DSP Mapping, Coding, Optimization

On TMS320C6000 Family using CCS (Code Composer Studio) ver 3.3

Started with writing a simple C code in the class, from scratch

Project called 'First', written for C6713 cycle accurate simulator

SPRU198i.pdf chapters 2,3,4,5 - Chassaing book chapters 3 and 8, CCS help, …

Now, rules to optimize DSP C code

Data Types:

- char 8 bits
- short 16 bits
- int 32 bits
- long 40 bits
- long long 64 bits
- float 32 bits
- double 64 bits

not the same as in PC Windows C programming

Basic Hints:

1) *Use short vars when multiplication is involved (1 cycle vs. 5 cycles)*
2) *Use int or unsigned int for loop counters*
3) *Use correct libraries → e.g. for C67 to use floats*

DSP Mapping, Coding, Optimization…

Analyzing Performance

1) Use clock() function

S =clock(); E= clock(); O= E-S; // Clock overhead

*S =clock(); /*Function or range To evaluate*/ ; E= clock(); printf …*

2) Use Profile mode

Setup Profile, Enable Clock … Run the code

3) Use Simulator Analysis in CCS

Tools > Simulator Analysis, … reset counters, choose counters, run the code

4) Use Stand-alone Simulator Analysis

load6x –g code.out *generates .vaa file*

Recognize Critical Areas → mostly loops

DSP Mapping, Coding, Optimization…

Analyzing Performance…

```
Program Name:    c:\users\igh\desktop\sa-sim\first.out
Start Address:   000012bc main, in line 18, "first.c"
Stop Address:    00005460 exit, in line 19, "exit.c"
Run Cycles:      16222
Profile Cycles:  16222
BP Hits:         788
```

Total cycles in, Including inner functions

One call Max cycles including inner functions

Name	Count	Inclusive	Incl-Max	Exclusive	Excl-Max	Address	Size	Full Name
fputc	54	14860	5825	2611	50	00004c60	368	fputc
wrt_ok	59	5122	3374	2274	70	00006000	288	_wrt_ok
writemsg	11	1062	156	1062	156	000070c0	184	writemsg
printf	5	15352	5993	714	162	00007360	132	printf
fputs	5	7673	3448	682	142	00001da0	904	fputs
readmsg	11	638	58	638	58	00007000	176	readmsg
memset	15	630	78	630	78	000056e0	296	memset
ltostr	5	1440	324	569	123	00004900	428	_ltostr
setfield	5	9715	4096	555	111	00000000	1300	_setfield
pproc_diouxp	5	2200	476	485	97	000031e0	708	_pproc_diouxp
div	13	676	52	416	32	00005dc0	268	_div
outc	54	324	6	324	6	00007780	8	_outc
pproc_fwp	5	10126	4233	320	64	000024e0	896	_pproc_fwp
write	5	1657	347	285	57	00006480	288	write
HOSTclock	6	1056	176	276	46	000069a0	224	HOSTclock
getarg_diouxp	5	275	55	275	55	00000520	1124	_getarg_dioux
HOSTwrite	5	1312	278	260	52	00005a60	260	HOSTwrite
_divu	13	260	20	260	20	00006c40	184	__divu
memccpy	5	234	50	234	50	00007620	128	memccpy
pproc_fflags	5	10276	4283	220	44	00004de0	348	_pproc_fflags
doflush	5	1862	388	205	41	00006d00	184	_doflush
nop	34	204	6	204	6	00007760	8	_nop
mpyll	13	195	15	195	15	00007220	148	_mpyll
memcpy	5	150	30	150	30	00003c60	556	memcpy
malloc	1	3304	3304	99	99	00004700	500	malloc
main	1	16222	0	74	0	000012bc	796	main
_strasgi	2	36	18	36	18	000051e0	312	__strasgi
clock	6	36	6	36	6	000077e0	8	clock
MAC	1	32	32	32	32	00001200	188	MAC

DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

1) Use Compiler options

-o2/o3 → SW pipelining, SIMD, loop unrolling -pm combining src files ,…

-ms2/ms3 → size optimization

2) Memory Dependencies

Compiler must know if two vars are independent to schedule them in parallel

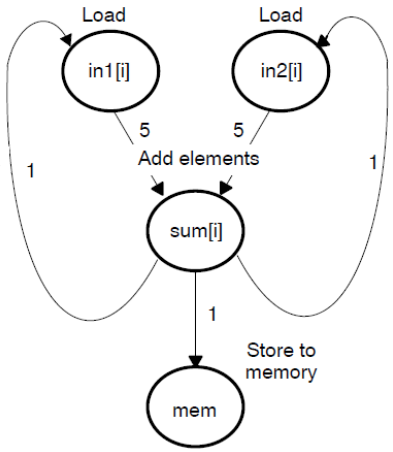
- *Use restrict keyword → indicates that the pointer is the only one*
- *Use -pm option, (program-level opt) → gives global access to compiler*
- *Use -mt option → allows compiler to assume no dependencies*

```
void vecsum(short *sum, short *in1, short *in2, unsigned int N)
{
  int i;

  for (i = 0; i < N; i++)
    sum[i] = in1[i] + in2[i];

}
```

short a[10], b[10]; // calling vecsum
vecsum(a, a, b, 10);



restrict is a type qualifier, load should not wait for store!

void vecsum(short * restrict sum, short * restrict in1, short * restrict in2, unsigned int N)

DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

3) Program Level Optimizations

–pm option → In addition to dependency detection

All source files go into one file

- *If a particular argument in a function always has the same value, the compiler replaces the argument with the value and passes the value instead of the argument.*
- *If a return value of a function is never used, the compiler deletes the return code in the function.*
- *If a function is not called, directly or indirectly, the compiler removes the function.*

4) Using Intrinsics

C6000 compiler provides intrinsics, special functions that map directly to inline C62x/C64x/C64x+/C67x instructions, to optimize C/C++ code quickly

DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

4) Using Intrinsics…

Complicated code can be replaced by HW mapped instructions

```
int sadd(int a, int b)
{
  int result;

  result = a + b;

  if (((a ^ b) & 0x80000000) == 0)
  {
    if ((result ^ a) & 0x80000000)
    {
      result = (a < 0) ? 0x80000000 : 0x7fffffff;
    }
  }
  return (result);
}
```

```
result = _sadd(a,b);
```

DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

4) Using Intrinsics…

Part of C6000 C/C++ Compiler intrinsics → spru198

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int **_abs**(int *src*); int **_labs**(long *src*);	**ABS**	Returns the saturated absolute value of src
int **_add2**(int *src1*, int *src2*);	**ADD2**	Adds the upper and lower halves of src1 to the upper and lower halves of src2 and returns the result. Any overflow from the lower half add does not affect the upper half add.
ushort & **_amem2**(void **ptr*);	**LDHU** **STHU**	Allows aligned loads and stores of 2 bytes to memory†
const ushort & **_amem2_const**(const void **ptr*);	**LDHU**	Allows aligned loads of 2 bytes from memory†
uint & **_amem4**(void **ptr*);	**LDW** **STW**	Allows aligned loads and stores of 4 bytes to memory†
const uint & **_amem4_const**(const void **ptr*);	**LDW**	Allows aligned loads of 4 bytes from memory†
double & **_amemd8**(void **ptr*);	**LDW/LDW** **STW/STW**	Allows aligned loads and stores of 8 bytes to memory†‡

DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

4) Using Intrinsics…

Part of C64X/C64X+ C/C++ Compiler intrinsics → spru198

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int **_abs2**(int *src*);	**ABS2**	Calculates the absolute value for each 16-bit value
int **_add4**(int *src1*, int *src2*);	**ADD4**	Performs 2s-complement addition to pairs of packed 8-bit numbers
long long & **_amem8**(void **ptr*);	**LDDW** **STDW**	Allows aligned loads and stores of 8 bytes to memory.
const long long & **_amem8_const**(const void **ptr*);	**LDDW**	Allows aligned loads of 8 bytes from memory.‡
double & **_amemd8**(void **ptr*);	**LDDW** **STDW**	Allows aligned loads and stores of 8 bytes to memory†‡ For C64x/C64x+ _amemd corresponds to different assembly instructions than when used with other C6000 devices; see Table 2–6.
const double & **_amemd8_const**(const void **ptr*);	**LDDW**	Allows aligned loads of 8 bytes from memory†‡
int **_avg2**(int *src1*, int *src2*);	**AVG2**	Calculates the average for each pair of signed 16-bit values
uint **_avgu4**(uint, uint);	**AVGU4**	Calculates the average for each pair of signed 8-bit values

DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

4) Using Intrinsics…

Part of C67X C/C++ Compiler intrinsics → spru198

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int **_dpint**(double *src*);	**DPINT**	Converts 64-bit double to 32-bit signed integer, using the rounding mode set by the CSR register
double **_fabs**(double *src*);	**ABSDP**	Returns absolute value of src
float **_fabsf**(float *src*);	**ABSSP**	
double **_mpyid** (int *src1*, int *src2*);	**MPYID**	Produces a signed integer multiply. The result is placed in a register pair.
double **_rcpdp**(double *src*);	**RCPDP**	Computes the approximate 64-bit double reciprocal
float **_rcpsp**(float *src*);	**RCPSP**	Computes the approximate 32-bit float reciprocal

DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

5) Wider memory access for smaller data types

Maximizing the throughput using single load/store to access more data elements of smaller size

```
void vecsum4(short * restrict sum, short * in1, short * restrict in2, unsigned int N)
{
    int i;
    #pragma MUST_ITERATE (10);
    for (i = 0; i < N; i+=2)
    _amem4(&sum[i]) = _add2(_amem4_const(&in1[i]), _amem4_const(&in2[i]));
}
```

N must be (made) even

Don't use the old style of pointer casting …

```
void vecsum4_old(short * restrict sum, short * restrict in1, short * restrict in2, unsigned int N)
{
    int i;
    const int *restrict i_in1 = (const int *)in1;
    const int *restrict i_in2 = (const int *)in2;
    int *restrict i_sum = (int *)sum;
    #pragma MUST_ITERATE (10);
    for (i = 0; i < (N/2); i++)
    i_sum[i] = _add2(i_in1[i], i_in2[i]);
}
```

```
int test(short *x)
{
    int t;

    *x = 0;
    t = *(int *)x);
    return t;
}
```

```
t = _amem4(x);
```

DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

5) Wider memory access for smaller data types…

p can be of any type just to fill the wider memory size …

C Compiler Intrinsic		Description
_memd8(p)		unaligned access of double beginning at address p (existing intrinsic)
_memd8_const(p)	*long long or double*	unaligned access to const double beginning at address p
_amemd8(p)		aligned access of double beginning at address p
_amemd8_const(p)		aligned access to const double beginning at address p
_mem4(p)		unaligned access of unsigned int beginning at address p (existing intrinsic)
_mem4_const(p)		unaligned access to const unsigned int beginning at address p
_amem4(p)		aligned access of unsigned int beginning at address p
_amem4_const(p)		aligned access to const unsigned int beginning at address p
_mem2(p)		unaligned access of unsigned short beginning at address p
_mem2_const(p)		unaligned access to const unsigned short beginning at address p
_amem2(p)		aligned access of unsigned short beginning at address p
_amem2_const(p)		aligned access to const unsigned short beginning at address p

DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

5) Wider memory access for smaller data types…

Non-aligned memory access for C64 X and C64X+ only → two times slower

```
void vecsum4a(short * restrict sum, short * in1, short * restrict in2, unsigned int N)
{
    int i;
    #pragma MUST_ITERATE (10);
    for (i = 0; i < N; i+=2)
    _mem4(&sum[i]) = _add2(_mem4_const(&in1[i]), _mem4_const(&in2[i]));
}
```

8 byte access:

```
void vecsum4_C64(short * restrict sum, short * in1, short * restrict in2, unsigned int N)
{
    int i;
    int a3_a2, a1_a0, b3_b2, b1_b0, c3_c2, c1_c0;

    #pragma MUST_ITERATE (5)
    for (i = 0; i < N; i+=4)
    {
        a3_a2 = _hi(_amemd8_const(&in1[i]));
        a1_a0 = _lo(_amemd8_const(&in1[i]));
        b3_b2 = _hi(_amemd8_const(&in2[i]));
        b1_b0 = _lo(_amemd8_const(&in2[i]));
        c3_c2 = _add2(b3_b2, a3_a2);
        c1_c0 = _add2(b1_b0, a1_a0);
        _amem8(&sum[i]) = _itoll(c3_c2, c1_c0);
    }
}
```

DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

5) Wider memory access for smaller data types…

Non-aligned only on C64X and C64X+

128 bits / cycle memory BW for aligned data

64 bits / cycle memory BW for non-aligned data → used when there is a good reason

- *Generic routines which cannot impose alignment*

→ e.g. memcpy()

- *Single sample algorithms which update their input or output pointers by only one sample → e.g. adaptive filters*
- *Nested loop algorithms*

→ e.g. 2D convolution

- *Routines which have an irregular memory access pattern, or whose access pattern is data-dependent and not known until run time*

→ e.g. motion compensation in image processing

DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

5) Wider memory access for smaller data types…

Spru187.pdf

To tell the compiler about the loop trip count:

#pragma MUST_ITERATE(Min trip count, Max trip count, multiple of)

To tell the compiler to align arrays of any memory bank on double word boundaries:

#pragma DATA_MEM_BANK (pointer, memory bank number)

0 means double word alignment 4k+0(bank0), 4k+1(bank1), …, 4k+3(bank3)

To tell the compiler about the location of the variable… sections, defined in linker file

#pragma DATA_SECTION (symbol , " section name ");

```
#pragma DATA_MEM_BANK (x, 2);
short x[100];
#pragma DATA_MEM_BANK (z, 0);
#pragma DATA_SECTION (z, ".z_sect");
short z[100];
void main()
{
    #pragma DATA_MEM_BANK (y, 2);
    short y[100];
    ...
}
```

DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

5) Wider memory access for smaller data types…

_nassert(boolean) intrinsic provides alignment information to the compiler
Claims the argument is true …

Example : _nassert(((int)sum & 0x3) == 0); // generates no code…

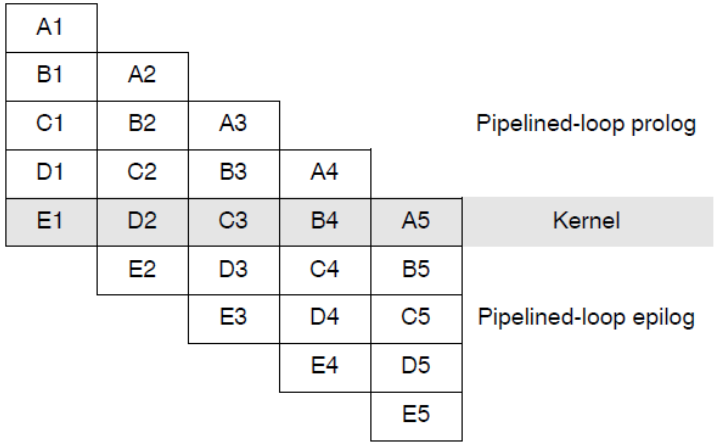
With -pm option it might be useless → Compiler already has enough information!

6) Software pipelining

Scheduling the loop instructions to execute instructions of different iterations together

Let's look at loops more carefully

Loop Qualification



DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

6) Software pipelining…

- *Trip count : # of loop iterations*

Minimum trip count: sometimes by compiler, or MUST_ITERATE pragma

- *Minimum safe trip count: minimum trip count to safely execute the software pipelined version of the loop*

If min-trip-count < min-safe-trip count → Redundant loop! (next slide)

- *Loop reversal: count down loops are more efficiently software pipelined*

Original code

```
for (i = 0; i < N; i++) /* i = trip counter, N = trip count */
```

Optimized code

```
for (i = N; i != 0; i--) /* Downcounting trip counter */
```

DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

6) Software pipelining…

- *Eliminating the redundant loop*

If compiler cannot determine the minimum trip count → makes two versions

Non-pipelined to be executed if min<min-safe else pipelined …

→ A redundant loop will be required : reduces performance, increases code size

- *Use –mhn : n-byte memory pad will be assumed by the compiler, get prolog and epilog in the loop*
- *Provide info to the compiler when possible, must iterate and _nassert*

- *Loop unrolling*

Expanding the small loops so that some iterations appear in the loop body

Either compiler does it, or programmer suggests by #pragma UNROLL(n) (n times), or programmer does it …

