SENG440 Embedded Systems

– Lesson 4: Software Optimization Techniques II –

Mihai SIMA

msima@ece.uvic.ca

Academic Course

Copyright © 2019 Mihai SIMA

Disclaimer

The purpose of this course is to present general techniques and concepts for the analysis, design, and utilization of embedded systems. The requirements of any real embedded system can be intimately connected with the environment in which the embedded system is deployed. The presented design examples should not be used as the full design for any real embedded system.

Lesson 4: Software optimizations for embedded systems – II

1 Long versus short integers
2 Writing efficient loops
3 Signed versus unsigned types
4 Memory alias disambiguation
5 Cache memory
6 Multi-way decision by table lookup
7 Unaligned data pointers
8 Register spilling
9 Custom operations

Lesson 4: Course Progress

- **Software optimization techniques**
  - What is wrong with plain software
  - Profile driven compilation
  - Efficient C programming
- **Standard peripherals for embedded systems**
  - **Timers, counters**, watchdog timers, real-time clocks
  - **Digital-to-Analog (D/A) and Analog-to-Digital (A/D) converters**
  - Pulse-Width Modulation (PWM) peripherals
  - Universal Asynchronous Receiver/Transmitters (UART)
- **Hardware – software – firmware.**
  - A taxonomy of processors

Long versus short integers: are optimizations possible?

- Clear **N** bytes of memory at address **data**

```
void memory_clear( char *data, int N) {
  while( N > 0) {
    *data = 0;
    data = data + 1;
    N = N - 1;
  }
}
```

- Is the **data** array pointer four-byte aligned or not?
  - YES: four bytes can be cleared at a time using an **int** rather than **char**
- Is **N** a multiple of four or not?
  - NO: the previous technique cannot be used

The optimized code

```
void memory_clear( char *data, int N) {
  int *data_int;
  data_int = (int *) data;
  N = N >> 2;
  while( N > 0) {
    *data_int = 0;
    data_int = data_int + 1;
    N = N - 1;
  }
}
```

- The compiler translates the C function into assembly so that it works for all possible inputs
- No matter how advanced the compiler, it does not know whether **data** is a four-byte aligned pointer or not, or **N** a multiple of four or not
- The compiler must be conservative $\Longrightarrow$ it tests for these cases explicitly

Variable types for loop counters

- Assume the following program in which **N** is known to be less than 100

```
for( i=0; i<N; i++) {
  < do something >
}
```

- Typical compilation rules:
  - **int** declares a 32-bit integer
  - **short int** declares a 16-bit integer
- Shall we use the type **int** for the loop counter, **i**?
- Or shall we try to reduce the memory footprint and use the type **short int** for the loop counter, **i**?

Variable types for loop counters (cont'd)

- The answer depends on the particular architecture the compiler is mapping to
- Let us assume ARM architecture:
  - Most operations are 32-bit only
- Let us assume the **short int** type for the loop counter:

```
short int i;
for( i=0; i<N; i++) {
  < do something >
}
```

- Let us focus on the **i++** operation:
  - 16-bit integer arithmetic: +32,767 + 1 = -32,768 (wraps around)
  - 32-bit integer arithmetic: +32,767 + 1 = +32,768
  - **i** is a 16-bit integer, while the ARM performs only 32-bit addition

Variable types for loop counters (cont'd)

- The compiler is conservative and **emulates** the 16-bit integer arithmetic with 32-bit integer operations
- In particular, the compiler implements the wrap around when the 32-bit argument reaches +32,768:

```
int i;
for( i=0; i<N; i++) {
  < do something >
  if( i == +32,767)
    i = -32,769;
}
```

- Since **N** is known to be less than 100, **i** never reaches +32,767, but the compiler does not know that
- Good programming technique: use **int** types for loop counters

Variable types for loop counters (cont'd)

- If the processor supports 8-bit or 16-bit operations in hardware, then such problem does not exist
- Homework: go to `http://www.intel.com` and figure out wheather Pentium supports 8-bit or 16-bit integer operations in hardware
- Typically, an embedded platform supports only N-bit integer operations in hardware where N is the machine word size
- The easiest way to figure out if a particular operation is supported in hardware is to experiment with the compiler yourself
- Compile the following program and take a look on the resulting assembly

```
char a, b, c;
int main( void) {
  c = a + b;
  exit( 0);
}
```

