Chapter 5

C6678's Assembly

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

2021-2022

C6678 INSTRUCTION SET ARCHITECTURE

ENSI CAEN

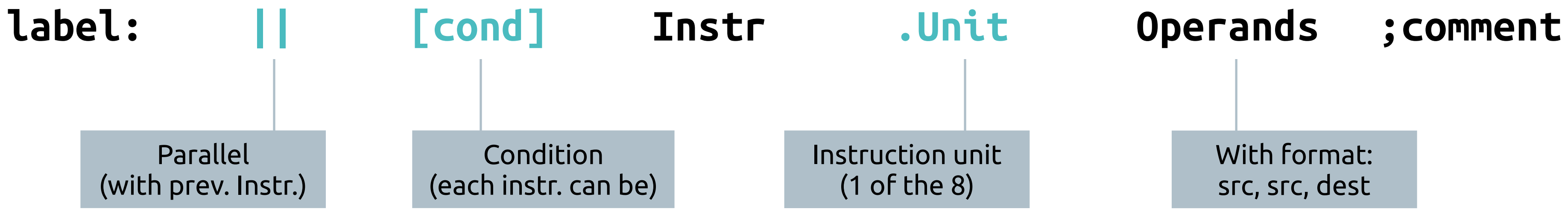
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

C6678 INSTRUCTION SET ARCHITECTURE

Instruction fields

Let’s see the different fields of an instruction line in assembly language for the Texas Instruments C6600 architectures.

Note that some fields are specific to VLIW architectures.

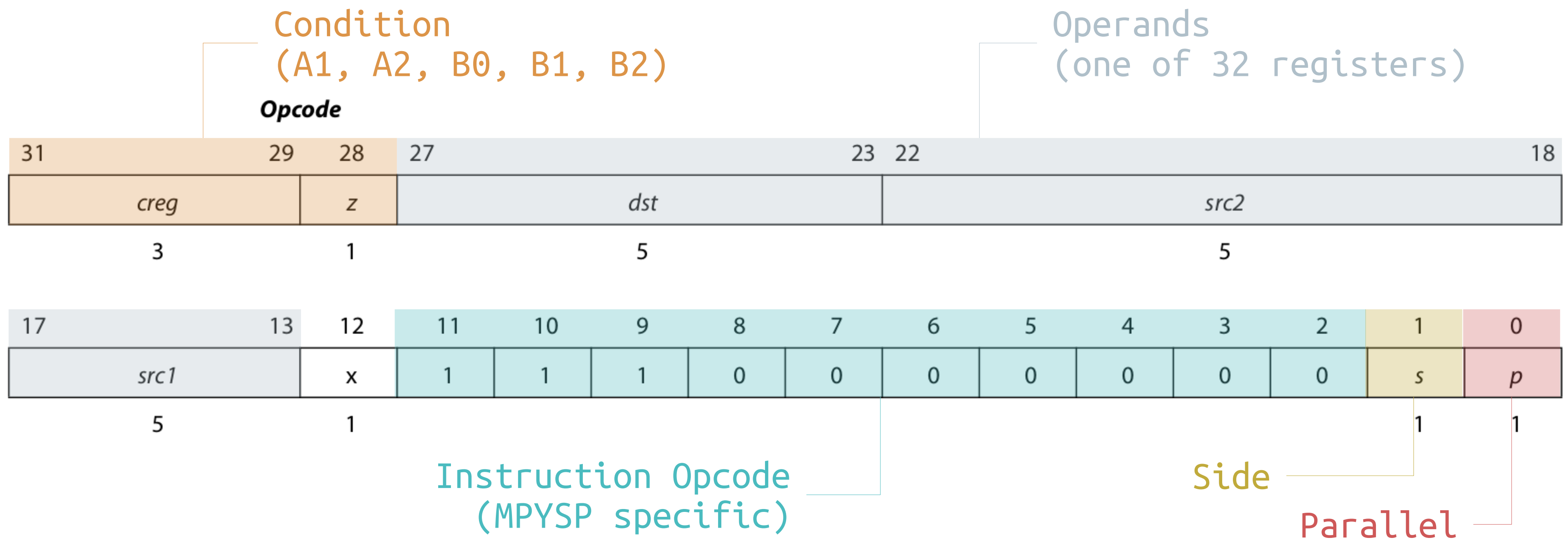


label:
||
[cond]
Instr
.Unit
Operands
;comment
Parallel (with prev. Instr.)
Condition (each instr. can be)
Instruction unit (1 of the 8)
With format: src, src, dest

Instruction fields

Remember that all fields of an instruction in assembly language correspond to a field in the binary code of the instruction (except for the label and the comment).

