ASIP Designer

Tmicro Core

Processor Manual

Version M-2017.03

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at :
http://www.synopsys.com/Company/Pages/Trademarks.aspx.
All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
700 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Changes

Version	Date	Change
11R1.1	Apr 2011	Renamed BASE into TMICRO.
12R1.1	Feb 2012	Initialization of `SR` (section 4.2) .
12R1.2	May 2012	New divide step instruction (section 7.5) .
J-2014.09	Oct 2014	First release under Synopsys.
J-2015.03	Mar 2015	No change.
L-2016.08	Aug 2016	New implementation of `macl` instruction.
M-2017.03	Feb 2017	Update of control flow instructions.

Abstract

This document describes the TMICRO micro controller. The TMICRO core is intended to be used as a starting point for the development of application specific instruction set processors. It contains a collection of general purpose instructions such as 16 bit integer arithmetic, bitwise logical and compare instructions; load and store instructions, and various control instructions. In addition to these fixed features, a number of optional features such as shift instructions, multiplication instructions, zero overhead loops, interrupts and on chip debugging can be enabled. Next to these pre-defined instructions, the user can add application specific instructions to the TMICRO core.

Contents

1

Introduction

This document describes the TMICRO micro controller. The TMICRO core is an example processor intended to illustrate various concepts related to processor modelling, as well as tool capabilities such as on chip debugging and processor verification. TMICRO can also be used as a starting point for the development of application specific instruction set processors. TMICRO contains a collection of architecture features that are common in 16 bit micro controllers, such as:

- 16 bit integer arithmetic, bitwise logical and compare instructions. These instructions are executed on a 16 bit ALU and operate on an 8 field register file.
- Integer multiplications with 16 bit operands and 32 bit results.
- 16 bit shift instructions.
- A 16 bit division step instruction.
- Load and store instructions from and to a 16 bit data memory with an address space of 64k words, using indirect addressing. Address computations are executed on a 16 bit AGU.
- Various control instructions such a jumps and subroutine call and return.
- Zero overhead loops.
- Support for interrupts.
- Support for on chip debugging.

In addition to these predefined instructions, the user can add application specific instructions to the TMICRO core.

2

Overview of the processor model files

The TMICRO processor model is located in the `tmicro/lib` directory. It consist of the following files:

Processor model

`tmicro.prx`	The processor project file, contains settings settings that are common to all projects.
`tmicro.h`	The primitive processor header file, contains declarations for the primitive data types and primitive functions of the TMICRO core.
`tmicro.p`	The primitives definition file, contains the behaviour models for the primitive functions.
`tmicro.n`	The top level nML file, contains declarations for the storage elements of the processor, as well as the the top level instruction rules.
`alu.n`	Contains the nML model of the ALU instructions, the shift instructions, the multiplication instructions, and the division step instruction.
`load_store.n`	Contains the nML model of the load and store instructions.
`move.n`	Contains the nML model of the register move instructions.
`control.n`	Contains the nML model of the control instructions.
`ocd_if.n`	Contains the nML model for the On Chip Debugging interface.
`hazard.n`	Contains the hazard rules.
`tmicro_pcu.p`	Contains the behavioural model of the program control unit.
`tmicro_iapcu.h`	Contains the instruction accurate C model for the program control unit.

Compiler model

`tmicro_chess.h`	The top level compiler processor header file, specifies the mapping of C built-in types and operators to the primitive processor.
`tmicro_int.h`	Contains the application layer that maps the C built in types `int` and `unsigned` and associated operations on the TMICRO primitives.

`tmicro_long.h`	Contains the application layer that maps the C built in types `long` and `unsigned long` and associated operations on the TMICRO primitives.
`tmicro_bitfield.h`	Contains definitions for bit-field manipulation functions.
`tmicro_rewrite.h`	Contains Chess rewrite rules.
`tmicro_interrupt.h`	Contains functions for controller the interrupt behaviour of the processor.
`tmicro_div.c`	Contains the C functions that implement the iterative division algorithm for `int` and `unsigned`.
`tmicro_long_div.c`	Contains the C functions that implement the iterative division algorithm for `long` and `unsigned long`.
`tmicro_init.s`	Contains initialization code, which is executed before the `main()` function is entered.
`libtmicro.prx`	Project file,used to build the `libtmicro.a` archive, which contains the code of the above mentioned C and assembly files.
`tmicro_native.h`	Header file for native comipation.

Miscellaneous files

`tmicro.r`	Relocator definition file.
`tmicro.bcf`	Default linker configuration file.

