(1)	(1)	(1)
Ins5	(X)	(X)	(X)	(1)	(1)	(X)	(1)	(1)	(1)	(1)
Ins6	(X)	(X)	(X)	(1)	(1)	(1)	(X)	(1)	(1)	(1)
Ins7	(X)	(X)	(1)	(1)	(1)	(1)	(1)	(X)	(1)	(1)
Ins8	(X)	(X)	(X)	(1)	(1)	(1)	(1)	(1)	(X)	(1)
Initl	(X)	(X)	(X)	(X)	(X)	(X)	(X)	(X)	(X)	(X)

Another FIR Filter from a Customer

MinII = 6
MinDist[ ][ ] Matrix

	Strt	Ins1	Ins2	Ins3	Ins4	Ins5	Ins6	Ins7	Ins8	Initl
Strt	X	0	1	1	2	3	4	5	6	6
Ins1	X	***-5***	1	1	2	3	4	5	6	6
Ins2	X	-6	***-2***	0	1	2	3	4	5	5
Ins3	X	-8	-2	***0***	1	2	3	4	5	5
Ins4	X	-9	-3	-1	***0***	1	3	3	4	4
Ins5	X	-10	-4	-2	-1	***0***	1	2	3	3
Ins6	X	-11	-5	-3	-2	-1	***0***	1	2	2
Ins7	X	-12	-6	-4	-3	-2	-1	***0***	1	1
Ins8	X	-13	-7	-5	-4	-3	-2	-1	***0***	0
Initl	X	X	X	X	X	X	X	X	X	X

Step 1: Partition

```
/* initialization */
void partition( ) {
  create a new group G1;
  add an instruction I1 to G1;
  i = 2;
  j = 1;

  FOR each instruction Ii in the loop DO {
     FOR each instruction Ik in  Gj  {
        IF (Ik and Ii are potentially pipelinable) {
           j = j+1;
           create a new group  Gj ;
           Tag Gj and Gj-1 with complex
              instruction templates;
           break;
        }
     }
     add an instruction Ii to Gj;
     i = i + 1;
  }
}
```

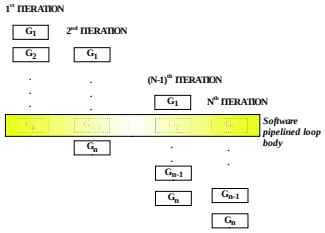
a0=a0+p0 p0=xh*yh p1=xl*yl y=*r0++ x=*pt0++

a0=a0+p0 p0=xh*yh p1=xl*yl y=*r0++ x=*pt0++

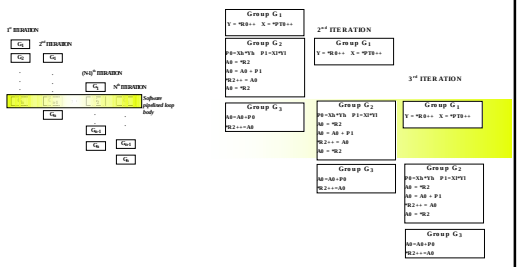
Instruction	DSP Code Segment	
	do2{	
I1	y=*r0++ x=*pt0++	$G_1 = \{I_1\}$
I2	p0=xh*yh p1=xl*yl	$G_2 = \{I_2\}$
I3	a0=a2	$G_2 = \{I_2, I_3\}$
I4	a0=a0+p1	$G_2 = \{I_2, I_3, I_4\}$
I5	*r2+=a0	$G_2 = \{I_2, I_3, I_4, I_5\}$
I6	a0=a2	$G_2 = \{I_2, I_3, I_4, I_5, I_6\}$
I7	a0=a0+p0	$G_3 = \{I_7\}$
I8	*r2+=a0	$G_3 = \{I_7, I_8\}$
	}	

a0=a0+p0 p0=xh*yh p1=xl*yl y=*r0++ x=*pt0++

Step 2: Project a Maximally Pipelined Loop



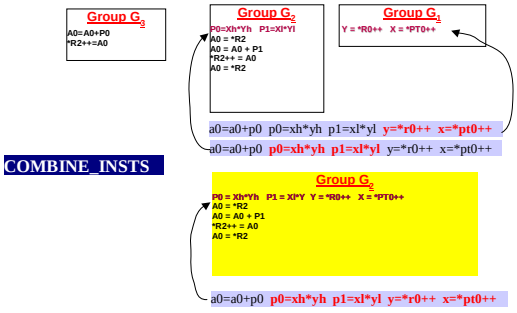
Step 2: Project a Perfectly Pipelined Loop



