Signal Processors Lecture Notes, Pt 5

Institut für Technische Informatik, DI Dr. Eugen Brenner

Signal Processors, 448.032

Chapter Overview

5. Code Optimization

Introduction

Motivation

- Meeting design constraints for code
  - . . . Size
  - . . . Speed
  - . . . Interruptibility
- Choosing the right architecture
  - . . . for MIPS
  - . . . Peripherals
  - . . . Data type
  - . . . Functional units
  - . . . Memory

Code Optimization Procedure

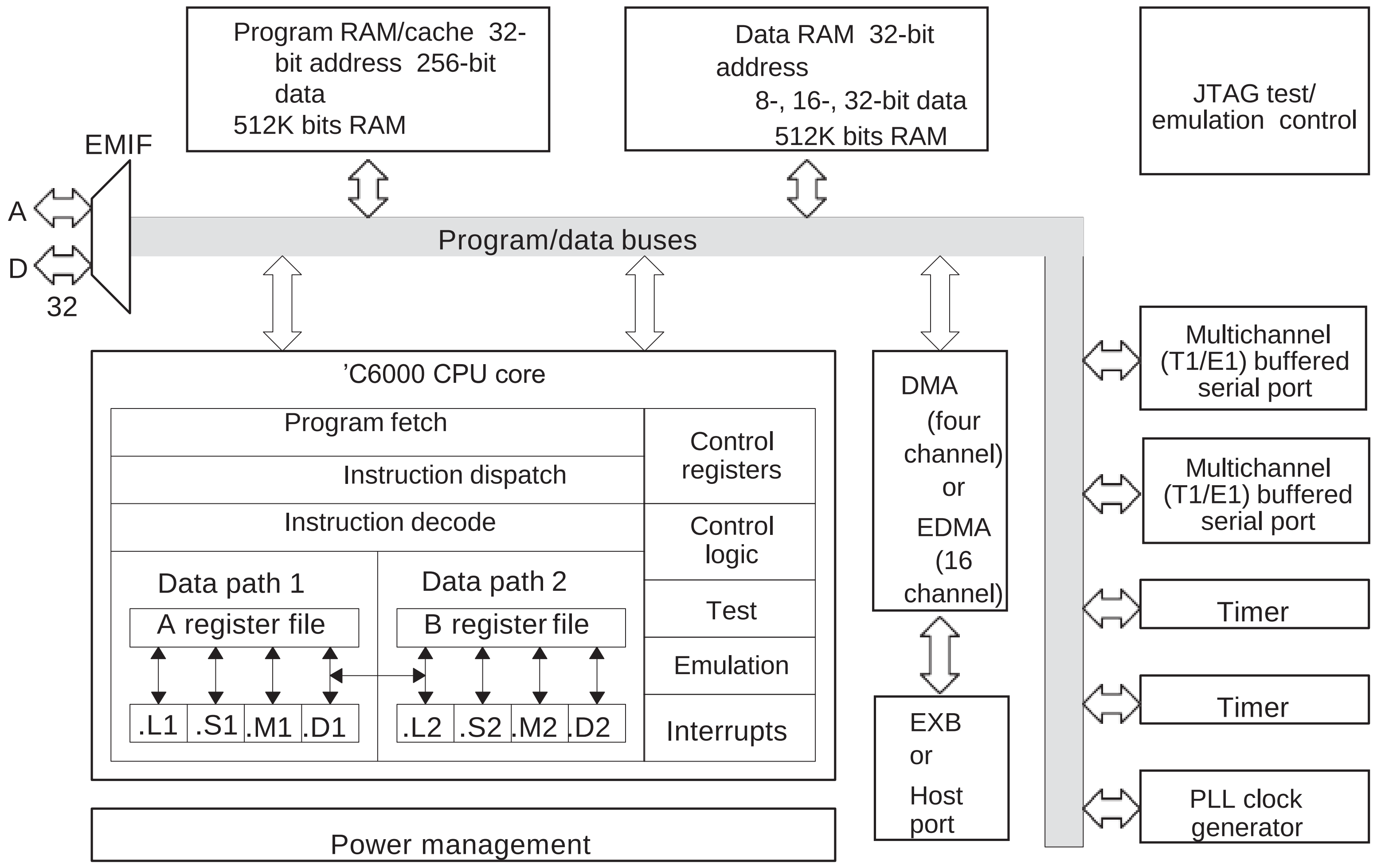
1. Write algorithm in C and verify it
   - Give the compiler some hints for optimization
2. Write algorithm in Linear Assembly code
3. Write code in Scheduled Assembly code
   - Fill delay slots
   - Maximize hardware utilization – ILP
   - Use word-wide optimization (packed data processing)
   - Use software-pipelining

Example: Dot-Product

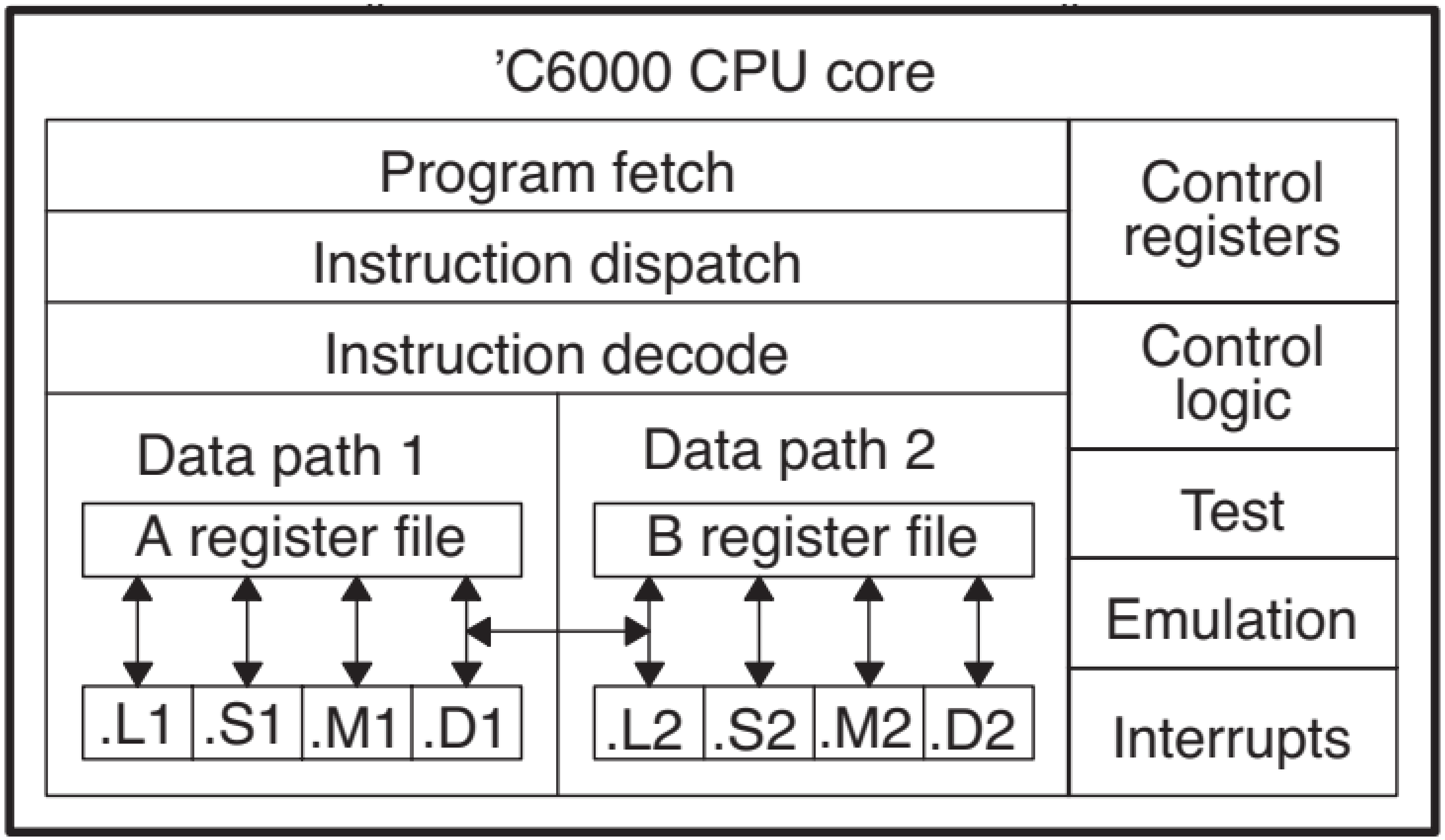
```
short vec_m[256], vec_n[256];

int dotp(short *vec_m, short *vec_n, int count) {
    int i, sum = 0
    for (i = 0; i < count; i++) {
        sum += *vec_n++ * *vec_m++;
    }
}
```

TMS320C6000 Architecture

C6000 Architecture Overview



C6000 Processor Core



C6000 Summary

- 2 symmetrical data paths
  - Register file
  - 4 functional units
    - L, .M, .S, .D
- 1 cross path in each direction
- Data widths
  - 16, 32, and 40-bits
  - SIMD instructions (ADD2, ADD)
- Instructions have different latencies (delay slots!)
- Conditional execution
- 256-bit VLIW

Assembler: Important Basics

- 3-address architecture: ADD A1, A2, A3
- Instruction set
  - Arithmetic: ADD, ABS, MPY, Zero, ...
  - Logical: AND, OR, XOR, SHL, ...
  - Bit operations: CLR, EXT, SET, ...
  - Data management: LDB, MV, STB, ...
  - Program control: B, NOP, IDLE, ...
- Packed operations: DOTP2, DOTPU4, SADD2, ...
- Addressing modes: immediate, register, indirect
- Reference: SPRU733A

Assembler Instruction

```
[label [:]] [[register]] mnemonic [unit specifier]
             [operand list] [;comment]
```

e.g.:

```
loop:    mpy .m1x A2, B5, A4

         ldh .d1 *A8++,  A2
      || ldh .d2 *B9++,  B5

      [b0] b loop
```

Function Calling Conventions

```
int func1(int a, int b, int c);
A4         A4     B4     A6

int func2(int a, float b, int *c, struct A d, float e, int f, int g);
A4         A4     B4       A6      B6           A8        B8     A10

int func3(int a, double b, float c, long double d);
A4         A4     B5:B4     A6        B7:B6

// NOTE: The following function has a variable number of arguments

int vararg(int a, int b, int c, int d, ...);
A4          A4     B4     A6     stack ...

struct A func4(int y);
A3              A4
```

Delay Slots & Latency

- Latency: Total cycles an instruction requires
- Single cycle instructions

  All, except . . .