3

Data types

Primitive data types [5] are types that can be stored in one or more storage elements (registers or memories) of the processor. They are declared in the primitive processor header file, and are given some properties, such as a width and sign information. The TMICRO core supports the following data types. [See file `tmicro/lib/tmicro.h`]

3.1 The `word` type

The `word` type is declared as a 16 bit signed type.

```
class word property( 16 bit signed); // main data type
```

It is used as the data type of most registers and memories in the nML model.

3.2 The `addr` type

The `addr` type is a 16 bit unsigned type. It is used as the type by which memories are addressed in the nML model.

3.3 The `nint9` type

The `nint9` type is an 9 bit type with a negative range $[-256 \ldots -1]$. It is used to specify negative offsets to the stack pointer register. The `nint9` type is modelled as follows.

```
class nint9 property( 9 bit signed min=-256 max=-1);
```

3.4 The `sbyte` type

The `sbyte` type is an 8 bit signed type. It is used for the relative offset of short jump instructions, and as the type of short immediate values.

3.5 The `ubyte` type

The `ubyte` type is an 8 bit unsigned type. It is used to represent the interrupt registers (see [6]).

3.6 The `iword` type

The `iword` type models the instruction word. It is a 16 bit unsigned type.

3.7 Other types

The small unsigned types `uint1`, `uint2`, `uint3` and `uint4` are mainly used to address register files.

4

Memories, Registers and Functional Units

This chapter presents an overview of the hardware units of the TMICRO processor. Storage elements such as memories and registers are declared in the top level nML file `tmicro/lib/tmicro.n`. Functional units are declared in the same files as in which their instructions are modelled.

4.1 Memories

The TMICRO core has two memories `DM` and `PM`.

```
DM[2**16]<word,addr>
```

`DM` is a 16 bit wide data memory and has an address space of 64k locations. This memory is the default memory for the C compiler. All variables that are declared without a storage specifier are assigned to `DM`. The software stack that is needed to support C programs is organised in `DM`. The `DM` memory does not support memory wait states.

```
PM[2**16]<iword,addr>
```

`PM` is a 16 bit wide memory with an address space of 64k locations. This memory is intended to store the program code.

4.2 Registers

The TMICRO core has the following registers.

```
R[8]<word,uint3>
```

`R` is an 8 field, 16 bit wide general purpose register file. The ALU, shift and multiply instructions operate on this register. This register is also used as address register for indirect load and store instructions. Finally, some control instructions also make use of this register.

Following ports are defined for the `R` register file. The use of ports restricts the number of registers file accesses that can be executed in one cycle. It also reduces the hardware area.

`rrid` is used for reading in the `ID` stage.

`rre1` and `rse1` are used for reading in the `E1` stage.

rtid is used for writing in the ID stage.

rte1 is used for writing in the E1 stage.

For reading from and writing to R via these ports, mode rules rrid, rr, rs and rt have been defined.

PL. PL is the 16 bit low product register. It holds the 16 least significant bits of a 32 bit product.

PH. PH is the 16 bit high product register. It holds the 16 most significant bits of a 32 bit product.

LR<word>

LR is the link register. It is used to store the return address of a subroutine call (see [5] for more information on subroutine linkage).

SP<word>

SP is the stack pointer register (see [5] for more information on the software stack).

PC<addr>

PC is the program counter register.

LS[0..2]<word,uint2>

LE[0..2]<word,uint2>

LC[0..2]<word,uint2>

These registers are the hardware loop control registers. LS holds the loop start address, LE holds the loop end address, and LC holds the loop count. The TMICRO processor supports up to three levels of hardware controller loops. Consequently each loop control register is a register file with three fields. The loop control registers are indirectly addressed by the LF register.

LF<uint3>

LF is the loop flag register. The value in this register indicates how many hardware loops are active. LF has an initial value of 3 (after a reset, the content of LF will be 3) to indicate the absence of active hardware loops (see section 12.13.2).

Note that in the nML model, the loop control registers are not used in the __programmers_view__. In order to avoid warnings when processing the other views, an unconnected property is added.

```
property unconnected : LF, LS, LE, LC;
```

SR<word>

SR is the status register. It as a concatenation of the following fields.

CND<bool> (SR bit 0) CND is the condition register. It is set by compare instructions on the ALU, and read by conditional jump instructions.

CB<uint1> (SR bit 1) CB is the carry/borrow register. It is set when the ALU executes an add or subtract instruction. It is read when the ALU executes an add with carry or subtract with borrow instruction.