Integer Data Types

- No standardized sizes for fundamental data types (`char`, `short`, `int`, ...)
- When you need to know the exact number of bits of variables:

```
#include <stdint.h>

  int8_t sa;
 uint8_t ua;

 int16_t sb;
uint16_t ub;

 int32_t sc;
uint32_t uc;

 int64_t sd;
uint64_t ud;
```

How to write efficient loops

- `for` loops in C language:

```
for( <initialize counter>; <test condition>; <update counter>) {
  <LOOP BODY>
}
```

- A common way to write the loop

```
#define N 256
register int sum;
int a[ N];

sum = 0;
for( i=0; i<N; i++)
  sum += a[ i];
```

- Better ways to write the loop? Need to look at processor architecture

Loop initialization

- Different ways to initialize the contor i to zero – which one is better?

```
i = 0; /* move 0 to a register */
i ^= i; /* exclusive-or with itself */
i -= i; /* i = i - i; */
```

- Move zero to a register
  - Usually, there is an instruction **move immediate** that allow zero to be coded within instruction. Latency: 1 cycle typ.
  - If such instruction is not available, a memory access is needed. Latency: 3-5 cycles typ. (if a cache miss is not encountered).
  - Some processors have one of the registers wired to zero. This allow a move from register to register. Latency: 1 cycle typ.

Loop initialization (cont'd)

- Exclusive-or with itself
  - If the instruction set contains an XOR instruction, this initialization always work, since this is an operation from register to register. Latency: 1 cycle typ.
  - Check whether 8051 has a XOR instruction
- `i = i - i`
  - It should work in any condition
  - This is an operation from register to register. Latency: 1 cycle typ.
- `i = 0` versus `i ^= i`, `i -= i`
  - `i = 0` is a move operation, thus the condition register (zero flag) is not affected
  - `i ^= i` and `i -= i` are arithmetic operations, thus the condition register is affected by the operation result (zero flag is asserted)

The exit condition

- Increment and compare with N:

```
int a[N];

sum = 0;
for( i=0; i<N; i++)
  sum += a[ i]; /* a[0] + a[1] + ... + a[N-1] */
```

- Decrement and compare with 0:

```
int a[N+1];

sum = 0;
for( i=N; i!=0; i--)
  sum += a[ i]; /* a[1] + a[2] + ... + a[N] */
```

- Which one is better?

The exit condition (cont'd)

- Incrementation and comparison with N:

```
for( i=0; i<N; i++)
  sum += a[ i]; /* a[0] + a[1] + ... + a[N-1] */
```

  - `i++` (incrementation) takes 1 cycle
  - `i<N` takes 1 cycle to subtract N (if N is a constant, otherwise a memory access is needed), and 1 cycle to test the result (that is, the zero flag)

- Decrementation and comparison with 0:

```
for( i=N; i!=0; i--)
  sum += a[ i]; /* a[1] + a[2] + ... + a[N] */
```

  - `i-` (decrementation) takes 1 cycle
  - `i!=0` takes only 1 cycle, since only the zero flag has to be tested (zero flag has been updated during decrementation)

The exit condition – example 1

```
#include <stdio.h>

#define N 64 /* N is a constant */

int main( void) {
  int sum = 0;
  for( i=0; i<N; i++)
    sum += i; /* the sum of the first N integers */

  printf( "sum of %i integers = %i\n", N, sum);
  exit( 0);
}
```

Compile: **arm-linux-gcc -static -S file.c**

Example 1 – assembly code

```
main:   mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        sub     sp, sp, #8
        mov     r3, #0
        str     r3, [fp, #-20]
        mov     r3, #0
        str     r3, [fp, #-16]
.L3:
        ldr     r3, [fp, #-16]
        cmp     r3, #63
        ble     .L6
        b       .L4
.L6:
        ldr     r3, [fp, #-20]
        ldr     r2, [fp, #-16]
```

```
.L5:
        ldr     r3, [fp, #-16]
        add     r2, r3, #1
        str     r2, [fp, #-16]
        b       .L3
.L4:
        ldr     r0, .L7
        mov     r1, #64
        ldr     r2, [fp, #-20]
        bl      printf
        mov     r0, #0
        bl      exit
.L8:
        .align  2
.L7:
        .word   .LC0
```

The exit condition – example 2

```
#include <stdio.h>

int N; /* N is a global variable */

int main( void) {
  int sum = 0, local_N;
  scanf( "Enter N: %i\n", &N);
  local_N = N;
  for( i=0; i<local_N; i++)
    sum += i; /* the sum of the first N integers */

  printf( "sum of %i integers = %i\n", N, sum);
  exit( 0);
}
```