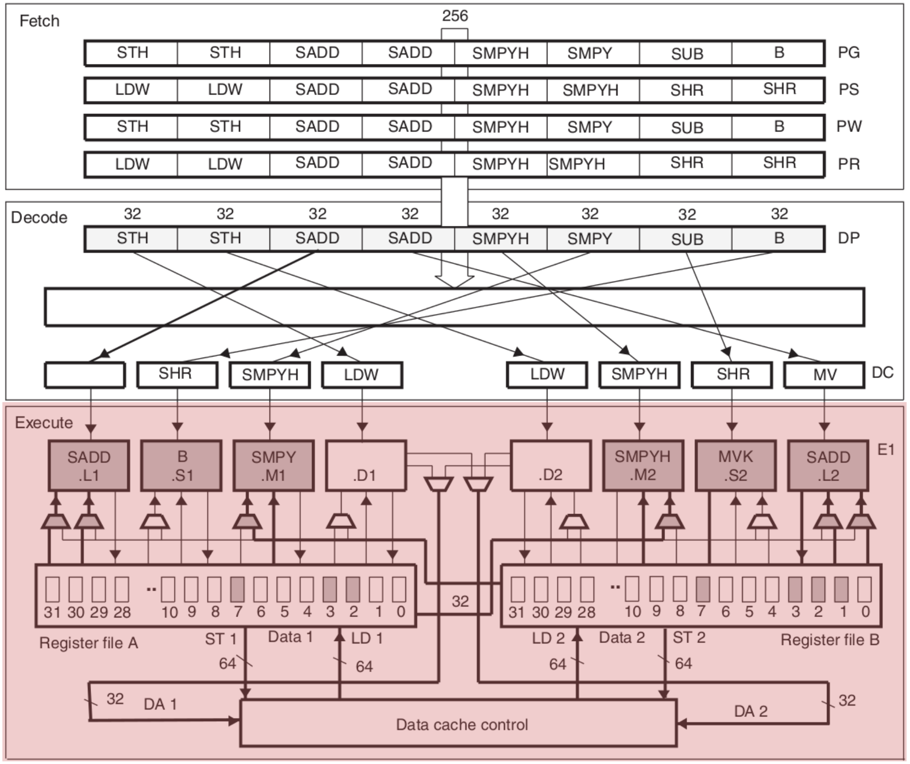
See for instance the MPYSP instruction.



C6678 INSTRUCTION SET ARCHITECTURE

Execution units

In order to ease the understanding of the C6600 Instruction Set Architecture, we'll look at the effects of the assembly instructions onto the execution stage.



Execution units

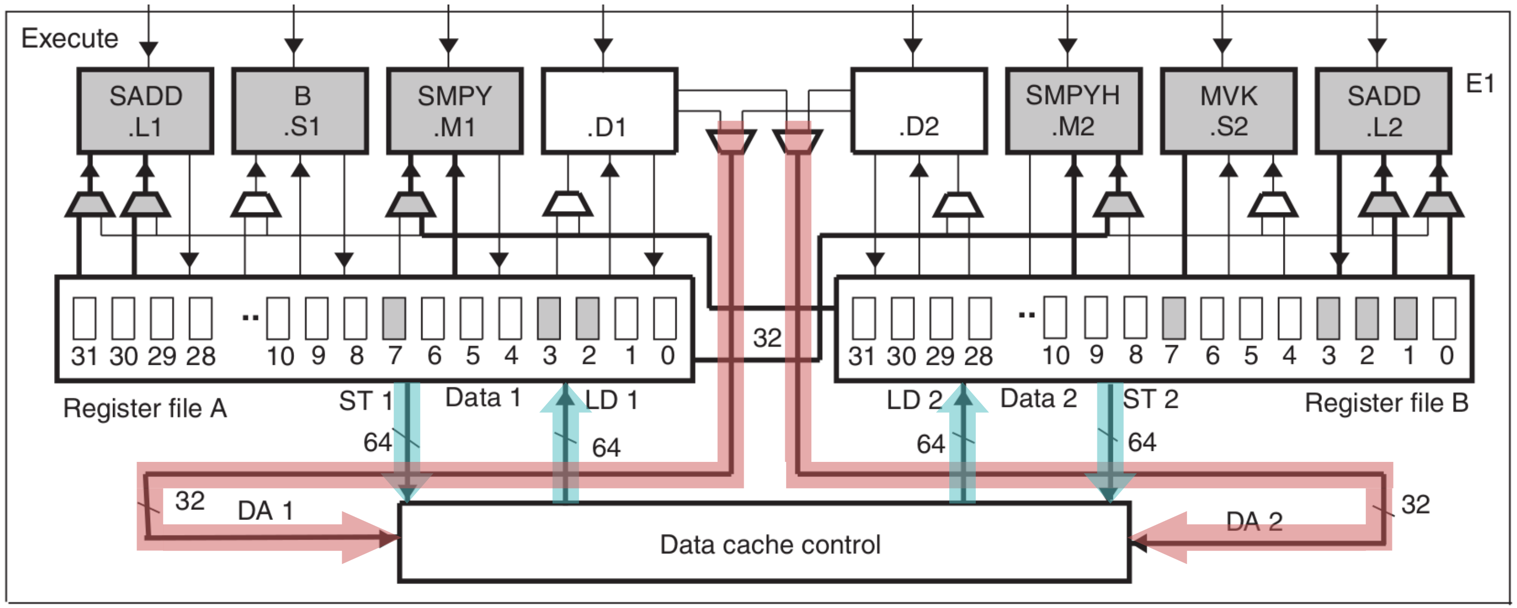
The **C6000 CPU has a Load-Store architecture**.