IE<uint1> (SR bit 3) IE is the interrupt enable register (see [6]).

`IM<ubyte>` (SR bits 8 to 15) `IM` (see [6]).

In order to avoid that interrupts are enabled after a reset, the `IE` bit must be reset expicitly. To achieve this a `hw_init` value of zero is specified for the complete status register.

`ILR<word>`

`ILR` is the interrupt link register (see [6]).

`ISR<word>`

`ISR` is the interrupt status register (see [6]).

4.3 Functional Units

The TMICRO core has the following functional units.

`alu`

`alu` is a 16 bit general purpose ALU.

`sh`

`sh` is a 16 bit shift unit.

`mul`

`mul` is a 16x16 multiplier unit that produces a 32 bit product.

`ag1`

`ag1` is a 16 bit address computation unit.

`dlflg`

`dlflg` is the hardware loop flag update unit.

`dlls`

`dlls` is the hardware loop start address unit.

5

The instruction pipeline

To meet the timing constraints imposed by the clock frequency, TMICRO executes instructions in a pipelined fashion. There are three pipeline stages: IF, ID and E1.

Pipeline stage		Actions
Instruction Fetch	IF	• During the instruction fetch stage, a new instruction is read from the program memory PM. The program memory address has been placed on the PM address bus at the end of the previous clock cycle (a synchronous memory is assumed).
Instruction Decode	ID	• During the decode stage, the instruction that was fetched in the previous cycle is decoded. • The effective address of load instructions is placed onto the DM address bus. • Address modifications and stand alone address computations on the address generation units are executed. • Unconditional control instructions modify the flow of control.
Execute 1	E1	• ALU, shift, multiply and division step operations are executed. • Move instructions take place in this stage. • The effective address of store instructions is placed onto the DM address bus. The source register is placed on the data bus. • For load instructions, the memory places the data on the data bus. The data is then latched into the destination register. • Conditional control instructions modify the flow of control.

The operation of the pipeline is depicted in figure 5.1. In this diagram, time progresses from left to right, and the program instructions progress from top to bottom. In cycle 3, instruction A is executing the actions of its execute stage E1. At the same time, instruction B is being decoded; and instruction C is being fetched from the program memory. Also in cycle 3, the next PC value of 3 is placed on the program memory address bus.

	cycle	1	2	3	4	5	6	7	8
	next PC	1	2	3	4	5	6	7	8
PC	Instruction								
0	**A**	IF	ID	E1					
1	**B**		IF	ID	E1				
2	**C**			IF	ID	E1			
3	**D**				IF	ID	E1		
4	**E**					IF	ID	E1	
5	**F**						IF	ID	E1

Figure 5.1: The instruction pipeline.

6

The Instruction Set

A basic instruction word of the TMICRO processor is 16 bit wide (data type `iword`). TMICRO supports both single word and multi word instructions. Most instructions are single word instructions, where one 16 bit instruction word encodes one or more actions that are performed by the processor. In the case of multi word instructions, the first instruction word also encodes the instruction. The second instruction word extends the first word with a 16 bit immediate value. There is one instruction, `doi`, which has three instruction words. The second and third instruction word never contain encoding bits.

The instruction set is partitioned into 8 groups, based the value of the three most significant bits of the instruction word. The following instruction groups are pre-defined.

`000` ALU, shift, multiply and division step instructions (see chapters 7, 8, 9).

`001` Register move instructions (see chapters 10), and control flow instructions (see 12).

`010` Load and store instructions with indirect addressing (see chapter 11).

`011` Load and store instructions with SP-relative addressing (see section 11.2).

Instruction groups with prefix `100`, `101`, `110` and `111` are available to add extensions to the TMICRO core. Also the instruction groups listed in the table above are not completely utilised, many combinations within those groups are still available to make extensions.

7

ALU instructions

In this chapter, the different ALU instructions are discussed. The TMICRO core has a 16 bit ALU. The instructions of this unit are intended to execute ANSI C arithmetic, logical and relational operations. All ALU operations execute in the E1 stage. Figure 7.1 depicts the ALU data path.

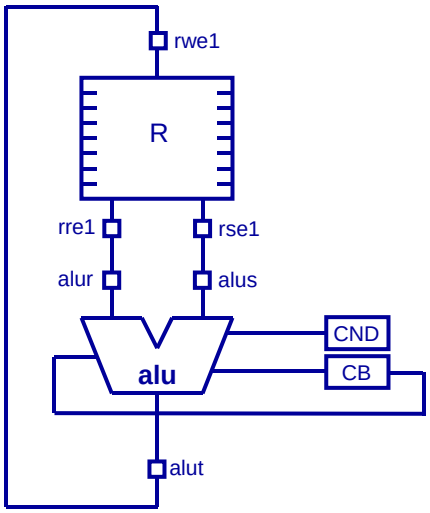


Figure 7.1: The ALU data path.