Compiler performs SW pipelining only on inner loops

Unrolling makes larger inner loops…

DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

6) Software pipelining…

- *Loop unrolling… Increases code size*

```
void vecsum4(short *restrict sum, const short *restrict in1,
const short *restrict in2, unsigned int N)
{
  int i;
  #pragma MUST_ITERATE (10);
   for (i = 0; i < (N/2); i++)
       {
        _amem4(&sum[i]) = _add2(_amem4_const(&in[1]), _amem4_const(&in2[i]));
       }
 }
```

```
void vecsum6(int *restrict sum, const int *restrict in1, const int *restrict
in2, unsigned int N)
{
  int i;
  int sz = N >> 2;

  #pragma MUST_ITERATE (10);   5

  for (i = 0; i < sz; i++)
  {
     sum[i]   = _add2(in1[i],   in2[i]);
     sum[i+sz] = _add2(in1[i+sz], in2[i+sz]);
  }
}
```

DSP Mapping, Coding, Optimization…

C refining and compiler guideline…

6) Software pipelining…

What Disqualifies a loop from being SW pipelined

- *Having a register value being live too long*
- *If the loop has complex condition code, needing 5 (6 for C64) condition registers*
- *Having function calls (use inline functions)*
- *Having conditional break or early exit in a loop*
- *Increasing trip counter (use loop reversal if compiler cannot)*
- *Modifying trip counter in the loop*
- *Big loop body that uses more than DSP registers*

Compiler Feedback

*-k –mw options: -k is to make the *.asm files and –mw to have feedback in there*

Understanding the feedback is important

DSP Mapping, Coding, Optimization…

Compiler Feedback… (chap 4, spru198)

First part

```
;*      Known Minimum Trip Count          : 2
;*      Known Maximum Trip Count          : 2
;*      Known Max Trip Count Factor       : 2
```

Second part

```
;*      Loop Carried Dependency Bound(^) : 4
;*      Unpartitioned Resource Bound     : 4
;*      Partitioned Resource Bound(*)    : 5
;*      Resource Partition:
;*                                  A-side   B-side
;*      .L units                      2        3
;*      .S units                      4        4
;*      .D units                      1        0
;*      .M units                      0        0
;*      .X cross paths                1        3
;*      .T address paths              1        0
;*      Long read paths               0        0
;*      Long write paths              0        0
;*      Logical  ops (.LS)            0        1      (.L or .S unit)
;*      Addition ops (.LSD)           6        3      (.L or .S or .D unit)
;*      Bound(.L .S .LS)              3        4
;*      Bound(.L .S .D .LS .LSD)      5*       4
```

→ Write to read

=ceil((.L + .S + .LS) / 2)

= ceil((.L + .S + .D + .LS + .LSD) / 3)

DSP Mapping, Coding, Optimization…

Compiler Feedback… (chap 4)

Third part

Iteration Interval : Cycles between successive initiations of the loop

```
;*      Searching for software pipeline schedule at ...
;*         ii = 5  Register is live too long
;*         ii = 6  Did not find schedule
;*         ii = 7  Schedule found with 3 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 1
;*
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad : 2 bytes
;*
;*      Minimum safe trip count     : 2
```

ii increments until a schedule is found

Removes prolog by increasing the loop trip count… -mh option

ii: always the maximum of loop carried dependency bound and the partitioned resource bound

Did not find Schedule parameters:

- ***Regs Live Always***
- ***Max Regs Live***
- ***Max Cond Regs***

Example:

Regs Live Always : 1/5 (A/B–side)
Max Regs Live : 14/19
Max Cond Regs Live : 1/0

DSP Mapping, Coding, Optimization…

Compiler Feedback…

More info

```
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 3  Schedule found with 5 iterations in parallel
;*
;*      Register Usage Table:
;*          +---------------------------------+
;*          |AAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBB|
;*          |0000000000111111|0000000000111111|
;*          |0123456789012345|0123456789012345|
;*          |----------------+----------------|
;*       0: | * *** **       |**  ******      |
;*       1: | * ******       |**  *****       |
;*       2: | * *** **       |*   ******      |
;*          +---------------------------------+
;*
;*      Done
;*
;*      Epilog not removed
;*      Collapsed epilog stages     : 0
;*
;*      Prolog not entirely removed
;*      Collapsed prolog stages     : 3
;*
;*      Minimum required memory pad : 0 bytes
;*
;*      For further improvement on this loop, try option -mh16
```


Sharif University of Technology, EE Department, Digital Signal Processors Course Notes, imangh@sharif.edu

DSP Mapping, Coding, Optimization…

Compiler Feedback…

Loop disqualification messages (format as appeared in spru198 chapter 5)

Bad Loop Structure

Description

This error is rare and may stem from the following:

- An asm statement inserted in the C code inner loop.
- Parallel instructions being used as input to the linear assembly optimizer.
- Complex control flow such as GOTO statements, breaks, nested if statements, if-else statements, and large if statements.