Compile: **arm-linux-gcc -static -S file.c**

Example 2 – assembly code

```
main:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        sub     sp, sp, #12
        mov     r3, #0
        str     r3, [fp, #-20]
        ldr     r0, .L7
        ldr     r1, .L7+4
        bl      scanf
        ldr     r3, .L7+4
        ldr     r2, [r3, #0]
        str     r2, [fp, #-24]
        mov     r3, #0
        str     r3, [fp, #-16]
.L3:
        ldr     r3, [fp, #-16]
        ldr     r2, [fp, #-24]
        cmp     r3, r2
        blt     .L6
        b       .L4
.L6:
        ldr     r3, [fp, #-20]
        ldr     r2, [fp, #-16]
        add     r3, r3, r2
        str     r3, [fp, #-20]
.L5:
        ldr     r3, [fp, #-16]
        add     r2, r3, #1
        str     r2, [fp, #-16]
        b       .L3
```

Example 2 – assembly code (cont'd)

```
.L4:
        ldr     r3, .L7+4
        ldr     r0, .L7+8
        ldr     r1, [r3, #0]
        ldr     r2, [fp, #-20]
        bl      printf
        mov     r0, #0
        bl      exit
.L8:
        .align  2
.L7:
        .word   .LC0
        .word   N
        .word   .LC1
.L2:
        ldmea   fp, {fp, sp, pc}
```

The exit condition – example 3

```
#include <stdio.h>

#define N 64 /* N is a constant */

int main( void) {
  int sum = 0;
  for( i=N+1; i!=0; i--)
    sum += i; /* the sum of the first N integers */
  sum -= (N+1);

  printf( "sum of %i integers = %i\n", N, sum);
  exit( 0);
}
```

Compile: **arm-linux-gcc -static -S file.c**

Example 3 – assembly code (cont'd)

```
main:   mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        sub     sp, sp, #8
        mov     r3, #0
        str     r3, [fp, #-20]
        mov     r3, #65
        str     r3, [fp, #-16]
.L3:
        ldr     r3, [fp, #-16]
        cmp     r3, #0
        bne     .L6
        b       .L4
.L6:
        ldr     r3, [fp, #-20]
        ldr     r2, [fp, #-16]
```

```
.L5:    ldr     r3, [fp, #-16]
        sub     r2, r3, #1
        str     r2, [fp, #-16]
        b       .L3
.L4:
        ldr     r3, [fp, #-20]
        sub     r2, r3, #65
        str     r2, [fp, #-20]
        ldr     r0, .L7
        mov     r1, #64
        ldr     r2, [fp, #-20]
        bl      printf
        mov     r0, #0
        bl      exit
.L8:    .align  2
```

The exit condition – example 4

```
#include <stdio.h>
int N; /* N is a global variable */

int main( void) {
  int sum = 0, local_N;
  scanf( "Enter N: %i\n", &N);
  local_N = N + 1;
  for( i=local_N; i!=0; i--)
    sum += i; /* the sum of the first N integers */
  sum -= local_N;

  printf( "sum of %i integers = %i\n", N, sum);
  exit( 0);
}
```

Compile: **arm-linux-gcc -static -S file.c**

Example 4 – assembly code

```
main:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        sub     sp, sp, #12
        mov     r3, #0
        str     r3, [fp, #-20]
        ldr     r0, .L7
        ldr     r1, .L7+4
        bl      scanf
        ldr     r3, .L7+4
        ldr     r2, [r3, #0]
        add     r3, r2, #1
        str     r3, [fp, #-24]
        ldr     r3, [fp, #-24]
        str     r3, [fp, #-16]
.L3:
        ldr     r3, [fp, #-16]
        cmp     r3, #0
        bne     .L6
        b       .L4
.L6:
        ldr     r3, [fp, #-20]
        ldr     r2, [fp, #-16]
        add     r3, r3, r2
        str     r3, [fp, #-20]
.L5:
        ldr     r3, [fp, #-16]
        sub     r2, r3, #1
        str     r2, [fp, #-16]
        b       .L3
```

Example 4 – assembly code (cont'd)

```
.L4:
        ldr     r3, [fp, #-20]
        ldr     r2, [fp, #-24]
        rsb     r3, r2, r3
        str     r3, [fp, #-20]
        ldr     r3, .L7+4
        ldr     r0, .L7+8
        ldr     r1, [r3, #0]
        ldr     r2, [fp, #-20]
        bl      printf
        mov     r0, #0
        bl      exit
```

```
.L8:
        .align  2
.L7:
        .word   .LC0
        .word   N
        .word   .LC1
.L2:
        ldmea   fp, {fp, sp, pc}
```