7.1 Arithmetic and Logical instructions

7.1.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	t			r			s			`add rt,rr,rs`
			0	0	0	1										`addc rt,rr,rs`
			0	0	1	0										`sub rt,rr,rs`
			0	0	1	1										`subb rt,rr,rs`
			0	1	0	0										`and rt,rr,rs`
			0	1	0	1										`or rt,rr,rs`
			0	1	1	0										`xor rt,rr,rs`

7.1.2 Description

The ALU supports the following arithmetic and logical operations: addition, addition with carry, subtraction, subtraction with borrow, bitwise AND, bitwise OR, and bitwise exclusive OR. The arithmetic and logical instructions are three register instructions; the operand registers are specified by the `r` and `s` fields, the result register is specified by the `t` field.

`add`	:	$\texttt{R[t]} \leftarrow \texttt{R[r]} + \texttt{R[s]}$,	$\texttt{CB} \leftarrow \text{carry\_out}(\texttt{R[r]} + \texttt{R[s]})$
`addc`	:	$\texttt{R[t]} \leftarrow \texttt{R[r]} + \texttt{R[s]} + \texttt{CB}$,	$\texttt{CB} \leftarrow \text{carry\_out}(\texttt{R[r]} + \texttt{R[s]} + \texttt{CB})$
`sub`	:	$\texttt{R[t]} \leftarrow \texttt{R[r]} - \texttt{R[s]}$,	$\texttt{CB} \leftarrow \text{carry\_out}(\texttt{R[r]} - \texttt{R[s]})$
`subb`	:	$\texttt{R[t]} \leftarrow \texttt{R[r]} + \sim \texttt{R[s]} + \texttt{CB}$,	$\texttt{CB} \leftarrow \text{carry\_out}(\texttt{R[r]} + \sim \texttt{R[s]} + \texttt{CB})$
`and`	:	$\texttt{R[t]} \leftarrow \texttt{R[r]}$ & `R[s]`	
`or`	:	$\texttt{R[t]} \leftarrow \texttt{R[r]} \mid \texttt{R[s]}$	
`xor`	:	$\texttt{R[t]} \leftarrow \texttt{R[r]}$ ^ `R[s]`	

7.1.3 nML model

[See file `tmicro/lib/alu.n`, rule `alu_rrr`]

In the nML model, the arithmetic and logical instructions are specified in the `alu_rrr` operation rule. First an enumerated type `arith_op` is defined to model the op codes of the ALU and other instructions. Each enumerator corresponds to one op code in the op code field $[12\ldots9]$. Note that a set of enumerators corresponds to the instructions of section 7.1.1. The other enumerators will be use further on.

```
enum alu_op { add,      addc,     sub,      subb,
              and,      or,       xor,      compare,
              asr,      lsr,      lsl,      unary,
              mul,      equal     min,      max};
```

Next, a functional unit `alu`, two input ports `alur`, `alus` and an output port `alut` are declared.

```
fu alu;
trn alur<word>;
trn alus<word>;
trn alut<word>;
```

The `alu_rrr` rule has four parameters: an op code `op` to select the desired function, two parameters `r` and `s` to select the operand registers and one parameter `t` to select the result register. The `action` attribute specifies the register transfers that take place for this rule. All register transfers occur in the `E1` pipeline stage. The `r` and `s` fields of the register file `R` are assigned to the ports `alur` and `alus`. Based on the op code, one of the primitive operations is selected, and the result is assigned to the `alut` port. Finally, the `alut` port is assigned to the `t` field of `R`.

```
opn alu_rrr(op: alu_op, t: rt, r: rr, s: rs)
{
    action {
        stage E1:
        alur = r;
        alus = s;
        switch (op) {
        case add:    alut = add (alur,alus,CB)    @alu;
        case addc:   alut = addc(alur,alus,CB,CB) @alu;
        case sub:    alut = sub (alur,alus,CB)    @alu;
        case subb:   alut = subb(alur,alus,CB,CB) @alu;
        case and:    alut = andw(alur,alus)       @alu;
        case or:     alut = orw (alur,alus)       @alu;
        case xor:    alut = xorw(alur,alus)       @alu;
        }
        t = alut;
    }
    syntax : op " " t "," r "," s;
```

```
    image  : op::t::r::s;
}
```

The operations of the ALU are declared as primitive functions in the `tmicro.h` header file [5]. For example the addition with carry is specified as follows:

```
primitive word add (word,word,uint1,uint1&) property(commutative);
```

This declaration states that `add()` is a function that takes two `word` operands and a carry-in operand of type `uint1`, and produces a `word` result and a carry bit of type `uint1`. The `primitive` keyword means that this is a primitive operation that is executed by the hardware of the processor.