This means that some execution units (.D1 and .D2) are dedicated to memory access, and both of them have a direct access to the L1 cache memory (64-bit bus).

The other execution units are used for control and processing.

Data bus (64-bit)
L1 cache ←→ Reg file A/B

Memory address bus (32-bit)
.D1/.D2 → L1 cache



Addressing modes

Only **3 addressing modes** are supported by the C6000 ISA.

Remind that addressing modes correspond to data manipulation strategies , as used by the instructions.

Being a calculation processor, the C6000 CPU heavily uses register addressing mode.

- **Register addressing**
  - 324 instructions (full ISA)
- **Indirect addressing**
  - 18 instructions (load/store instructions)
- **Immediate addressing**

DISCRETE CONVOLUTION ALGORITHM IN CANONICAL C6678 ASSEMBLY

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Presentation

For the remaining part of this lecture, we'll translate the C algorithm into a C6678 assembly language program.

The canonical C version of the program is on the next slide.

For educational purpose, we will ignore the delay slots that are associated to instructions execution time.

The absence of the NOP instructions will facilitate the understanding of the canonical assembly program.

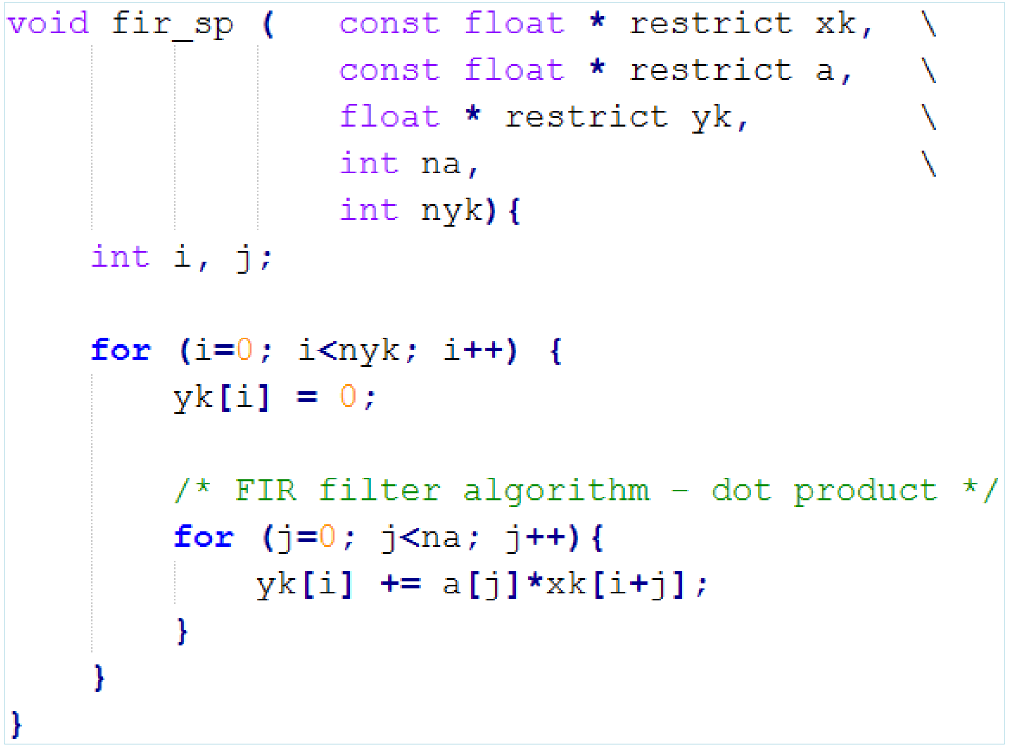
Do not forget to add the delay slots when programming the target DSP!