  - Multiply (`MPY/SMPY`): 1 delay slot
  - Load (`LDB/H/W`): 4 delay slots
  - Branch (`B`): 5 delay slots

Insert *NOPs* or other instructions to fill delay slots!

NOP: Not Optimized Properly ;-)

Linear Assembly Code

- Assembler code that is “un-scheduled” and has not been register-allocated
- Benefits
  - Easy integration in C-code
  - Symbolic variable names
  - Ignore pipeline issues (*no delay slots*)
  - No parallel instructions
  - Optimized by assembly optimizer
- File extension: .sa

Linear Assembly – Example

```
_vecsum:     .cproc pn, count
             .reg n, sum

             zero   sum
loop:        ldh    *pn++, n
             add    sum, n, sum
             sub    count, 1, count
    [count]  b      loop
             .return sum
             .endproc
```

Scheduled Assembler

- Manually “schedule” assembler instructions
  - Schedule machine instructions to the DSP pipeline
    - Consider delay slots!
    - Exploit ILP
  - Provides highest flexibility
  - Requires expert skills
  - Requires detailed knowledge of the hardware
  - No compiler optimization

Optimize C Code

Compiler Options

- Turn off full source-level debug options (-g)
- Turn on Optimization (-o0 . . . -o3)

Memory Aliasing

```
void fcn(*in, *out) {
   LDW *in++, A0
   ADD A0, 4, A1
   STW A1, *out++
}

fcn(*data, *data + 1);
```

Memory Aliasing

- Solutions
  - Compiler solves aliasing on its own (conservative)
  - Program level optimization (-pm)
  - No bad aliasing option (-mt)
  - Restrict keyword

```
(void fcn(short *restrict in, *out))
```

Intrinsics

- Intrinsic operations are automatically inlined into the code
- Access hardware functionality which is unsupported by C
- Think of it as specialized library function
- `#include <c6x.h>`
- Can use C variable names instead of register names
- Are compatible with C environment
- Adhere to C’s function call syntax

```
add2(), mpy()
mpylh(), ...
```

Intrinsics – Examples

```
int x1, x2, y;
y = _sadd(x1, x2);

short a[50], b[50];
y = _add2(*(int *)a, *(int *)b);
```

Intrinsics – Example DOTP2

```
for (i = 0; i < len; i += 4){                       SUB || B
   a3_a2 = _hill(_amemd8_const(&a[i]));  || LDDW
   a1_a0 = _loll(_amemd8_const(&a[i]));
   b3_b2 = _hill(_amemd8_const(&b[i]));  || LDDW
   b1_b0 = _loll(_amemd8_const(&b[i]));
   /*Perform dot-products on pairs of
     elements, totaling the results in
     the accumulator. */                         || DOTP2
   sum_high+= _dotp2(a3_a2, b3_b2);              || ADD

   sum_low += _dotp2(a1_a0, b1_b0);              || DOTP2
}                                                    || ADD
```

Provide Additional Information: Pragmas

- The more information the compiler has, the better job it can do
  - Otherwise generate code for the worst case!
- #pragma ... can be included into the C code
  - #pragmas are ignored by other C compilers if they are not supported

Pragma: Unroll

```
#pragma UNROLL(2)
for (i = 0; i < count; i++) {
    sum += a[i] * x[i];
}
```

- #pragma must come right before the loop
- Tells the compiler to unroll the loop twice
- Compiler will generate extra code to handle the case that the count is odd

Pragma: Must Iterate

```
#pragma UNROLL(2)
#pragma MUST_ITERATE(10, 100, 2)
for (i = 0; i < count; i++) {
   sum += a[i] * x[i];
}
```

- Gives the compiler information about the loop count
- If you break your promise, you might break your code

Pragma: Data align

- Tell the compiler how to create variables
  - Support packed data processing

```
 #pragma DATA_ALIGN(a, 8)
short a[256] = { 1, 2, 3, ... }
```

Compile-time Checks

```
 #pragma DATA_ALIGN(myVar, 4)
_nassert(myVar & 0x3 == 0);
for (i = 40; i > 0; i--) {
    ...
```

- `_nassert` statement evaluated at compile time
- Allow more aggressive optimization
- `_nassert` used for libraries
  - -pm only works for source code
  - Library code cannot benefit from many optimizations
  - Tells the compiler that symbols are aligned

Assembler Optimization

Example: Dot-Product

```
short vec_m[256], vec_n[256];

int dotp(short *vec_m, short *vec_n, int count){
   int i, sum = 0
   for (i = 0; i < count; i++){
      sum += *vec_n++ * *vec_m++;
   }
}
```

Dot-Product in Linear Assembler

```
_dotp:    .cproc pm, pn, count
          .reg m, n, prod, sum
          ZERO sum

loop:     LDH *pm++, m
          LDH *pn++, n
          MPY m, n, prod
          ADD prod, sum, sum
          SUB count, 1, count
 [count]  B loop
          .return sum
          .endproc
```

Dot-Product in Scheduled Assembler

```
_dotp:     MV    A6, A1
           MV    A4, A6
           ZERO  A4

loop:      LDH   *A6++, A5
           NOP   4
           LDH   *B4++, A7
           NOP   4
           MPY   A5, A7, A9
           NOP   1
           ADD   A9, A4, A4
           ADDK  -1, A1
      [A1] BNOP  loop, 5
           BNOP  B3, 5
```

Dot-Product – Fill Delay Slots (I)

```
_dotp:     MV    A6, A1
           MV    A4, A6
           ZERO  A4

loop:      LDH   *A6++, A5
           LDH   *B4++, A7
           ADDK  -1, A1
           NOP   3
           MPY   A5, A7, A9
           NOP   1
           ADD   A9, A4, A4
      [A1] BNOP  loop, 5
           BNOP  B3, 5
```

Dot-Product – Fill Delay Slots (II)

```
_dotp:     MV     A6, A1
           MV     A4, A6
           ZERO   A4

loop:      LDH    *A6++, A5
           LDH    *B4++, A7
           ADDK   -1, A1
      [A1] B      loop
     [!A1] B      B3
           NOP    1
           MPY    A5, A7, A9
           NOP    1
           ADD    A9, A4, A4
           NOP    1
```

Dot-Product – Parallel Instructions

```
_dotp:      MV     A6, A1
         || MV     A4, A6
         || ZERO   A4

loop:       LDH    *A6++, A5
         || LDH    *B4++, A7
            ADDK   -1, A1
       [A1] B      loop
      [!A1] B      B3
            NOP    1
            MPY    A5, A7, A9
            NOP    1
            ADD    A9, A4, A4
            NOP    1
```

Optimization Summary

- No optimization: 20 cycles × 256 iterations

  5120 cycles

- Fill delay slots: 9 cycles × 256 iterations

  2304 cycles

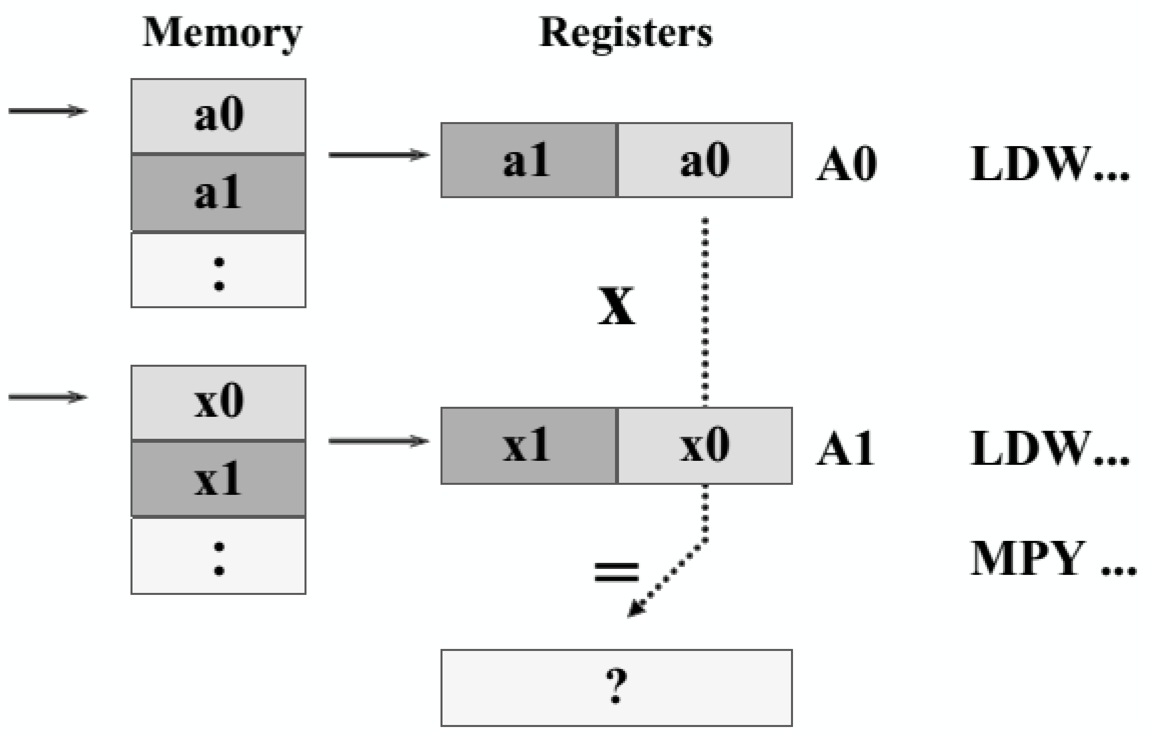
- Parallel instructions: 8 cycles × 256 iterations

