C6000 Architecture Specific Topics

Barath Ramesh

July 21, 2012

Contents

1 Intro to C6000 Architecture

Solve Digital signal procssing problem. ADC–¿DSP(Algorithm)–¿DAC. Heart of DSP algo is a MAC (sum of products). Solve this problem efficiently, most of the DSP algo should run faster.

2 C6x CPU Architecture

Load store architecture, read values in section A or B. Exectuion unit (.M1, .M2, .L1, .L2). M–¿Multiply, .L add(logical) together called MACs. Supports 8 MACs/cycle (8x8 and 16x16).D1 .D2 load/store, .S1 .S2 shifts and other logical operations. 8 independent functions (8 independent functional units) or * instructions per cycle. Thus while MMACs speed math intensive algorithms, flexibility of 8 independent functional units allows comiler to perform other types fo processing. C6x CPU can dispatch up to eight parallel instructions each cycle. All C6x instruction are conditional allowing efficient hardware pipelining.

3 C6000 C compiler optimization

What does "Optimal Mean?"

- When my processing keeps up with my I/O (real time)
- When my algo acheives theoritical minumum (purple patch)
- After I have applied all known optmimization techniques

Real-time vs. CPU Min

- Typically, meeting real-time only requires setting a few compiler options (easy)
- Acheiving "CPU Min" often requires extensive knowledge of the architecture (harder, requires more time).

4 Optimization - Intro

"Optimization is a continuous process of refinement in which code being optimized executes faster and takes fewer cycles, until a specific objective is acheived (real time execution).

- Learn as many optimizations as possible and try them all if necessary

5 Debug vs Optmized mode

Turn on the optimizer for better performance. See Table 1 for few benchmarks with and without optimizations.

Table 1: Benchmarking

Alog	FIR(256, 64)	DOTP (256-term)
Debug (no opt, -g)	817K	4109
Opt (-o3, no -g)	18K	42
Add'I pragmas	7K	42
(DSPLib)	7K	42
CPU Min	4096	42

Table 2: Minimizing code Space

-ms level	Performance	Code Size
none	100%	0%
-ms0	90%	10%
-ms1	60%	40%
-ms2	20%	80%
-ms3	0%	100%

6 Build Configurations

Two build configurations available debug and release (Set Active). Custom optimizations can be made by clicking manage.

- Debig no opt, -g
- Active -o3, no -g

Include paths are not copied from Debug to Active build configuration.

7 Size Optimization

Code size -ms works in conjunction with -o. Table 2

8 File and Function specific options

Use compiler directives:

#pragma FUNCTION_OPTINS( )

9 Viewing Options in CCSv5.2

Need to be done (how to set compiler optimization)

Table 3: C6000 Data Types

Type	Bits	Representation
char	8	ASCII
short	16	Binary 2's complement
int	32	Binary 2's complement
long	40	Binary 2's complement
float	32	IEEE 32-bit
double	64	IEEE 64-bit
long double	64	IEEE 64-bit
pointers	32	Binary

10 Coding Guideline

C, C++ source code –Compiler Optimizer–¿ 80%-100% efficiency with minumum effort. ASM –Hand Optimize–¿ 100% with high effort. Linear ASM – Assembly Optimizer–¿ 95%-100% with medium effort.

10.1 Basic C coding guidelines

- Keep code complexity simple
- No functions calls in tight loops
- Keep llops relatively small
- Look at the assembly file - SPLOOP, -k option to keep the assembly file

11 Data types and Alignment

11.1 Supported Data types and format

Data types supported is as shown in table 3

11.2 Data Alignment

```
#pragma DATA_ALIGN(x, 4)
short z;
short x;
```

12 Restricting memory dependence, Aliasing

12.1 Aliasing

Two different ways of accessing a memory location:

```
int x;
int *p;

main(){

p = &x;

x=5;
*p = 8;
}
```

Such simple programs do not affect the compiler. To help the compiler for s/w pipelining tell the compiler that they are two different memory location (no aliasing) use *restrict* key word.

12.2 Alias Solution

- Program level opttimization (-pm -03)
- No bad Alisasing option (-mt), tell the compiler that there is no bad aliasing in my entire project
- "Restrict" keyword (ANSI C) void fcn(short * in, short * restrict out)

13 Accessing H/W features

C code using intrinsics:

```
y = a * b;
y = \_mpyh(a, b);
```

In-Line Assembly:

```
asm(" MPYH A0, A1, A2");
```

Assembly code

```
MPYH A0, A1, A2
```

Intrinsics:

- Can use C variable names instead of register names
- Are compatible with C environment
- Adhere to C's function call syntax
- Do not use in-line assembly

14 Using Pragmas

- #pragma UNROLL(# times to unroll)
- #pragma MUST_ITERATE(min, max ,%factor)