DISCRETE CONVOLUTION ALGORITHM IN CANONICAL C6678 ASSEMBLY LANGUAGE

Canonical C implementation

This is the algorithm that will be studied during the lab sessions.

It’s a canonical C implementation, using IEEE-754 single-precision floats.

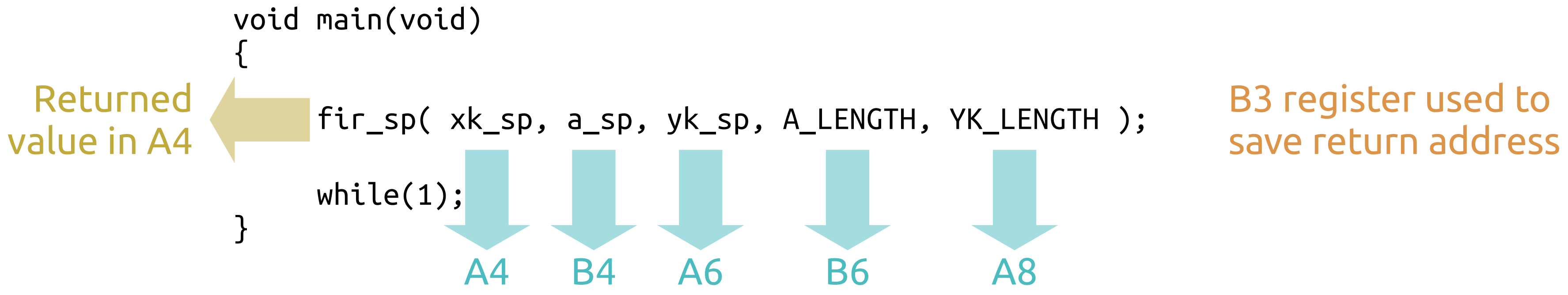


```c
void fir_sp (   const float * restrict xk,   \
                const float * restrict a,    \
                float * restrict yk,         \
                int na,                      \
                int nyk){
    int i, j;

    for (i=0; i<nyk; i++) {
        yk[i] = 0;

        /* FIR filter algorithm - dot product */
        for (j=0; j<na; j++){
            yk[i] += a[j]*xk[i+j];
        }
    }
}
```

Registers use

Registers used for passing parameters through function call:

See "TMS320C6000 Optimizing Compiler V7.6" User's guide, Chapter "7.3 Register conventions"



```
void main(void)
{
    fir_sp( xk_sp, a_sp, yk_sp, A_LENGTH, YK_LENGTH );

    while(1);
}
```

We decide to use those registers:

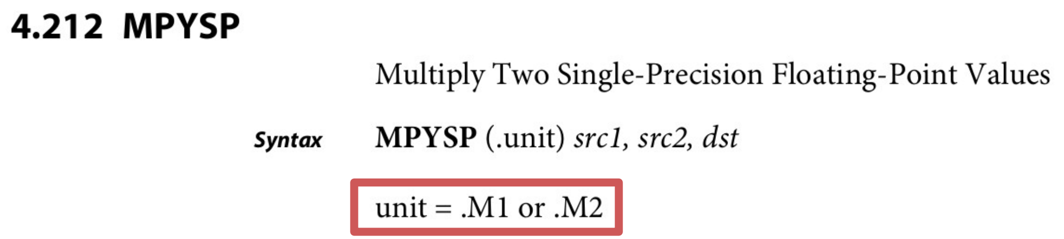
- i = B0
- j = A1
- yk (temp) = A5
- xk_sp = A19
- a_sp = B19
- xk = A9
- a = B9

Multiplication-Accumulation

The easiest way to translate the C into assembly language is to start from the main operation, inside the inner loop.

Like many other digital signal processing algorithms, the discrete convolution uses **MAC** (Multiply-Accumulate) or **SOP** (Sum Of Products) instructions.

First let's look at the MPYSP instruction.



4.212 MPYSP

Multiply Two Single-Precision Floating-Point Values

Syntax **MPYSP** (.unit) *src1, src2, dst*

unit = .M1 or .M2

```
fir_sp_asm:









          MPYSP  .M1       A9,   B9, A17
```

DISCRETE CONVOLUTION ALGORITHM IN CANONICAL C6678 ASSEMBLY LANGUAGE

Multiplication-Accumulation

The current example shows a cross path (i.e. the two sources are not from the same side).

This brings some limitations:

- The destination register and the execution unit must be on the same side
- Only one source register can be on the other side
- Add the 'x' suffix when specifying the execution unit

```
fir_sp_asm:




        MPYSP .M1x     A9,  B9, A17
```

Multiplication-Accumulation

Now we can use an addition instruction.