  2048 cycles

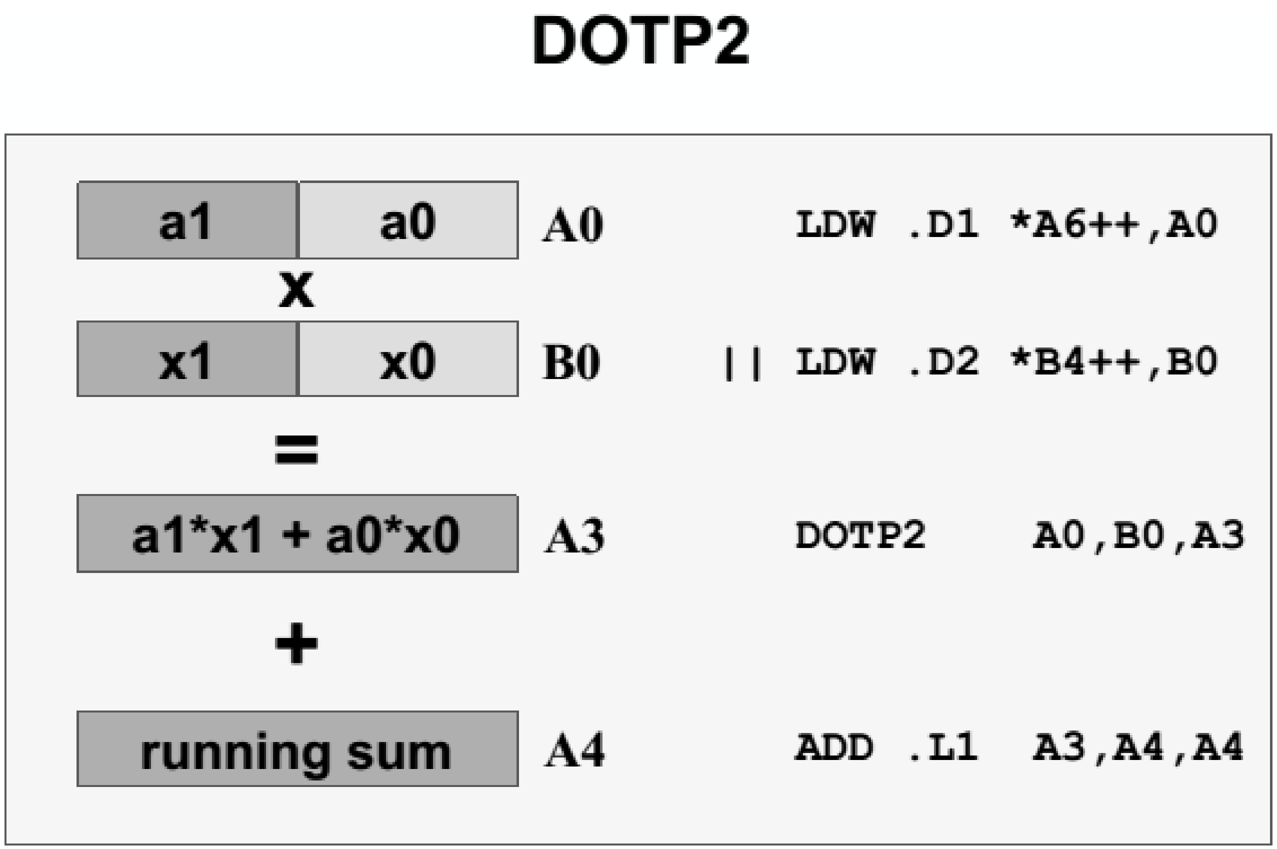
Word-Wide Optimization

- We multiply `shorts` (16-bit)
- Processor data width: 32-bit
- Approach: use the whole bus, SIMD
- Instructions
  - ADD2, SUB2
  - SADD2
  - ABS2
  - MPYH, MPYHL, MPYLH (signed/unsigned variants)
  - DOTP2
  - . . .

Word-Wide Optimization

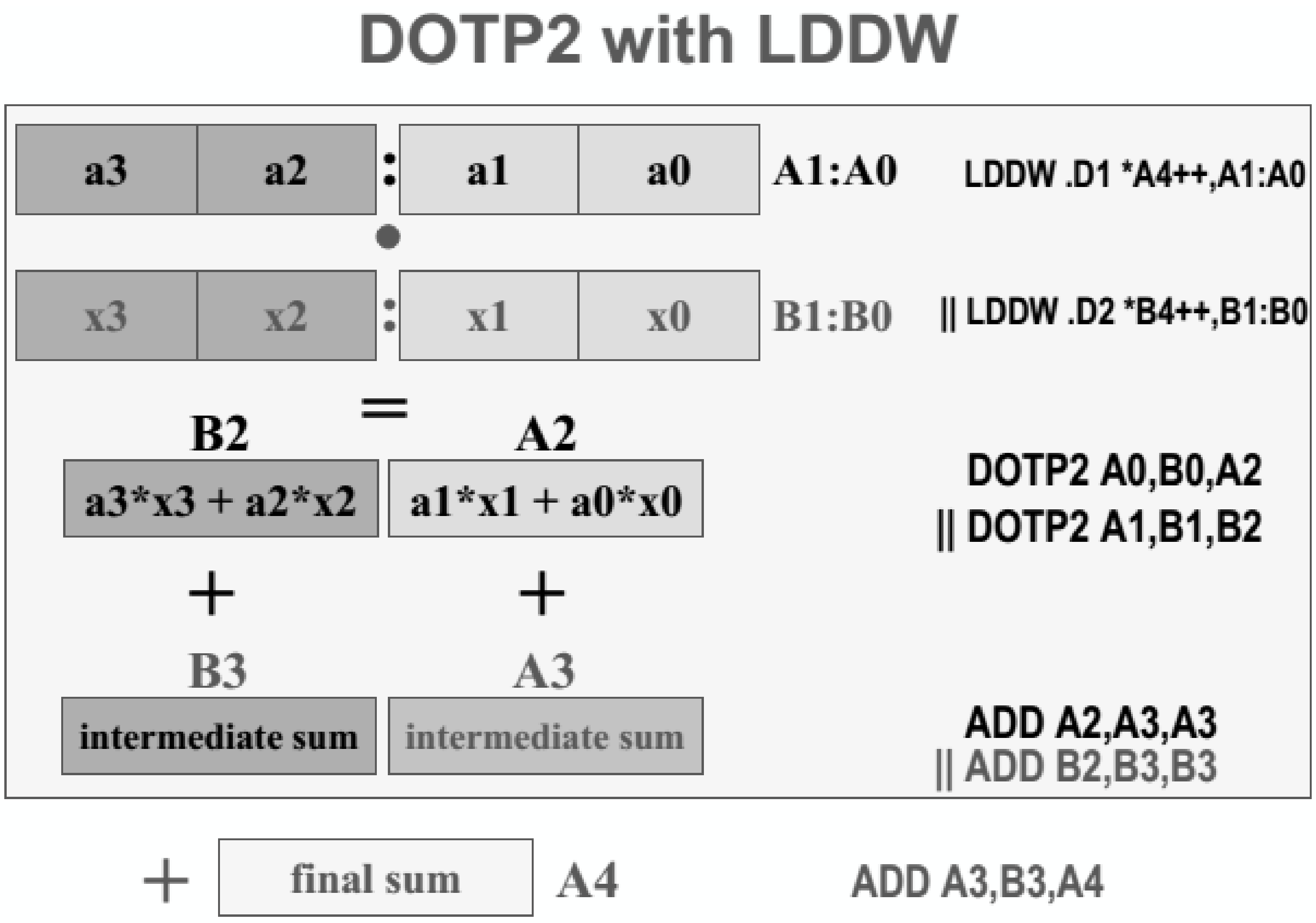


Word-Wide Optimization



```
      LDW  .D1  *A6++,A0
   || LDW  .D2  *B4++,B0
      DOTP2       A0,B0,A3
      ADD  .L1    A3,A4,A4
```

Process Four Entries at once



Word-Wide Optimization (Double Word)

```
_dotp:      MV     A6, A1
         || ZERO   A7
         || ZERO   B7
loop:       LDDW   *A4++, A9:A8
         || LDDW   *B4++, B9:B8
         || ADD    -4, A1, A1
            NOP    3
   [A1]     B      loop
            DOTP2  A9, B9, A5
         || DOTP2  B8, A8, B5
  [!A1]  || B      B3
            NOP    3
            ADD    A5, A7, A7
         || ADD    B5, B7, B7
            ADD    A7, B7, A4
```

Optimization Summary

- No optimization: 20 cycles × 256 iterations

  5120 cycles

- Fill delay slots: 9 cycles × 256 iterations

  2304 cycles

- Parallel instructions: 8 cycles × 256 iterations

  2048 cycles

- Word-wide optimization: 10 cycles × 64 iterations

  640 cycles

Can we do better?