Signed versus unsigned types

- Typically, there is no performance difference between signed and unsigned addition, subtraction, multiplication
- Some problems when it comes to division:

```
int x, y;
y = x / 2;
```

- Divide by two may not be a right shift for negative arguments

```
if( x < 0)
  y = (x + 1) >> 1; /* one is added before right shifting */
else
  y = x >> 1;
```

- The compiler converts unsigned power of two divisions into right shifts

```
unsigned int x, y;
y = x >> 1;
```

Memory alias disambiguation

- **Memory aliasing** occurs when two instructions can access the same memory location – such memory references are called **ambiguous**
- **Memory alias disambiguation** is the process of determining when such ambiguity is not possible
- Ambiguity generates memory dependencies, which in turn decreases the parallelism
- **Memory disambiguation**, or **alias analysis**, is a key component of modern compilers – any optimization that reorders or changes code containing memory operations must analyze the memory references to ensure that the original semantics of the program is preserved
- There exist array references that cannot be disambiguated at compile time – thus, human intervention is needed

Memory aliasing – basic issues

```
void convolution( int *a, int *b, int *c) {
  int k;

  for( k=0; k < NO_SAMPLES; k+=2) {
    c[ 0] = b[ 0]*a[ 0] + b[ 1]*a[-1]
          + b[ 2]*a[ 2] + b[ 3]*a[-3];
    c[ 1] = b[ 0]*a[ 1] + b[ 1]*a[ 0]
          + b[ 2]*a[-1] + b[ 3]*a[-2];
    a += 2;
    c += 2;
  }
}
```

- **The programer intention:** `a`, `b`, and `c` represent different signals; thus, their memory locations do not overlap

Memory aliasing – basic issues (cont'd)

- That is, the programer intend to call the function `convolution` ONLY with different arguments:

```
int main( void) {
  int signal_a[100], signal_b[100], signal_c[100];

  ...
  convolution( signal_a, signal_b, signal_c);
  ...
}
```

- When compiling the `convolution` routine, the **compiler cannot "guess" the programer intention**. Thus, the compiler follows a conservative approach and assumes that the memory locations referenced by `a`, `b`, and `c` might overlap

Memory aliasing – basic issues (cont'd)

- Thus, the compiler assumes that calls with identical arguments can occur

```
int main( void) {
  int signal_a[100], signal_b[100], signal_c[100];

  convolution( signal_a, signal_b, signal_a);
}
```

  (of course the `convolution` result is incorrect in this case, but this has no relevance at compile time)

  The parallelism decreases, and thus the parallel computing capabilities of the processor cannot be exploited

- Two general solutions to this problem:
  - **restrict** pointers
  - **no_alias** pragma

Restrict-qualified pointers

- **restrict** keyword
  - Introduced by 1999 ANSI/ISO C standard (C++, too?)
  - A hint only, so may do nothing but still be conforming
- A restrict-qualified pointer is basically a promise to the compiler that for the scope of the pointer, the target of the pointer will only be accessed through that pointer (and pointers copied from it)
- Restricted Pointers in C:
  `http://www.lysator.liu.se/c/restrict.html`
  `http://cellperformance.beyond3d.com/articles/2006/0`

Keyword: restrict

- Using the **restrict** keyword:

```
void convolution( int * restrict a,
                  int * restrict b,
                  int * restrict c);
```

- This is a promise to the compiler that the three pointers (`a`, `b`, and `c`) will never overlap inside the `convolution` function
- The compiler assumes the programmer is not *lying*.

no_alias pragma

```
add( double *d, double *s1, double *s2, int n)
#pragma no_alias *d, *s1, *s2
{
  int i;

  for (i = 0; i < n; i++)
    d[i] = *s1 + *s2;
}
```

- Recall that *pragma*s are non-standard constructs
- Check the C handbook of the processor you are using
- Check what compilation flags are available (such as, `no-strict-aliasing`)

Cache memory – general considerations

- We want a large amount of fast and cheap memory – not possible
  - Fast memory is expensive (e.g., SRAM)
  - Cheap memory is slow (e.g., DRAM)
- Cache memory cheats the processor and make it "think" that a large amount of fast and cheap memory is deployed
- **Principle of locality**: Programs tend to reuse data and instructions they have used recently
  - **Temporal locality**: recently accessed items are likely to be accessed in the near future
  - **Spatial locality**: items whose addresses are near one another tend to be referenced close together in time