Solution

Remove any asm statements, complex control flow, or parallel instructions as input to linear assembly.

Loop Contains a Call

….

Too Many Instructions

….

DSP Mapping, Coding, Optimization…

See code examples enclosed … Addressing.rar, showing all modes of addressing

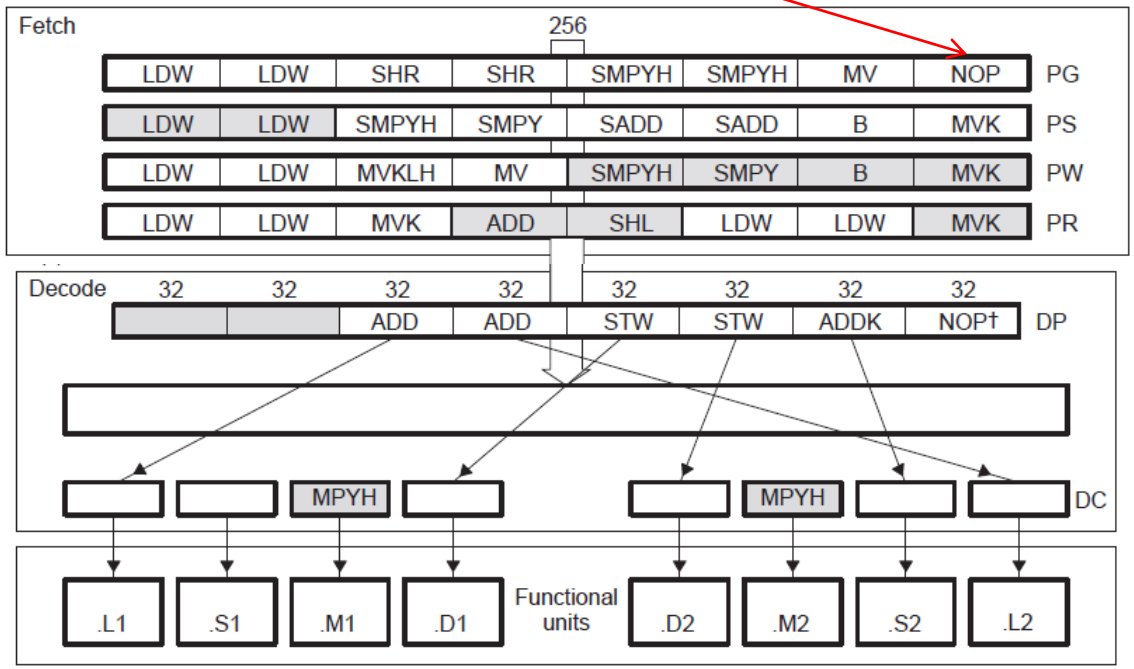
And division.rar for next slides … Also showing :

1) How to init memory in asm file 2) How to return double data from linear assembly files

3) How to write asm in C 4) How to pass a variable back to C…

Pipeline: Fetch Packet, Execution Packet …

VelociTI architecture does not allow execute packets to cross-fetch packet boundaries,
VelociTI.2 eliminates this restriction by instruction packing



DSP Mapping, Coding, Optimization…

To continue:

1) Go for spru198 chapter 3 for tutorial example, lesson 1-3

C62 version in the class → C64 version as an Exercise

Exercise-3

2) Move to TI C6000 ROM, chapters 7 and 11

Added missing stuff like linear assembly directives from spru198 chapter 5

3) And TI C6000 ROM, chapters 12

Hand optimization → added from different text books

4) And TI C6000 ROM, chapters 4

DSP BIOS, Simulator examples added

5) Same stuff, Updated for newer DSPPs in Optimization Workshop

Sharif University of Technology, EE Department, Digital Signal Processors Course Notes, imangh@sharif.edu