Software Pipelining

Motivation for Software Pipelining

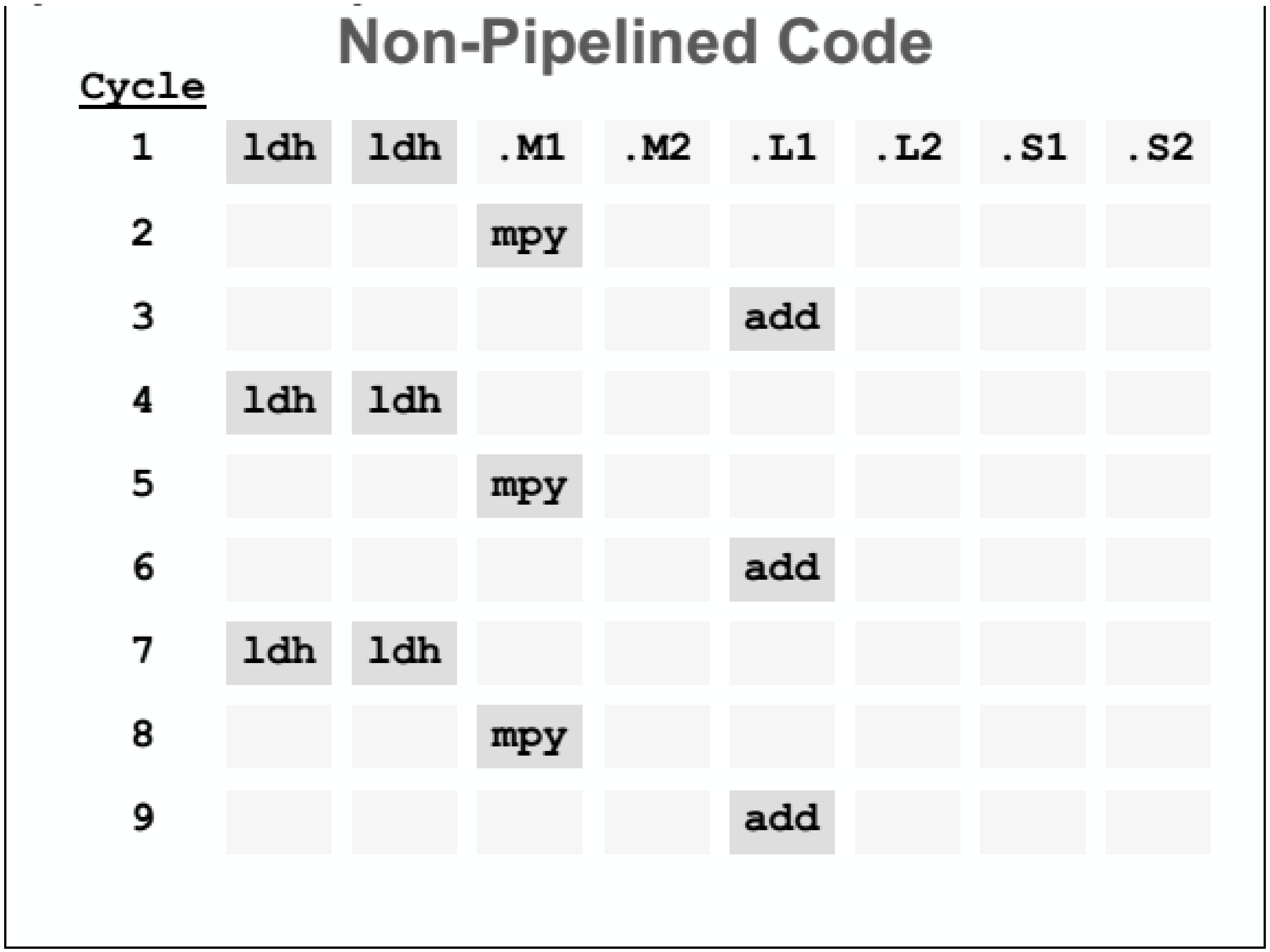
- Highly optimized loop-code
  - Implement parallel instructions
  - Fill delay slots
  - Maximize usage of functional units
- Why learn software pipelining?
  - Understand how tools create optimal code
    - Read the tool's output
    - Check tools efficiency
  - Write hand-optimized assembly
  - Know how it works!

Simple Example

```
      LDH ...
   || LDH ...
      MPY ...
      ADD ...
```

loop 5 times

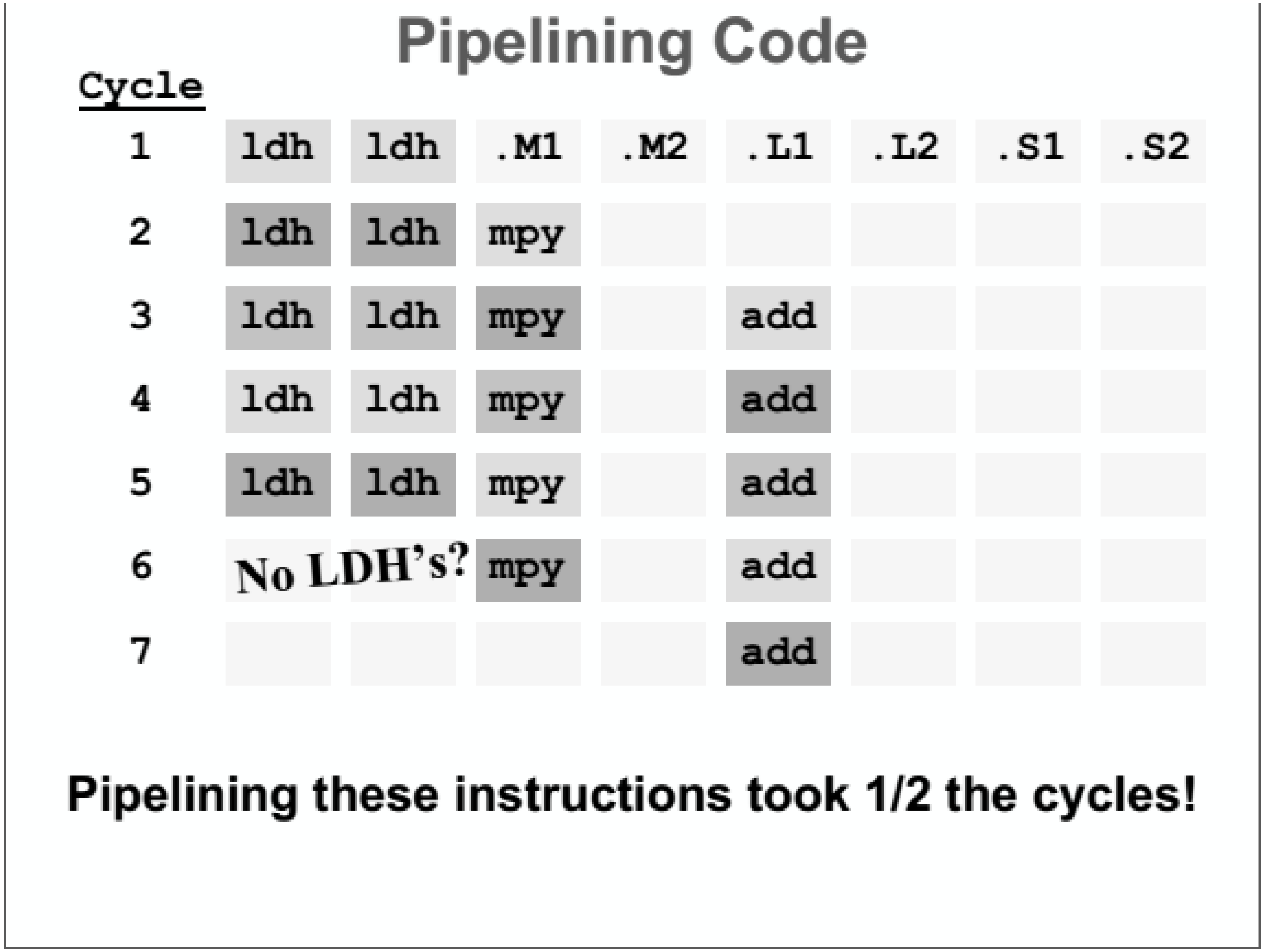
Simple Example



Non-Pipelined Code

Cycle								
1	ldh	ldh	.M1	.M2	.L1	.L2	.S1	.S2
2			mpy					
3					add			
4	ldh	ldh						
5			mpy					
6					add			
7	ldh	ldh						
8			mpy					
9					add			

Simple Example



Pipelining Code

Cycle								
1	ldh	ldh	.M1	.M2	.L1	.L2	.S1	.S2
2	ldh	ldh	mpy					
3	ldh	ldh	mpy		add			
4	ldh	ldh	mpy		add			
5	ldh	ldh	mpy		add			
6	No LDH's?		mpy		add			
7					add			

Pipelining these instructions took 1/2 the cycles!

SW Pipelining Procedure

1. Write Linear Assembly Code
2. Create dependency graph / data flow graph
3. Allocate registers & functional units
4. Create scheduling table
5. Translate scheduling table to code

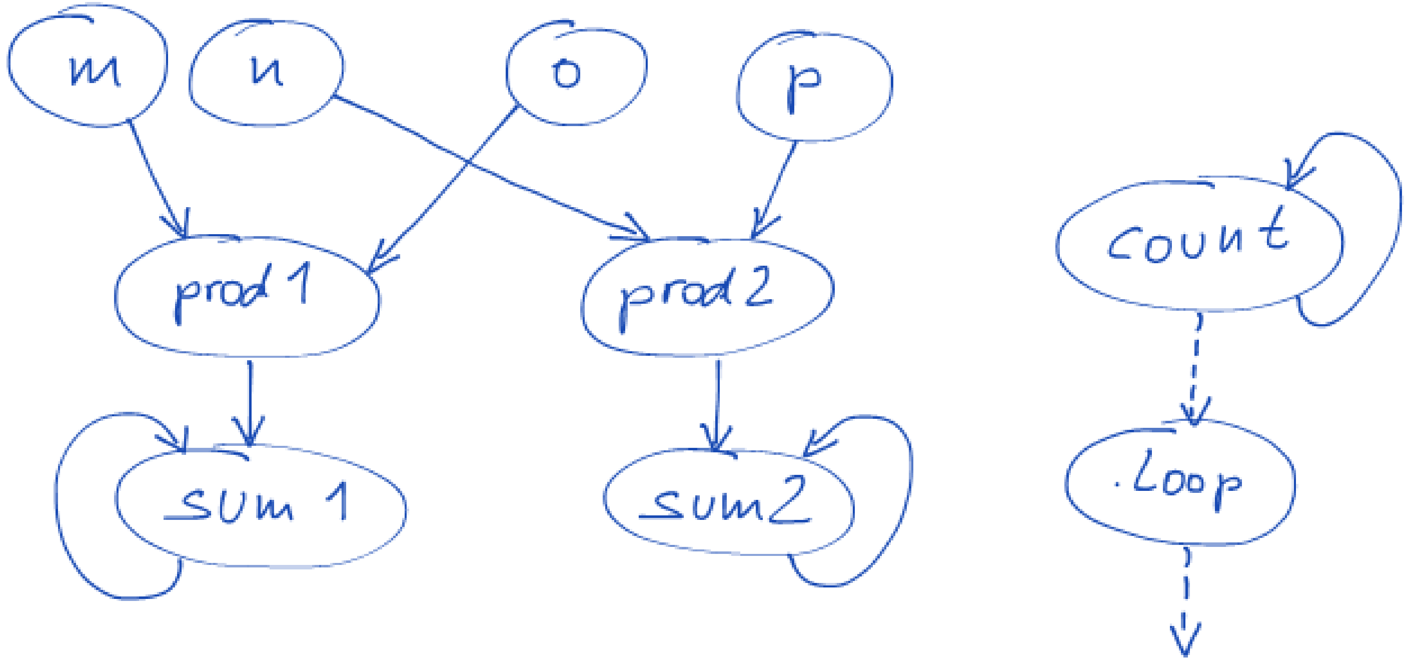
1. Dot-Product in C . . .

```
short vec_m[256], vec_n[256];

int dotp(short *vec_m, short *vec_n, int count){
   int i, sum = 0
   for (i = 0; i < count; i++){
      sum += *vec_n++ * *vec_m++;
   }
}
```