Cache memory – general considerations (cont'd)

- A **cache** is a small, fast memory located close to the processor that holds the most recently accessed code or data
  - When the processor finds a requested item in the cache, it is called a **cache hit**
  - When the processor does not find a data item it needs in the cache, a **cache miss** occurs.
  - A fixed-size block of data containing the requested word is retrieved from the memory and placed into the cache
- Typical latency for cache access is 3-5 cycles, while the typical overhead per cache miss is 10-50 cycles
- Can we optimize the code to minimize the number of cache misses?
- Yes, if we know the cache organization

Cache access optimization

- In C, the rightmost subscript of a multidimensional array varies fastest as elements are accessed in storage order
- Let us assume that the cache line size is 64 bytes
- When a miss is detected, an entire cache line is replaced

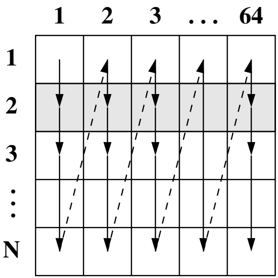
```
char a[300][64];

main( void) {
  int k, l; /* counters */

  for( l=0; l < 64; l++)
    for( k=0; k < 300; k++)
      a[k][l] = 0;
}
```

```
char a[300][64];

main( void) {
  int k, l; /* counters */

  for( k=0; k < 300; k++)
    for( l=0; l < 64; l++)
      a[k][l] = 0;
}
```

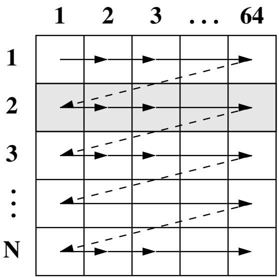
Cache access optimization (cont'd)

```
for( l=0; l < 64; l++)
  for( k=0; k < 300; k++)
    a[k][l] = 0;
```

```
for( k=0; k < 300; k++)
  for( l=0; l < 64; l++)
    a[k][l] = 0;
```



Cache miss at
every memory access



Cache miss per
64 memory accesses

Multi-way decision by table lookup

- If the `switch` statement is compiled into a sequence of test and jump operations, and if there are too many switch branches, then a significant CPU time is spent wasted to decide what work should be done next

```
enum Node_Type { Node_A, Node_B, Node_C};
switch ( get_Node_Type()) {
  case Node_A: ...;
  case Node_B: ...;
  case Node_C: ...;
  ...
}
```

- **Most likely cases first and the least likely cases last:** the average execution time is reduced, but the worst-case time isn't
- Can we do better?

Multi-way decision by table lookup (cont'd)

- **We can do better**
- To speed things up, replace the `switch` statement with a **table lookup**:
- First, create an array of function pointers:

```
int p_Node_A( void); /* process_Node_A */
int p_Node_B( void); /* process_Node_B */
int p_Node_C( void); /* process_Node_C */

int (* node_Functions[])(void) = {p_Node_A, p_Node_B, p_Node_C};
```

- Then, replace the entire `switch` statement with one-line function call:

```
node_Functions[ get_Node_Type()]();
```

Unaligned data pointers

```
void memory_clear( char *data, int N) {
  int *data_int;
  data_int = (int *) data;      /* is data_int properly aligned? */
  N = N >> 2;
  while( N > 0) {
    *data_int = 0;
    data_int = data_int + 1;
    N = N - 1;
  }
}
```

- A C program may manipulate pointers directly so that they may become unaligned (for example, by casting a **char** * to an **int** *)
- Some architectures may support unaligned pointers!
- To detect unaligned accesses you typically have a number of compilation options (e.g., alignment checking trap)

Forcing data alignment

- Assume a variable of type `char` whose address should be a multiple of 4 bytes (may be good for cache optimization)
- The linker does not guarantee such an alignment
- Trick: build an **union**:

```
union u_align {
  int ival;
  float fval;
  char cval;
} u;
```

- The variable **u** will be large enough to hold the largest of the three types
- The variable **u** will be aligned according to the largest of the three types

Register spilling

- There is a limited number of registers that are available to the compiler for allocation to register variables and temporary expression results
- If the compiler cannot allocate a register, then **spilling** occurs
- Spilling is the process of moving a register's content to memory to free the register for another purpose
- Register allocation is not under the direct control of the programer
- However, we can use the `register` keyword to advice the compiler that we would like a particular variable be stored into a register:

```
int example( register int input_arg) { /* formal parameter */
  register char a; /* automatic variable */
  ...
}
```