4.14 ADDSP

Add Two Single-Precision Floating-Point Values

Syntax **ADDSP** (.unit) *src1, src2, dst*

unit = .L1, .L2, .S1, .S2

```
fir_sp_asm:

        MPYSP  .M1x      A9,  B9, A17
        ADDSP  .L1       A17, A5, A5
```

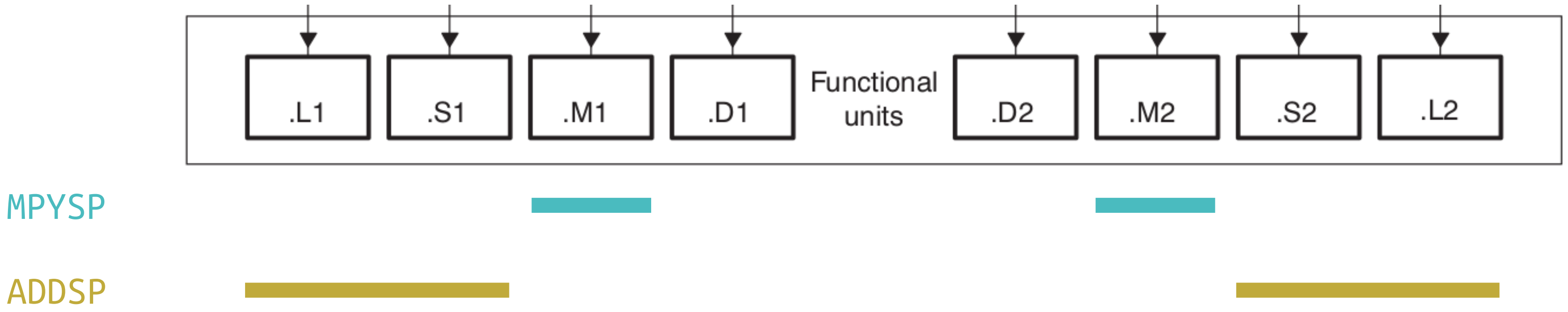
Well done!

You just implemented

a MAC operation!

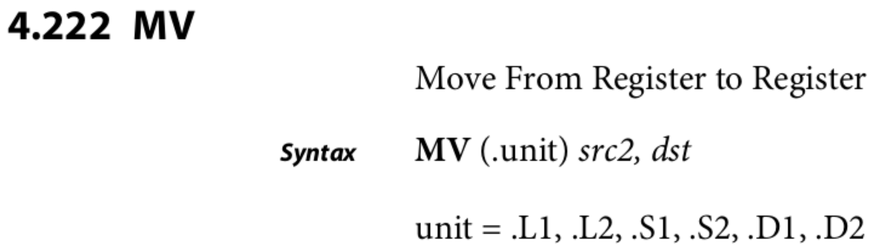
Multiplication-Accumulation

Use of execution units:



Data management

Move data from a CPU register to another one.



4.222 MV

Move From Register to Register

Syntax **MV** (.unit) *src2*, *dst*

unit = .L1, .L2, .S1, .S2, .D1, .D2

```
fir_sp_asm:
        MV     .L1        A8, B0




        MPYSP  .M1x       A9,  B9, A17
        ADDSP  .L1        A17, A5, A5
```

Data management

Before performing the MAC, we must load the cells values (stored in the L1 cache memory) into CPU registers.

We must use one of the LDx (load) instructions:

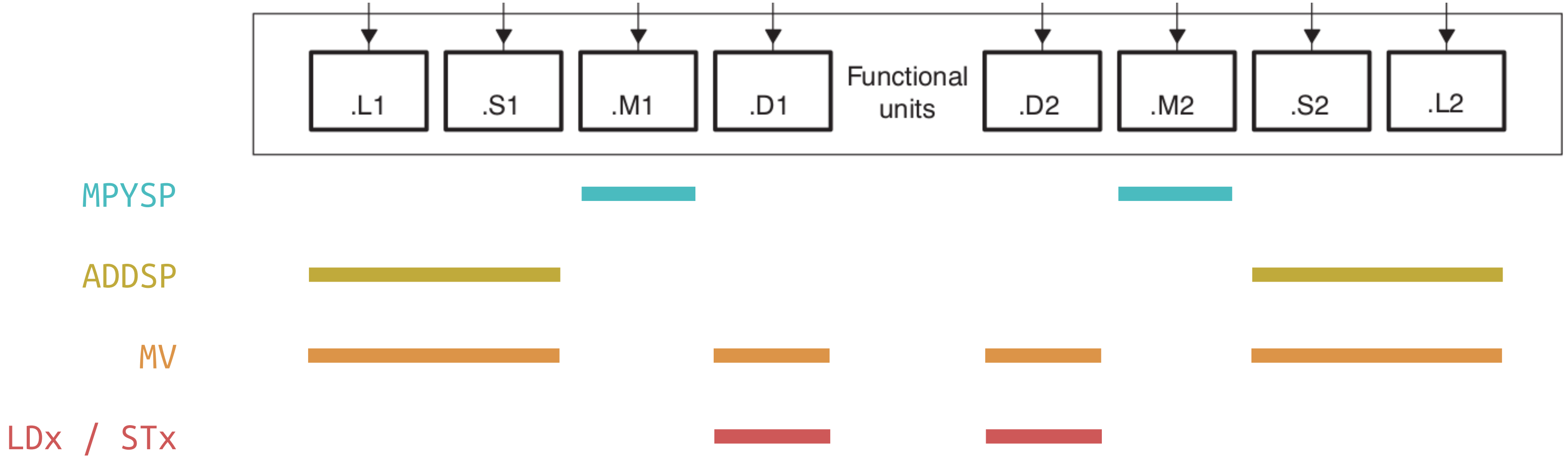
- `LDB,  B  = Byte        = 1 byte  = char`
- `LDH,  H  = Half-word   = 2 bytes = short int`
- `LDW,  W  = Word        = 4 bytes = int, float`
- `LDDW, DW = Double-Word = 8 bytes = long, double`

The .D1 and .D2 executions units are dedicated to ST and LD instructions only.

```
fir_sp_asm:
          MV     .L1        A8, B0






          LDW    .D1        *A19,  A9
          LDW    .D2        *B19,  B9
          MPYSP  .M1x       A9,  B9, A17
          ADDSP  .L1        A17, A5, A5
```

Data management

Use of execution units:



Data management

Note that a pointer-like style is used.

In this example case, A19 and B19 registers contain each an address. The '*' character before the register name indicates the use of **indirect addressing mode**.

This is equivalent to the use of pointers in C.

```
fir_sp_asm:
         MV    .L1        A8, B0









         LDW    .D1       *A19,  A9
         LDW    .D2       *B19,  B9
         MPYSP  .M1x      A9,  B9, A17
         ADDSP  .L1       A17, A5, A5
```

DISCRETE CONVOLUTION ALGORITHM IN CANONICAL C6678 ASSEMBLY LANGUAGE

Data management

Similarly to the pointers in C language, registers used in indirect addressing mode support **pre- and post-incrementations**.

Also, load and store operations can **be indexed with the [] notation**, like arrays in C.