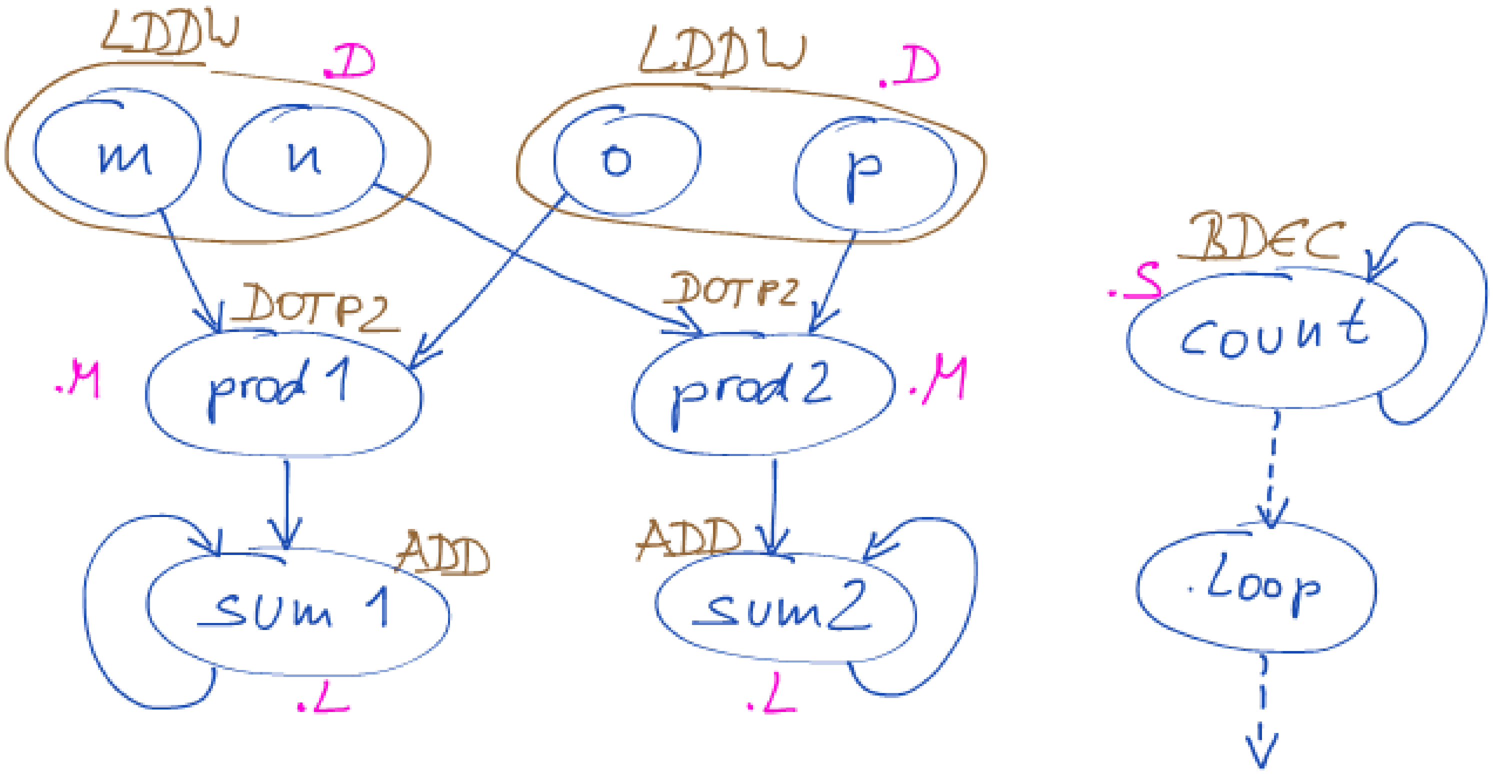
1. Dot-Product in C and Linear Assembler

```
_dotp:      .cproc  pm, pn, count
            .reg    m:n, o:p, prod1, prod2, sum1, sum2
            zero    sum
            SHR     count, 2, count
            SUB     count, 1, count
loop:       LDDW    *pm++, m:n
            LDDW    *pn++, o:p
            DOTP2   m, o, prod1
            DOTP2   n, p, prod2
            ADD     prod1, sum1, sum1
            ADD     prod2, sum2, sum2
    [count] BDEC    loop, count
            ADD     sum1, sum2, sum1
            .return sum1
            .endproc
```

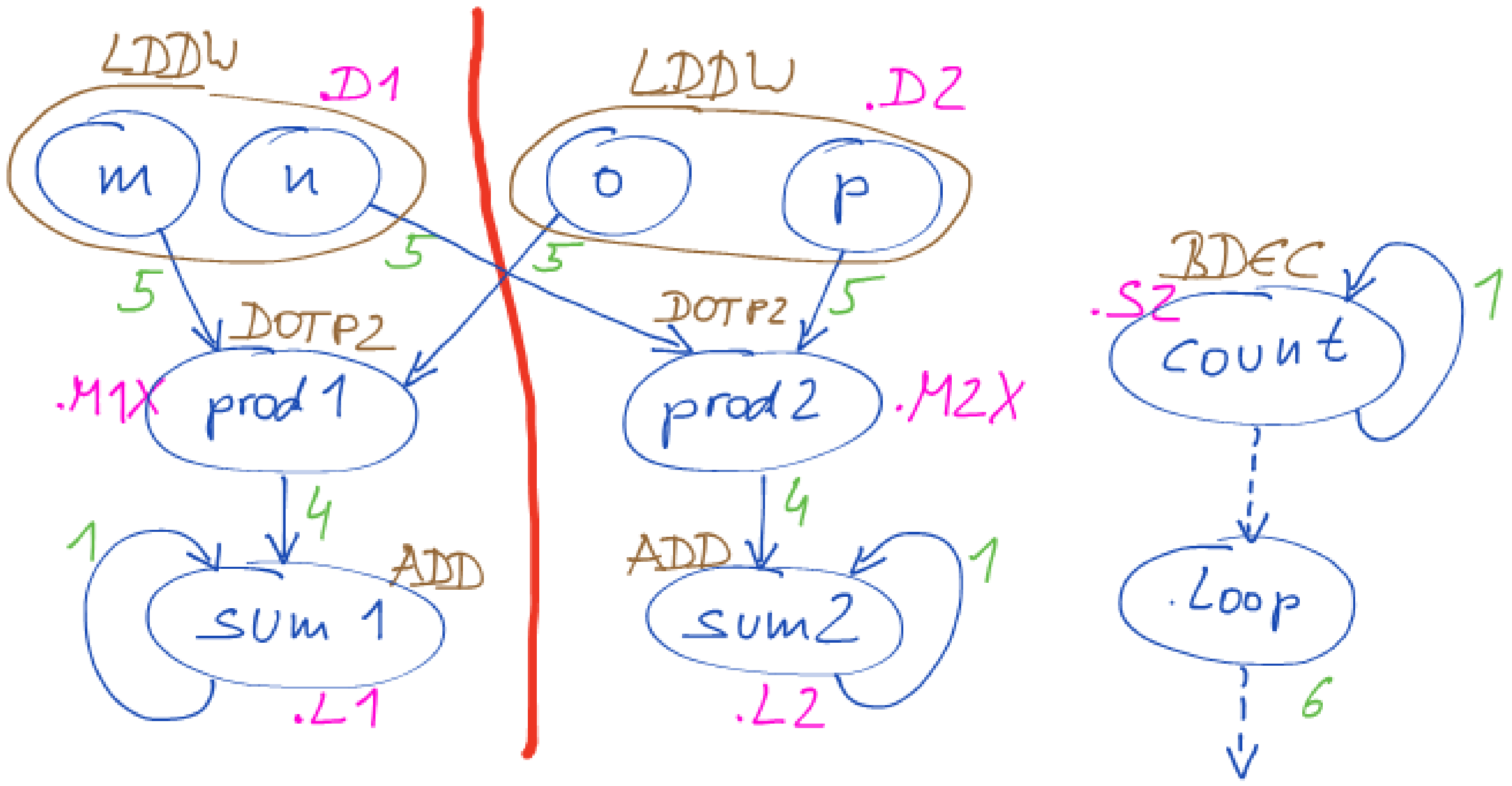
2. Data Flow Graph



3. Functional Units



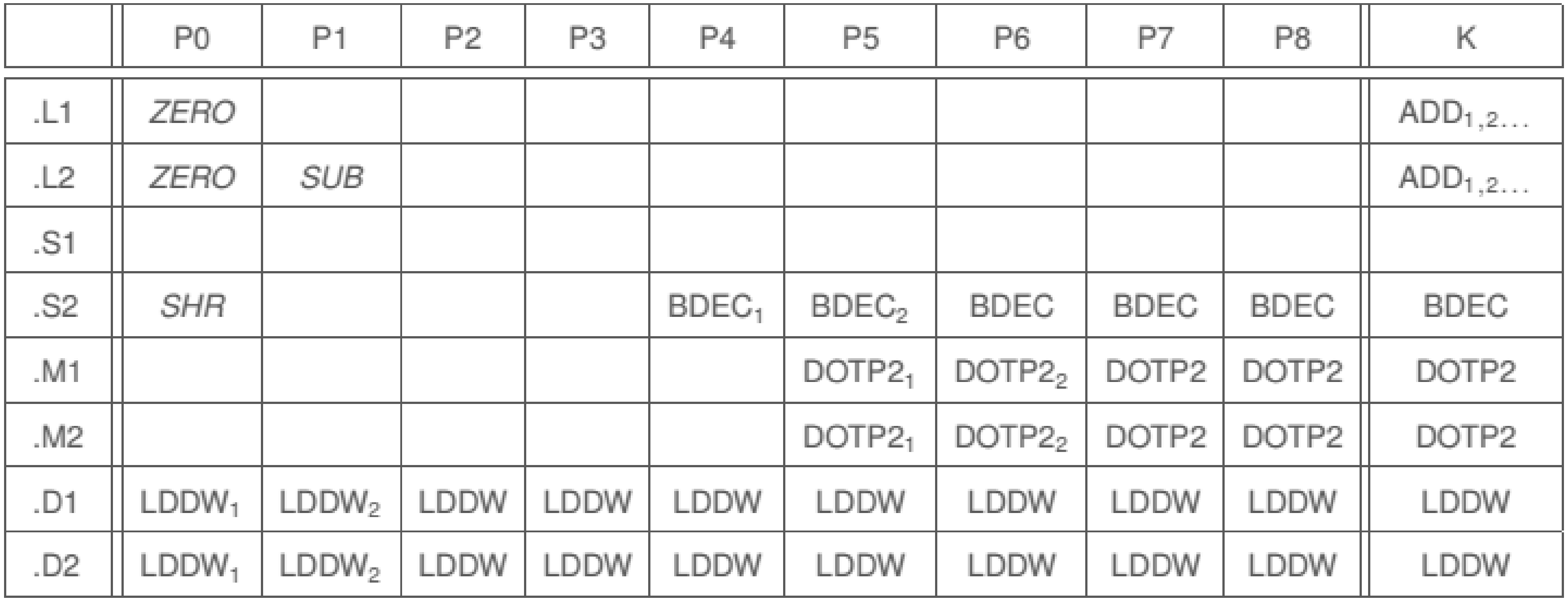
3. Functional Units & Register Allocation