- The compiler is free to ignore our advice
- It is not possible to take the address of a register variable

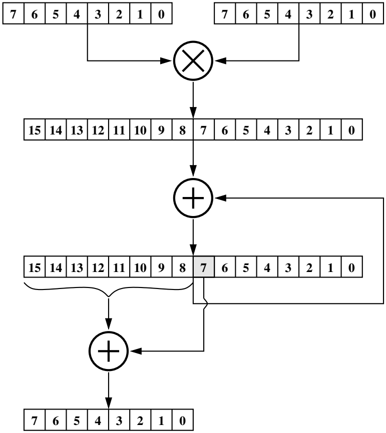
Application-specific libraries and custom operations

- Assume hardware support for a complex DSP operation: "Multiply And Accumulate with Shift and Rounding"

```
MACSR Rs1, Rs2, Rd
```

- Assume the following C code:

```
char macsr( char a, char b) {
  static int temp_acc;
  int shifted_acc, rounding_bit;

  temp_acc += a * b;
  shifted_acc = temp_acc >> 8;
  rounding_bit = (temp_acc >> 7) & 1;
  return shifted_acc + rounding_bit;}
```



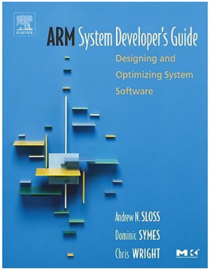
Application-specific libraries and custom operations (cont'd)

- Is the compiler able to match the semantics of the hardware operation and C routine? Usually **NO**
- We have to **write** that particular code in assembly and **call it**
- Two approaches to make this task easier:
  - The vendor provides for a library of optimized assembly functions that can be called or inlined into the source code. (Texas Instruments)
  - The vendor provides for a set of custom operations that are recognized by compiler (and thus translated into hardware operations) (Philips)

```
#include <trimedia.h>
int main( void) {
  char a, b, c;
  c = macsr( a, b);
}
```

Additional bibliography

1. ARM web site: `http://www.arm.com`
2. Very nice book!
   Andrew Sloss, Dominic Symes, Chris Wright,
   ARM System Developer's Guide. Designing and Optimizing System Software,
   Morgan Kaufmann 2004.
3. Other nice book:
   Daniel W. Lewis,
   Fundamentals of Embedded Software with the ARM Cortex-M3,
   Pearson 2013.



Additional bibliography

1. **J. Brenner, Cache Usage in High-Performance DSP Applications with the TMS320C64x, Application Report SPRA756, Dec. 2001, http://www.ti.com**
2. ***, TMS320C6000 Optimizing C Compiler Tutorial (Rev. A), Aug. 2002, http://www.ti.com
3. ***, TMS320C6000 Optimizing Compiler User's Guide (Rev. K), Oct. 2002, http://www.ti.com
4. ***, TMS320C6000 Programmer's Guide (Rev. G), Aug. 2002, Oct. 2002, http://www.ti.com
5. ***, TMS320C55x DSP Programmer's Guide (Rev. A), July 2001, http://www.ti.com

Questions, feedback

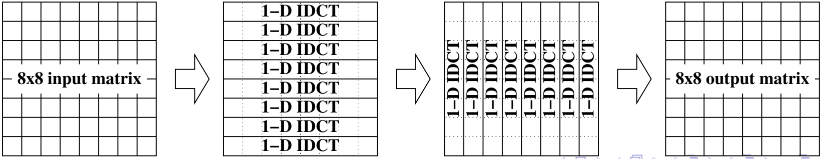
?

An example: (I)DCT for MPEG video

- (I)DCT = (Inverse) Discrete Cosine Transform
- (I)DCT is a 2-dimensional transform applied to 8×8 blocks of video data
- IDCT should usually be computed in real-time

$$x_{i,j} = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} K_u K_v X_{u,v} \cos \frac{(2i+1)u\pi}{16} \cos \frac{(2j+1)v\pi}{16}$$

- It is computationally intensive – the processor cannot do it in real-time
- Too costly to have a hardware assist for 2-D IDCT
- Trade-off: row-column separation computing strategy

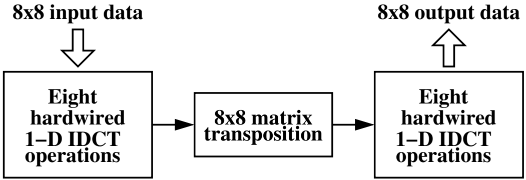


An example: (I)DCT for MPEG video (cont'd)