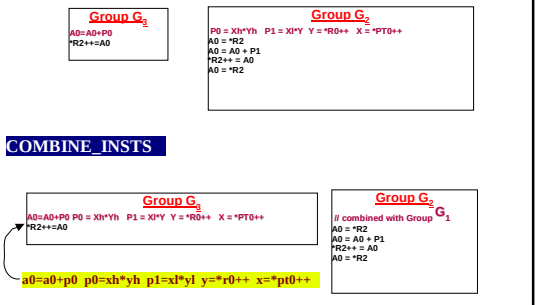
Instruction Selection Algorithm

```
/* MAIN */
DO {
   change = FALSE;;
   FOR each instruction group G_i DO {
      I_x = first instruction in group G_i;
      IF (I_x == NULL)
         CONTINUE;
      IF (change == FALSE)
         change = Combine_Insts(I_x,G_i+1);
      ELSE
        (void) Combine_Insts(I_x,G_i+1);
        /* Advance to the next instruction group */
        i = i + 1;
   } /* end of FOR */
} WHILE (change == TRUE)

/* COMBINE INSTRUCTIONS */
BOOLEAN Combine_Insts(I_w,G_w)
{
   BOOLEAN Success = FALSE;
   IF (G_w == NULL)
      /* G_w is empty */
      return FALSE;

   F= first instruction in G_w;
   FOR each instruction I_y in G_w DO {
      IF (there exists F1/F1E instruction template that
            accounts for effects I_xw )
      {
         IF (Scheduling I_y at the same time slot for I_z
               does violate a deadline imposed
               by some instruction in G_w)
            /* It is not legal schedule */
            continue;

         IF (I_xw satisfies the register encoding restrictions)
               {
            replace I_y with I_xw;
            remove I_w from G_w+1;
            Success = TRUE;
            break;
         }
         ELSE IF (there exist available register(s) that
                  can make I_xw satisfy the register
                  encoding restrictions)
         {
            perform register renaming;
            tag the I_xw with the F1/F1E instruction template;
            remove I_w from G_w+1;
            Success = TRUE;
            break;
         }
      }
      ELSE
         F = F + 1;
   } /* END FOR */

   Merge G_w+1 and G_w into one instruction group;
   RETURN FALSE;
} /* END COMBINE_INSTS */
```

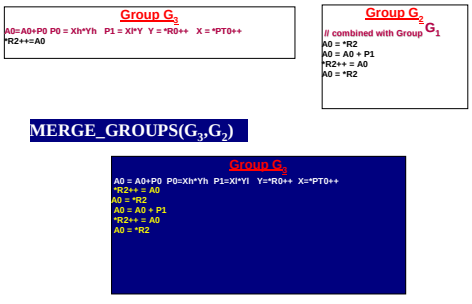
Discover Complex Instruction by Overlapping G_2 and G_1



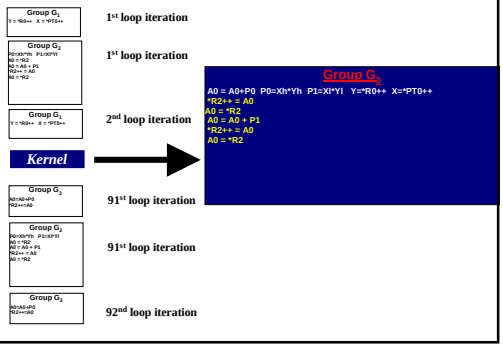
Discover Complex Instruction by Overlapping G_3 and G_2



Merge Instruction Groups G_3 and G_2



Restructured Loop



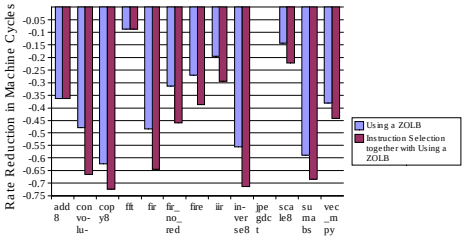
Results

Benchmark Test Programs

Program	Description
add8	Add two 8-bit images
convolution	Convolution code
copy8	Copy one 8-bit image to another
fft	128-point complex FFT
fir	Finite Impulse Response filter
fir_no_red_ld	FIR filter without redundant load elimination
fire	FIR Encoder
iir	IIR filtering
inverse8	Invert an 8-bit image
jpegdct	JPEG Discrete Cosine Transformation
scale8	Scale an 8-bit image
sumabsdiffs	Sum of absolute differences in images
vec_mpy	Simple vector multiply

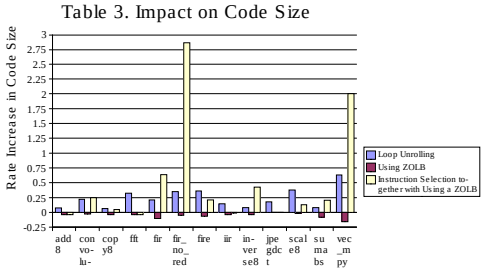
Execution Time

Table 2. Impact on Execution Time



Code Size

Table 3. Impact on Code Size



Conclusion

1. New Compiler Strategy that automatically exploits compound instructions
2. As a result of this work, the Zero Overhead Loop Buffer on Lucent DSP16000 can be further exploited

- SCOPES 2003
- Outline
- Signal Processing Algorithm
- DSP (Digital Signal Processor)
- At the least, compute one Tap in a Single Cycle
- Lucent DSP16000
- Lucent DSP16000
- Example of Using ZOLB on the DSP16000
- Instruction Set Design for Low-power Embedded Processors
- Instruction Set Design for Low-power Embedded Processors