4. Create Scheduling Table

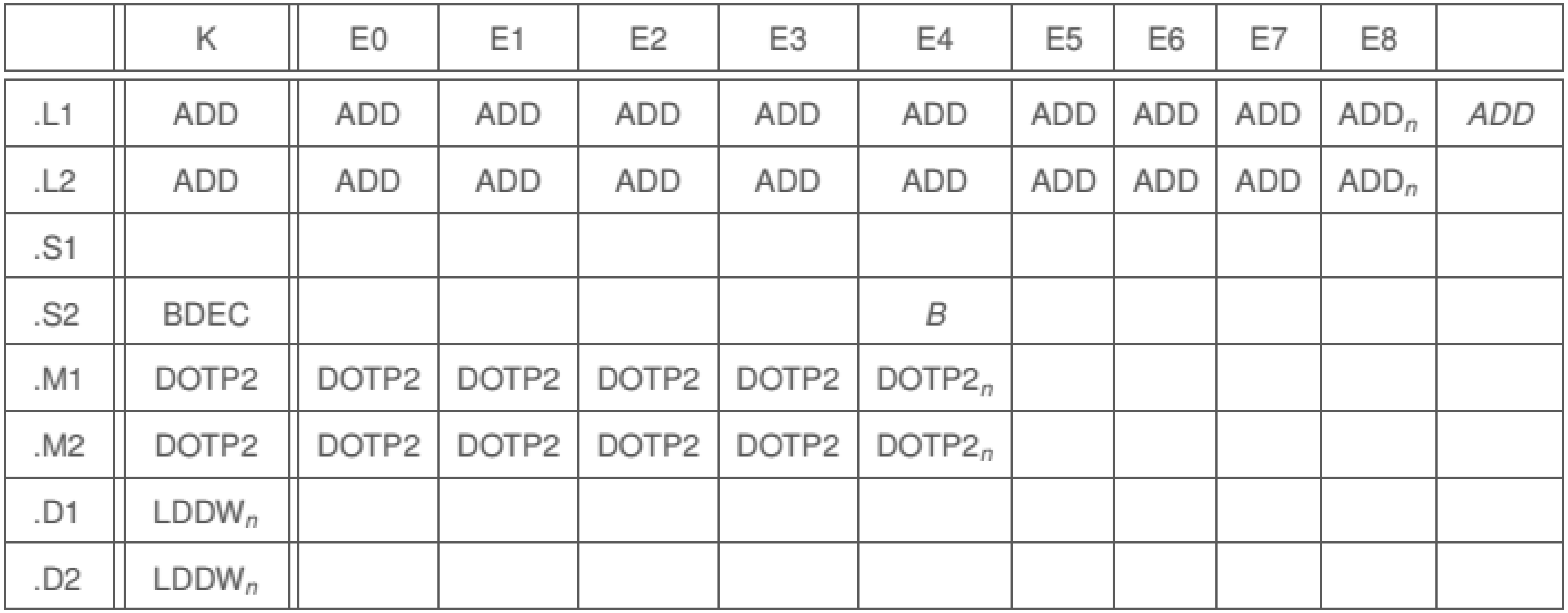
- Derived from the data flow graph
- 3 Phases
  - Prolog
  - Kernel
  - Epilog
- How long is the prolog?
  - Length of the longest path in the DFG

4. Create Scheduling Table – Prolog



	P0	P1	P2	P3	P4	P5	P6	P7	P8	K
.L1	*ZERO*									ADD$_{1,2...}$
.L2	*ZERO*	*SUB*								ADD$_{1,2...}$
.S1										
.S2	*SHR*				BDEC$_1$	BDEC$_2$	BDEC	BDEC	BDEC	BDEC
.M1						DOTP2$_1$	DOTP2$_2$	DOTP2	DOTP2	DOTP2
.M2						DOTP2$_1$	DOTP2$_2$	DOTP2	DOTP2	DOTP2
.D1	LDDW$_1$	LDDW$_2$	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW
.D2	LDDW$_1$	LDDW$_2$	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW	LDDW

4. Create Scheduling Table – Epilog



	K	E0	E1	E2	E3	E4	E5	E6	E7	E8	
.L1	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD$_n$	*ADD*
.L2	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD$_n$	
.S1											
.S2	BDEC					*B*					
.M1	DOTP2	DOTP2	DOTP2	DOTP2	DOTP2	DOTP2$_n$					
.M2	DOTP2	DOTP2	DOTP2	DOTP2	DOTP2	DOTP2$_n$					
.D1	LDDW$_n$										
.D2	LDDW$_n$										

5. Writing the Code

```
        LDDW    .D1   *A4++,A9:A8              LDDW    .D1   *A4++,A9:A8
||      LDDW    .D2   *B4++,B9:B8     ||       LDDW    .D2   *B4++,B9:B8
||      SHR     .S2   A6,2,B1         || [B1]  BDEC    .S2   loop,B1
||      ZERO    .L1   A7
||      ZERO    .L2   B7
                                               LDDW    .D1   *A4++,A9:A8
                                      ||       LDDW    .D2   *B4++,B9:B8
        LDDW    .D1   *A4++,A9:A8     ||       DOTP2   .M1X  A9,B9,A5
||      LDDW    .D2   *B4++,B9:B8     ||       DOTP2   .M2X  B8,A8,B5
||      ADD     .L2   B1,-10,B1       || [B1]  BDEC    .S2   loop,B1

                                               LDDW    .D1   *A4++,A9:A8
        LDDW    .D1   *A4++,A9:A8     ||       LDDW    .D2   *B4++,B9:B8
||      LDDW    .D2   *B4++,B9:B8     ||       DOTP2   .M1X  A9,B9,A5
                                      ||       DOTP2   .M2X  B8,A8,B5
        LDDW    .D1   *A4++,A9:A8     || [B1]  BDEC    .S2   loop,B1
||      LDDW    .D2   *B4++,B9:B8
```

5. Writing the Code: Loop Kernel

```
loop:
            LDDW        .D1     *A4++, A9:A8
||          LDDW        .D2     *B4++, B9:B8
||          DOTP2       .M1X    A9, B9, A5
||          DOTP2       .M2X    B8, A8, B5
||          ADD         .L1     A5, A7, A7
||          ADD         .L2     B5, B7, B7
||    [B1]  BDEC        .S2     loop, B1
```

Optimization Summary

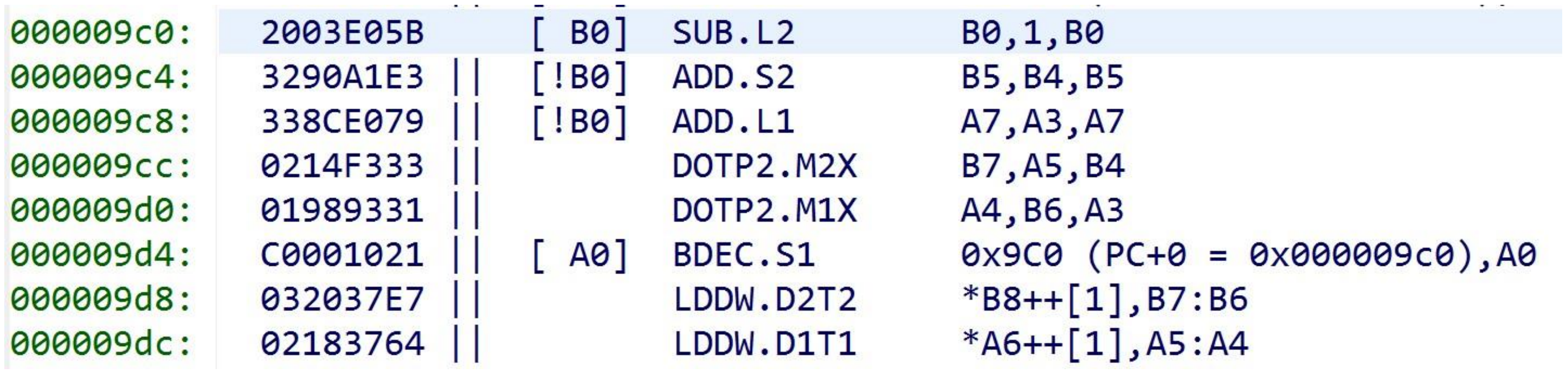
- No optimization: 20 cycles × 256 iterations
  5120 cycles
- Fill delay slots: 9 cycles × 256 iterations
  2304 cycles
- Parallel instructions: 8 cycles × 256 iterations
  2048 cycles
- Word-wide optimization: 10 cycles × 64 iterations
  640 cycles
- Software pipeline:
  10 c. prolog & kernel + 10 c. epilog & final sum
  + $(n/4 - 10)$ c. kernel = **74** cycles