- 1-D IDCT

$$x_i = \frac{1}{4} \sum_{u=0}^{7} K_u X_u \cos \frac{(2i+1)u\pi}{16}$$

- Still computationally intensive. This time, however, our silicon budget allows us to implement the 1-D operation in hardware
- We augment a general-purpose processor with custom operations: Application-Specific Instruction-set Processor (ASIP)
- Computing scenario in the ASIP



An example: (I)DCT for MPEG video (cont'd)

- Our application-specific instruction-set processor has:
  - general-purpose instructions:

    ADD (latency=1)
    SUBTRACT (latency=1)
    MULTIPLY (latency=2)
    LOAD/STORE (latency=5)
    TRANSPOSE (latency=1)
    JUMP (latency=5)

  - application-specific instructions:

    IDCT (latency=16)

- One computing unit for each operation
- Operations can be executed in parallel
- **How to write a 2-D IDCT routine**

Writing a 2-D IDCT routine – the brute force

```
int 2D_IDCT( int * input_stream, int * output_stream) {
  int i_block[8][8]; o_block[8][8]; /* input & output matrices */
  int t1_block[8][8]; t2_block[8][8]; /* temporary matrices */
  int k, l; /* counters */

  for( k=0; k<8; k++)  for( l=0; l<8; l++)
    i_block[k][l] = input_stream[8*k+l];

  for( k=0; k<8; k++)  idct( i_block[k], t1_block[k])
  transposition( t1_block, t2_block);
  for( l=0; l<8; l++)  idct( t2_block[l], o_block[k])

  for( k=0; k<8; k++)  for( l=0; l<8; l++)
    output_stream[8*k+l] = o_block[k][l];
}
```

Optimizing the “brute force” solution

- Store local variables into registers
- Loop unrolling for the `for` loops

```
int i_block_0_0, i_block_0_1, ..., i_block_0_7, i_block_1_0, ...
int o_block_0_0, o_block_0_1, ..., o_block_0_7, o_block_1_0, ...

i_block_0_0 = input_stream[ 0];
i_block_0_1 = input_stream[ 1];
...
i_block_1_0 = input_stream[ 8];
...
output_stream[ 0] = o_block_0_0;
output_stream[ 1] = o_block_0_1;
...
```

- **Are there enough registers to avoid spilling?**

Optimizing the “brute force” solution (cont’d)

- SW pipelining on LOAD and IDCT: input_current (ic) and input_next (in)

```
int ic_block_0_0, ic_block_0_1, ..., ic_block_0_7, ic_block_1_0, .
int in_block_0_0, in_block_0_1, ..., in_block_0_7, in_block_1_0, .

in_block_0_0 = input_stream[ 0];
in_block_0_1 = input_stream[ 1];
...
in_block_0_7 = input_stream[ 7];

idct( ic_block_0_0, ic_block_0_1, ..., t1_block_0_0, t1_block_0_1,

ic_block_0_0 = in_block_0_0;
ic_block_0_1 = in_block_0_1;
...
ic_block_0_7 = in_block_0_7;
```

Optimizing the “brute force” solution (cont’d)

- SW pipelining on IDCT and STORE: output_current output_next (on)
- SW pipelining on IDCT–TRANSPOSE, as well as TRANSPOSE–IDCT
- Memory alias disambiguation

```
int 2D_IDCT( int * restrict input_stream,
             int * restrict output_stream);
```

- Make sure that `idct()` is translated to machine operation

```
#pragma ASIP_OPERATION
idct( ic_block_0_0, ic_block_0_1, ..., t1_block_0_0, t1_block_0_1,
```

- Try to further optimize this code!

Second worked example – typical for midterm/final

Due to the large number of true dependencies, the code below cannot be executed efficiently on a parallel computing engine.

```
unsigned int i, a[100], b[100];

int main( void) {
  i = 0;                      /* initialize counter to 0 */
  while( i<100) {
    b[i] = a[i] * a[i+1];
    b[i] = Log( b[i]);
    b[i] = b[i] * b[i];
    i = i + 2;                /* increment counter       */
  } }
```

Use software optimization techniques to reduce the number of true dependencies within the loop body.