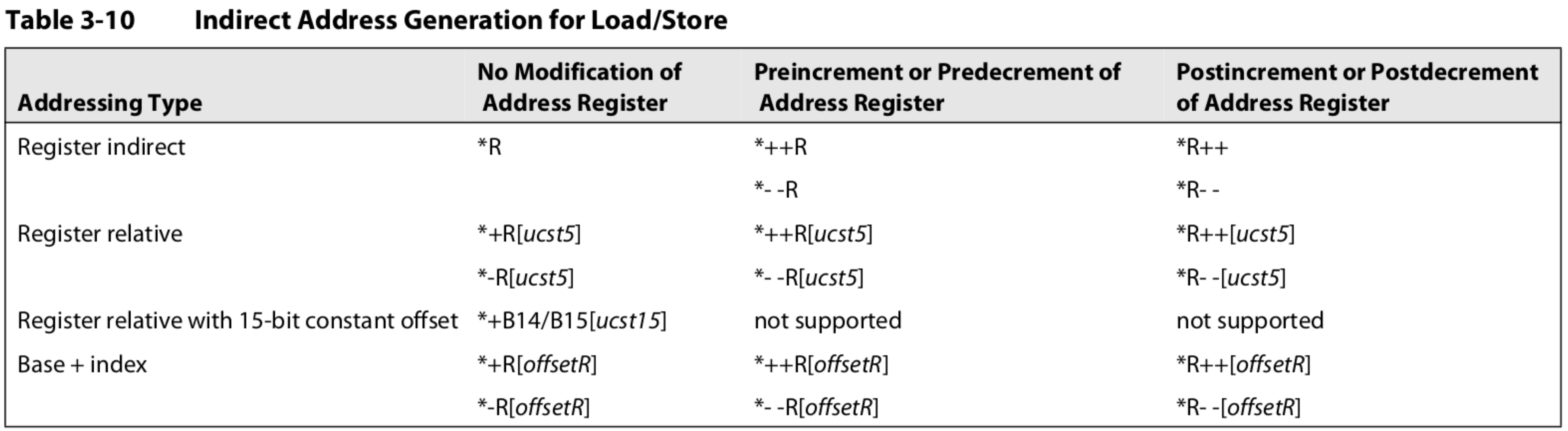


Table 3-10 Indirect Address Generation for Load/Store

Addressing Type	No Modification of Address Register	Preincrement or Predecrement of Address Register	Postincrement or Postdecrement of Address Register
Register indirect	*R	*++R	*R++
		*- -R	*R- -
Register relative	*+R[*ucst5*]	*++R[*ucst5*]	*R++[*ucst5*]
	*-R[*ucst5*]	*- -R[*ucst5*]	*R- -[*ucst5*]
Register relative with 15-bit constant offset	*+B14/B15[*ucst15*]	not supported	not supported
Base + index	*+R[*offsetR*]	*++R[*offsetR*]	*R++[*offsetR*]
	*-R[*offsetR*]	*- -R[*offsetR*]	*R- -[*offsetR*]

Data management

To summarize:

The A19 and B19 registers contain the address of the current cell of a[] and xk[] arrays.

The two LDW instructions load 4 bytes from the L1 cache memory to the CPU registers A9 & B9.

The address value contained in A19 and B19 registers are incremented afterward, making these registers pointing to the next cell.

```
fir_sp_asm:
         MV    .L1       A8, B0









         LDW    .D1      *A19++,  A9
         LDW    .D2      *B19++,  B9
         MPYSP .M1x      A9,  B9, A17
         ADDSP .L1       A17, A5, A5
```

DISCRETE CONVOLUTION ALGORITHM IN CANONICAL C6678 ASSEMBLY LANGUAGE

Control and branch

The C6000 family historically supported only one branch instruction: the **B (branch) instruction**.

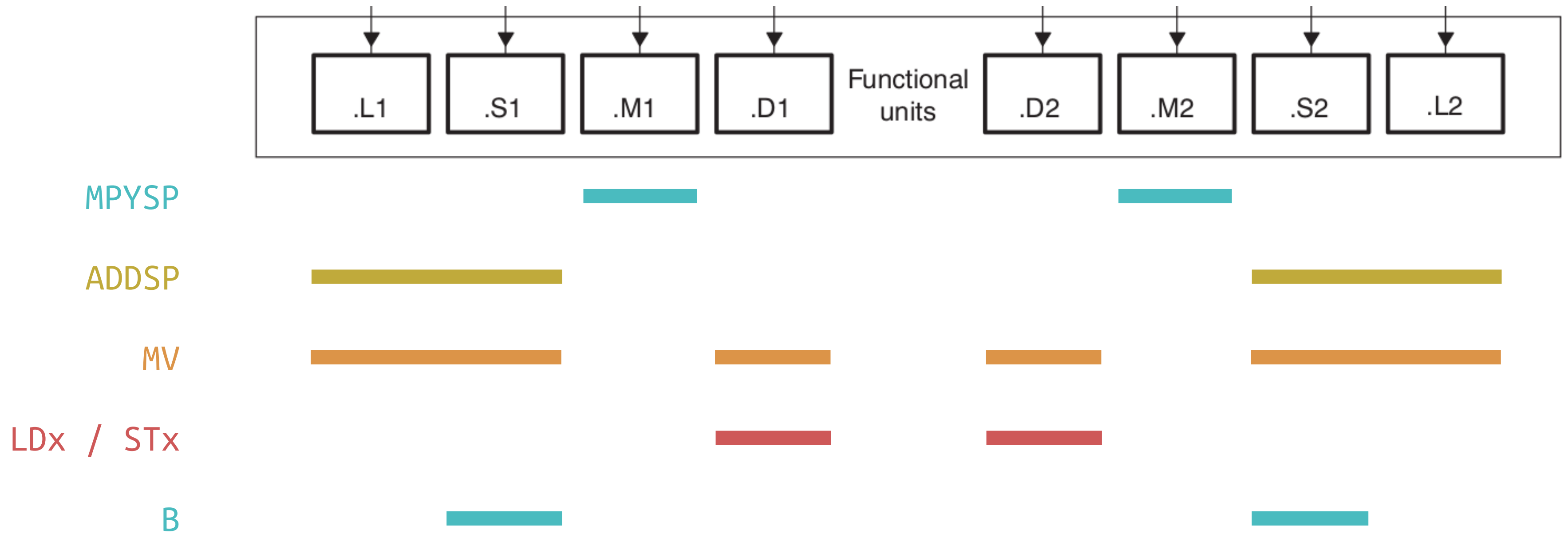
It allows to perform function calls as well as all known control structures (if, for, while, …).