Issues of Optimized Solution

- Word-wide optimization
  - Loop count has to be a *multiple of 4*

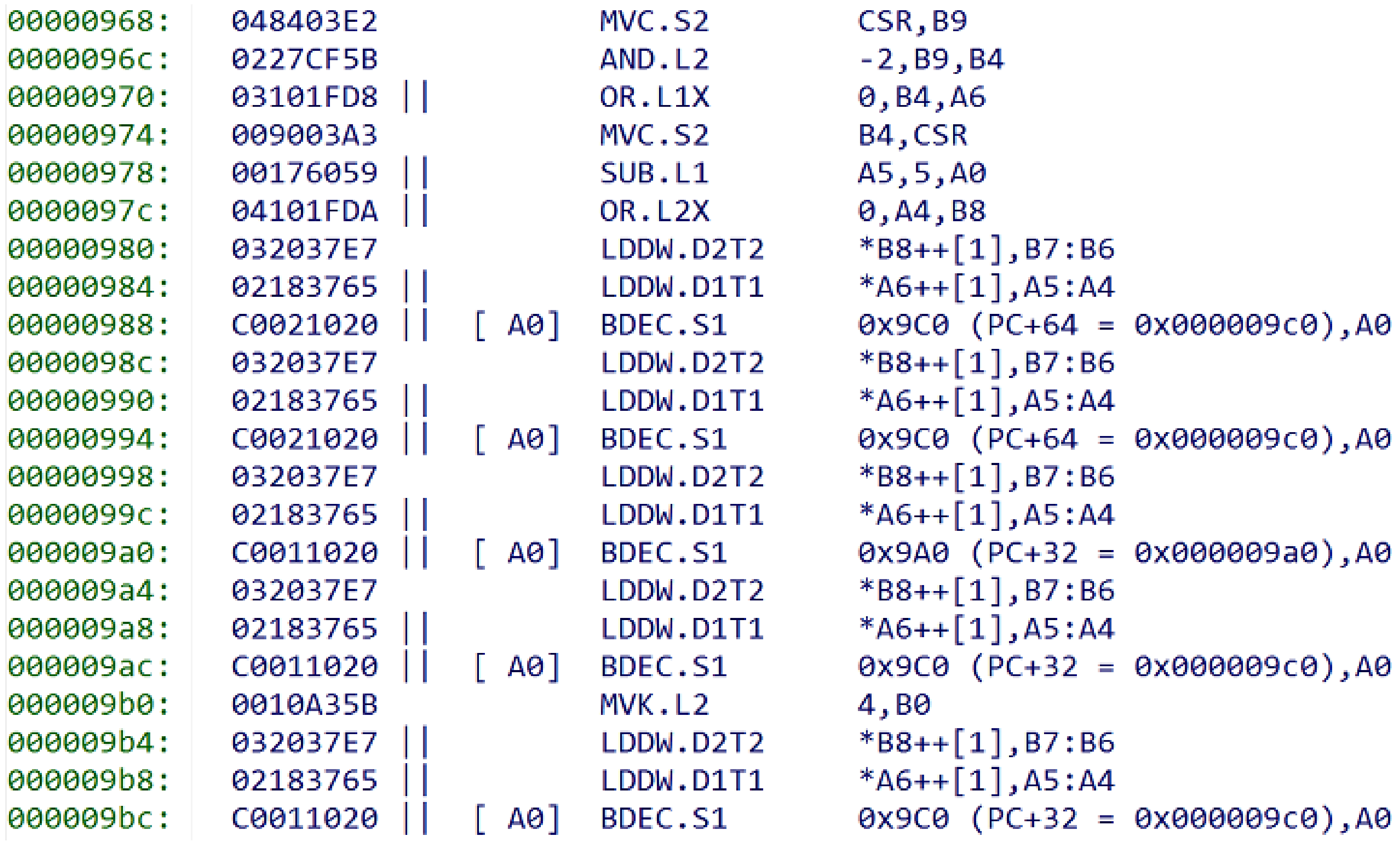
    => additional code to handle other cases
- Pipelining
  - Minimum number of iterations
  - Depends on the length of the prolog

Comparison: Optimized Linear Assembler



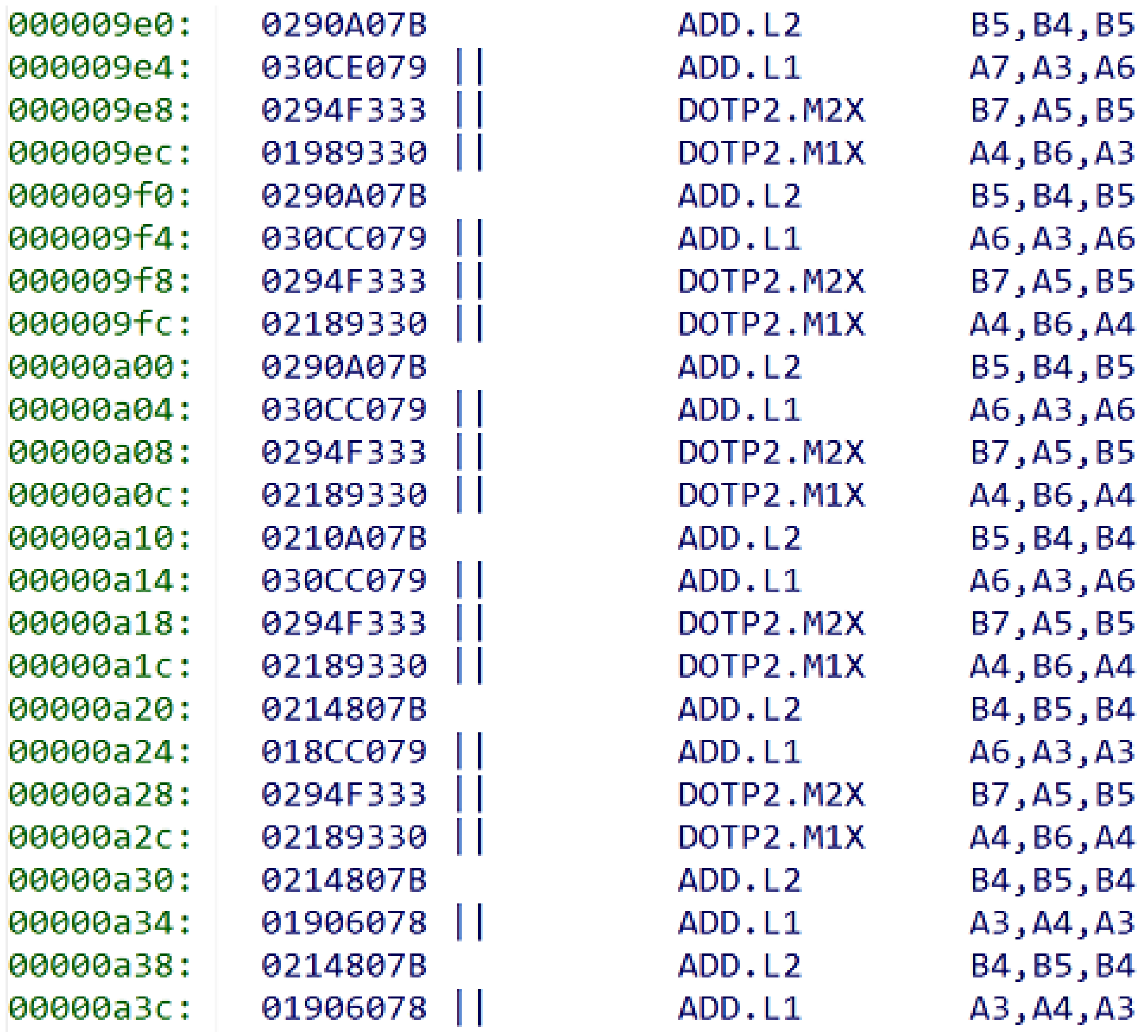
```
000009c0:    2003E05B      [ B0]  SUB.L2       B0,1,B0
000009c4:    3290A1E3 ||   [!B0]  ADD.S2       B5,B4,B5
000009c8:    338CE079 ||   [!B0]  ADD.L1       A7,A3,A7
000009cc:    0214F333 ||          DOTP2.M2X    B7,A5,B4
000009d0:    01989331 ||          DOTP2.M1X    A4,B6,A3
000009d4:    C0001021 ||   [ A0]  BDEC.S1      0x9C0 (PC+0 = 0x000009c0),A0
000009d8:    032037E7 ||          LDDW.D2T2    *B8++[1],B7:B6
000009dc:    02183764 ||          LDDW.D1T1    *A6++[1],A5:A4
```