Second worked example – typical for midterm/final

- True dependencies:
  - Operation 02 needs the result of Operation 01 in order to commence
  - Operation 03 needs the result of Operation 02 in order to commence
  - Operation 04 (incrementation of i) cannot commence before Operation 03
- Assume a parallel processor in which multiplication, logarithm, addition, ..., can run in parallel
- A good compiler might be able to perform software pipelining, unrolling, and any other software optimizations automatically
- No harm to rewrite the code in order to give the compiler a helping hand
  - We say that we **expose the parallelism to the compiler**
- Typically, a compiler cannot see beyond the loop boundaries, and as such we will focus to optimize the code within the loop body

Second worked example – typical for midterm/final

- First idea: two-stage software pipeline: swap Operations 01 and 02
- No true dependency between Operation 02 and Operation 01 within the loop body – these operations can be executed in parallel (if the processor has parallel computing capabilities, such as it is a superscalar or VLIW engine)
- There is still a true dependency between Operation 02 and Operation 01, but this dependency is across the loop boundary

```
          ...
          i = 0;                   /* initialize counter to 0 */
          temp = a[0] * a[1];      /* prologue                */
          while( i<100) {
02          b[i] = Log( temp);
03          b[i] = b[i] * b[i];
01          temp = a[i+2] * a[i+3];
04          i = i + 2;             /* increment counter       */
          }
```

Second worked example – typical for midterm/final

- **Common mistake**: by merging Operation 02 and Operation 03 (see the code below) the true dependency is eliminated – NOT TRUE!
- The true dependency still exists (since it is not possible to perform multiplication before logarithm completes), although the true dependency is hidden.

```
        ...
        i = 0;                      /* initialize counter to 0 */
        temp = a[0] * a[1];         /* prologue                */
        while( i<100) {
02-03     b[i] = Log( temp) * Log( temp);
01        temp = a[i+2] * a[i+3];
04        i = i + 2;                /* increment counter       */
        }
```

Second worked example – typical for midterm/final

- Please note that although the operation `temp = a[100] * a[101];` is dummy, it needs extra space in the `a[]` and `b[]` arrays.
- There is still a true dependency between Operation 03 and Operation 02. A three-stage software pipeline will eliminate it!

```
     unsigned int i, a[102], b[102], temp;
     int main( void) {
       i = 0;                 /* initialize counter to 0 */
       temp = a[0] * a[1];    /* prologue                */
       while( i<100) {
02       b[i] = Log( temp);
03       b[i] = b[i] * b[i];
01       temp = a[i+2] * a[i+3];
04       i = i + 2;           /* increment counter       */
       }
     }
```

Second worked example – three-stage software pipeline

- Operations `temp = a[100]*a[101];` and `temp = a[102]*a[103];` are dummy – extra space in the `a[]` and `b[]` arrays is needed

```
     unsigned int i, a[104], b[104], temp_1, temp_2;
     int main( void) {
       i = 0;                   /* initialize counter to 0 */
       temp_1 = a[0] * a[1];    /* prologue                */
       temp_2 = Log( temp_1);   /* prologue                */
       temp_1 = a[2] * a[3];    /* prologue                */
       while( i<100) {
03       b[i] = temp_2 * temp_2;
02       temp_2 = Log( temp_1);
01       temp_1 = a[i+4] * a[i+5];
04       i = i + 2;             /* increment counter       */
       }
     } /* Where is the epilogue? Why we don't need it here? */
```

Second worked example – typical for midterm/final

- Global variables have no chance to be stored in registers! Use local variables for `temp_1` and `temp_2` and try the modifier `register`

```
    unsigned int a[104], b[104];
    int main( void) {
      register int i;
      register unsigned int temp_1, temp_2;
      i = 0;                    /* initialize counter to 0 */
      temp_1 = a[0] * a[1];     /* prologue                */
      temp_2 = Log( temp_1);    /* prologue                */
      temp_1 = a[2] * a[3];     /* prologue                */
      while( i<100) {
03      b[i] = temp_2 * temp_2;
02      temp_2 = Log( temp_1);
01      temp_1 = a[i+4] * a[i+5];
04      i = i + 2;              /* increment counter       */
      } }
```

Second worked example – typical for midterm/final

- Regarding the memory access count: it is useless to declare the arrays as local entities (see the code below). Reason: an array needs to be stored in memory (it is not possible otherwise from obvious reasons)
- Can the array name (which is a pointer, that is, a single value) be stored only in a register?
  - Write some C code, compile, and find out!

```
int main( void) {
  register int i;
  register unsigned int temp_1, temp_2;
  unsigned int a[104], b[104];
  i = 0;                   /* initialize counter to 0 */
  temp_1 = a[0] * a[1];    /* prologue                */
  temp_2 = Log( temp_1);   /* prologue                */
  temp_1 = a[2] * a[3];    /* prologue                */
  ...
```

Questions, feedback

?

Notes I

Notes II

Notes III