The B instruction uses .S1 and .S2 execution units.

```
fir_sp_asm:
         MV    .L1       A8, B0




fir_sp_asm_l2:
         LDW    .D1      *A19++,  A9
         LDW    .D2      *B19++,  B9
         MPYSP  .M1x     A9,  B9, A17
         ADDSP  .L1      A17, A5, A5

         B      .S1      fir_sp_asm_l2
```

Control and branch

Use of execution units:



DISCRETE CONVOLUTION ALGORITHM IN CANONICAL C6678 ASSEMBLY LANGUAGE

Control and branch

A condition can be added to the execution of an instruction.

Five registers (A1, A2, B0, B1, B2) can be used as condition values.

Syntax:

- `[R]` = instruction executed if R ≠ 0
- `[!R]` = instruction executed if R = 0

All instructions can be executed conditionally.

```
fir_sp_asm:
         MV    .L1      A8, B0









fir_sp_asm_l2:
         LDW    .D1     *A19++,  A9
         LDW    .D2     *B19++,  B9
         MPYSP  .M1x    A9,  B9, A17
         ADDSP  .L1     A17, A5, A5

     [A1] B     .S1     fir_sp_asm_l2
```

Control and branch

Implement the internal loop's counter.

```
fir_sp_asm:
         MV    .L1      A8, B0



         MV     .L1     B6, A1




fir_sp_asm_l2:
         LDW    .D1     *A19++,  A9
         LDW    .D2     *B19++,  B9
         MPYSP  .M1x    A9,  B9, A17
         ADDSP  .L1     A17, A5, A5
    [A1] SUB    .L1     A1,  1,  A1
    [A1] B      .S1     fir_sp_asm_l2
```

DISCRETE CONVOLUTION ALGORITHM IN CANONICAL C6678 ASSEMBLY LANGUAGE

Control and branch

Implement the external loop.

```
fir_sp_asm:
         MV    .L1       A8, B0

fir_sp_asm_l1:

         MV     .L1      B6, A1



fir_sp_asm_l2:
         LDW    .D1      *A19++,  A9
         LDW    .D2      *B19++,  B9
         MPYSP  .M1x     A9,  B9, A17
         ADDSP  .L1      A17, A5, A5
     [A1] SUB    .L1      A1,  1,  A1
     [A1] B      .S1      fir_sp_asm_l2



     [B0] SUB    .L2      B0, 1, B0
     [B0] B      .S1      fir_sp_asm_l1
```

Control and branch

The **return address** of a function is always given by the calling function through the **B3 register**.

```
fir_sp_asm:
         MV    .L1       A8, B0

fir_sp_asm_l1:

         MV     .L1      B6, A1




fir_sp_asm_l2:
         LDW    .D1      *A19++,  A9
         LDW    .D2      *B19++,  B9
         MPYSP  .M1x     A9,  B9, A17
         ADDSP  .L1      A17, A5, A5
    [A1] SUB    .L1      A1,  1,  A1
    [A1] B      .S1      fir_sp_asm_l2



    [B0] SUB    .L2      B0, 1, B0
    [B0] B      .S1      fir_sp_asm_l1

         B               B3
```

DISCRETE CONVOLUTION ALGORITHM IN CANONICAL C6678 ASSEMBLY LANGUAGE

Final version

Final version

Without the required delay slots!

```c
void fir_sp (   const float * restrict xk,  \
                const float * restrict a,   \
                float * restrict yk,        \
                int na,                     \
                int nyk){
    int i, j;

    for (i=0; i<nyk; i++) {
        yk[i] = 0;

        /* FIR filter algorithm - dot product */
        for (j=0; j<na; j++){
            yk[i] += a[j]*xk[i+j];
        }
    }
}
```

```
fir_sp_asm:
          MV    .L1       A8, B0

fir_sp_asm_l1:
          ZERO   .L1       A5
          MV     .L1       B6, A1
          MV     .L1       A4, A19
          MV     .L1       B4, B19

fir_sp_asm_l2:
          LDW    .D1       *A19++,  A9
          LDW    .D2       *B19++,  B9
          MPYSP  .M1x      A9,  B9, A17
          ADDSP  .L1       A17, A5, A5
     [A1] SUB    .L1       A1,  1,  A1
     [A1] B      .S1       fir_sp_asm_l2

          STW    .D1       A5,   *A6++
          ADD    .L1       A4, 4, A4
     [B0] SUB    .L2       B0, 1, B0
     [B0] B      .S1       fir_sp_asm_l1

          B                B3
```