Comparison: Optimized Linear Assembler



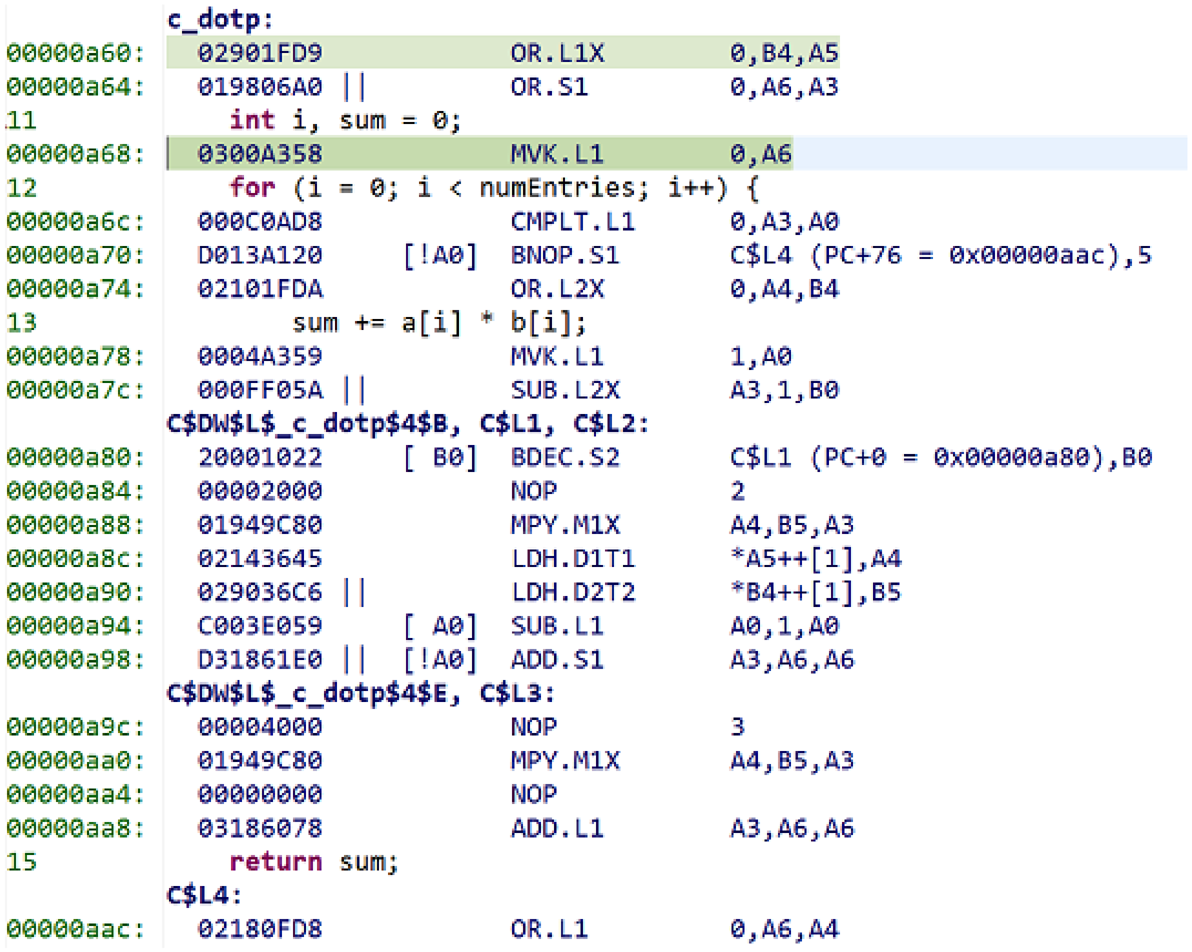
```
00000968:   048403E2                 MVC.S2        CSR,B9
0000096c:   0227CF5B                 AND.L2        -2,B9,B4
00000970:   03101FD8 ||              OR.L1X        0,B4,A6
00000974:   009003A3                 MVC.S2        B4,CSR
00000978:   00176059 ||              SUB.L1        A5,5,A0
0000097c:   04101FDA ||              OR.L2X        0,A4,B8
00000980:   032037E7                 LDDW.D2T2     *B8++[1],B7:B6
00000984:   02183765 ||              LDDW.D1T1     *A6++[1],A5:A4
00000988:   C0021020 ||     [ A0]    BDEC.S1       0x9C0 (PC+64 = 0x000009c0),A0
0000098c:   032037E7                 LDDW.D2T2     *B8++[1],B7:B6
00000990:   02183765 ||              LDDW.D1T1     *A6++[1],A5:A4
00000994:   C0021020 ||     [ A0]    BDEC.S1       0x9C0 (PC+64 = 0x000009c0),A0
00000998:   032037E7                 LDDW.D2T2     *B8++[1],B7:B6
0000099c:   02183765 ||              LDDW.D1T1     *A6++[1],A5:A4
000009a0:   C0011020 ||     [ A0]    BDEC.S1       0x9A0 (PC+32 = 0x000009a0),A0
000009a4:   032037E7                 LDDW.D2T2     *B8++[1],B7:B6
000009a8:   02183765 ||              LDDW.D1T1     *A6++[1],A5:A4
000009ac:   C0011020 ||     [ A0]    BDEC.S1       0x9C0 (PC+32 = 0x000009c0),A0
000009b0:   0010A35B                 MVK.L2        4,B0
000009b4:   032037E7 ||              LDDW.D2T2     *B8++[1],B7:B6
000009b8:   02183765 ||              LDDW.D1T1     *A6++[1],A5:A4
000009bc:   C0011020 ||     [ A0]    BDEC.S1       0x9C0 (PC+32 = 0x000009c0),A0
```

Comparison: Optimized Linear Assembler



```
000009e0:    0290A07B            ADD.L2       B5,B4,B5
000009e4:    030CE079 ||         ADD.L1       A7,A3,A6
000009e8:    0294F333 ||         DOTP2.M2X    B7,A5,B5
000009ec:    01989330 ||         DOTP2.M1X    A4,B6,A3
000009f0:    0290A07B            ADD.L2       B5,B4,B5
000009f4:    030CC079 ||         ADD.L1       A6,A3,A6
000009f8:    0294F333 ||         DOTP2.M2X    B7,A5,B5
000009fc:    02189330 ||         DOTP2.M1X    A4,B6,A4
00000a00:    0290A07B            ADD.L2       B5,B4,B5
00000a04:    030CC079 ||         ADD.L1       A6,A3,A6
00000a08:    0294F333 ||         DOTP2.M2X    B7,A5,B5
00000a0c:    02189330 ||         DOTP2.M1X    A4,B6,A4
00000a10:    0210A07B            ADD.L2       B5,B4,B4
00000a14:    030CC079 ||         ADD.L1       A6,A3,A6
00000a18:    0294F333 ||         DOTP2.M2X    B7,A5,B5
00000a1c:    02189330 ||         DOTP2.M1X    A4,B6,A4
00000a20:    0214807B            ADD.L2       B4,B5,B4
00000a24:    018CC079 ||         ADD.L1       A6,A3,A3
00000a28:    0294F333 ||         DOTP2.M2X    B7,A5,B5
00000a2c:    02189330 ||         DOTP2.M1X    A4,B6,A4
00000a30:    0214807B            ADD.L2       B4,B5,B4
00000a34:    01906078 ||         ADD.L1       A3,A4,A3
00000a38:    0214807B            ADD.L2       B4,B5,B4
00000a3c:    01906078 ||         ADD.L1       A3,A4,A3
```

Comparison: Optimized C-Function



```
            c_dotp:
00000a60:     02901FD9                OR.L1X        0,B4,A5
00000a64:     019806A0 ||             OR.S1         0,A6,A3
11              int i, sum = 0;
00000a68:     0300A358                MVK.L1        0,A6
12              for (i = 0; i < numEntries; i++) {
00000a6c:     000C0AD8                CMPLT.L1      0,A3,A0
00000a70:     D013A120       [!A0]    BNOP.S1       C$L4 (PC+76 = 0x00000aac),5
00000a74:     02101FDA                OR.L2X        0,A4,B4
13                  sum += a[i] * b[i];
00000a78:     0004A359                MVK.L1        1,A0
00000a7c:     000FF05A ||             SUB.L2X       A3,1,B0
            C$DW$L$_c_dotp$4$B, C$L1, C$L2:
00000a80:     20001022       [ B0]    BDEC.S2       C$L1 (PC+0 = 0x00000a80),B0
00000a84:     00002000                NOP           2
00000a88:     01949C80                MPY.M1X       A4,B5,A3
00000a8c:     02143645                LDH.D1T1      *A5++[1],A4
00000a90:     029036C6 ||             LDH.D2T2      *B4++[1],B5
00000a94:     C003E059       [ A0]    SUB.L1        A0,1,A0
00000a98:     D31861E0 ||    [!A0]    ADD.S1        A3,A6,A6
            C$DW$L$_c_dotp$4$E, C$L3:
00000a9c:     00004000                NOP           3
00000aa0:     01949C80                MPY.M1X       A4,B5,A3
00000aa4:     00000000                NOP
00000aa8:     03186078                ADD.L1        A3,A6,A6
15              return sum;
            C$L4:
00000aac:     02180FD8                OR.L1         0,A6,A4
```

Meeting Real-Time Requirements
Writing Interruptible Code

Interrupts

- Stops the current process so that the CPU can handle event.
  - Timer
  - Peripherals
  - . . .
- Servicing an interrupt:
  - Save current CPU state
  - Switch context
  - On return: restore CPU State

Interruptible Code

- For writing interruptible code we have to ask ourselves:
  If we stop the code and restart it again at any time:
  do we produce the same results?
- There are several things to consider:
  - Registers with multiple-assignment
  - Pending branches, tight loops
  - Compiler-generated code

Single-Assignment vs. Multiple-Assignment

```
cycle
   1    SUB  .S1 A4,A5,A1   ;writes to A1 in single cycle
   2    LDW  .D1 *A0,A1     ;writes to A1 after 4 delay sl.
   3    NOP
   4    ADD  .L1 A1,A2,A3   ;uses old A1 (result of SUB)
   5-6  NOP  2
   7    MPY  .M1 A1,A4,A5   ;uses new A1 (result of LDW)
```

- Multiple assignment: register has been assigned with more than one value
  - Pending assignments: in-flight
- Multiple assignment must/should not be interrupted

=> Register renaming

Branches & Interruptible Loops

- Delay slots of all branch operations are protected from interrupts
- Interrupts remain pending
- Loops smaller than 6 cycles are uninterruptible!
- Solutions
  - Make the loop more than 6 cycles (slow down the loop)
  - Unroll the loop until an iteration is more than 6 cycles
  - Nested loops: fast non-interruptible inner loop & slow outer loop

Interruptibility Options in Code Generation

- Compiler Option -mi
- `#pragma FUNC_INTERRUPT_THRESHOLD(func, value)`
- 3 Levels of control
  - `-mi / value = uint_max:`
    Specified code is guaranteed to not be interrupted
  - `-mi 1 / value = 1:`
    Specified code interruptible at all times
  - `-mi` *`threshold`* `/` *`value`* `=` *`threshold`*`:`
    Specified code interruptible within threshold cycles

Interruptible Code – Examples

- Compiler option: `-mi 100 -mi 1`

```
int dot_prod(short *a, short *b, int n) {
   int i, sum = 0;
   #pragma MUST_ITERATE (20);
   for (i = 0; i < n; i++)
      sum += a[i] * b[i];
   return sum;
}
```

Interruptible Code – Examples

- Compiler option: `-mi 100 -mi 1`

```
int dot_prod(short *a, short *b, int n) {
   int i, sum = 0;
   #pragma MUST_ITERATE (20, 50);
   for (i = 0; i < n; i++)
      sum += a[i] * b[i];
   return sum;
}
```

Interruptible Code – Examples

- Compiler option: `-mi 100 -mi 1`

```
int dot_prod(short *a, short *b, int n) {
   int i, sum = 0;
   #pragma MUST_ITERATE (20, 50, 2);
   for (i = 0; i < n; i++)
      sum += a[i] * b[i];
   return sum;
}
```

Interruptible Code – Examples

- Compiler option: `-mi 100 -mi 1`

```
int dot_prod(short *a, short *b, int n) {
   int i, sum = 0;
   #pragma MUST_ITERATE (16, 48, 4);
   for (i = 0; i < n; i++)
      sum += a[i] * b[i];
   return sum;
}
```

Things we have not considered

- Memory stalls
    - Memory bank conflicts
    - External memory access
    - Cache miss

Conclusion

Additional References

Additional References

- Johannes Fürtler, Konrad J. Mayer, Werner Krattenthaler, Ivan Bajla: “SPOT – Development tool for software pipeline optimization for VLIW-DSPs used in real-time image processing”
- “TMS320C6000 Programmer’s guide” (SPRU198k)
- “TMS320C6000 Optimizing Compiler” (SPRU187V)
- “C6000 Optimization Workshop Teaching Material”