The `syntax` attribute specifies the assembly syntax for this rule. For example, the instruction that subtracts register `R[3]` from `R[6]`, and places the result in `R[0]` has the following syntax.

```
sub r0,r6,r3
```

The `image` attribute specifies the binary encoding of the `alu_rrr` instructions. In this case, this a the concatenation of 4 op code bits, the three bits that specify the result register, the bits for the `r` operand, and the bits for the `s` operand. For the `sub r0,r6,r3` instruction, the image is the following.

```
0000010000110011
```

7.2 Compare instructions

7.2.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	1	1	1	0	0	0		r			s		`lt rr,rs`
							0	0	1							`ltu rr,rs`
							0	1	0							`le rr,rs`
							0	1	1							`leu rr,rs`
							1	0	0							`gt rr,rs`
							1	0	1							`gtu rr,rs`
							1	1	0							`ge rr,rs`
							1	1	1							`geu rr,rs`

7.2.2 Description

The ALU supports the following compare operations on two fields of the `R` register file: signed less than (`lt`), unsigned less than (`ltu`), signed less than or equal (`le`), unsigned less than or equal (`leu`), signed greater than (`gt`), unsigned greater than (`gtu`), signed greater than or equal (`ge`), unsigned greater than or equal (`geu`), equal (`eq`) and not equal (`ne`). The result is stored in the `CND` register.

`lt` : $\texttt{CND} \leftarrow \texttt{R[r]} < \texttt{R[s]}$, signed compare
`ltu` : $\texttt{CND} \leftarrow \texttt{R[r]} < \texttt{R[s]}$, unsigned compare
`le` : $\texttt{CND} \leftarrow \texttt{R[r]} \leq \texttt{R[s]}$, signed compare
`leu` : $\texttt{CND} \leftarrow \texttt{R[r]} \leq \texttt{R[s]}$, unsigned compare
`gt` : $\texttt{CND} \leftarrow \texttt{R[r]} > \texttt{R[s]}$, signed compare
`gtu` : $\texttt{CND} \leftarrow \texttt{R[r]} > \texttt{R[s]}$, unsigned compare
`ge` : $\texttt{CND} \leftarrow \texttt{R[r]} \geq \texttt{R[s]}$, signed compare
`geu` : $\texttt{CND} \leftarrow \texttt{R[r]} \geq \texttt{R[s]}$, unsigned compare
`eq` : $\texttt{CND} \leftarrow \texttt{R[r]} = \texttt{R[s]}$
`ne` : $\texttt{CND} \leftarrow \texttt{R[r]} \neq \texttt{R[s]}$

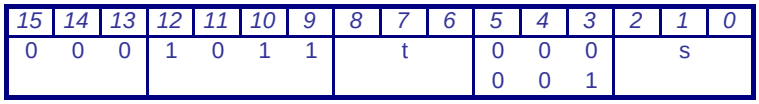
Signed compares interpret the 16 bit operands as signed values. This means that a value that starts with a 1 bit (a negative value) is smaller that a value of which the most significant but is 0 (which means it is a positive value). Unsigned compares interpret the 16 bit operands as unsigned values. This means that a value with an MSB of 1 bit is larger that a value with and MSB of 0.

7.2.3 nML model

[See file `tmicro/lib/alu.n`, rule `compare_rrr,equal_rr`]

7.3 Unary ALU instructions

7.3.1 Format and syntax



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	0	1	1	t			0	0	0	s			`cmpl rt,rs`
										0	0	1				`xs rt,rs`

7.3.2 Description

The ALU supports the following unary (two register) operations: logical complement and sign extraction. The operand register of these instructions is specified by the `s` field, the result register is specified by the `t` field.

`cmpl` : $\texttt{R[t]} \leftarrow \sim \texttt{R[s]}$

`xs` : $\texttt{R[t]} \leftarrow \begin{cases} -1 & \text{if } \texttt{R[s]}_{15} = 1 \\ 0 & \text{otherwise} \end{cases}$

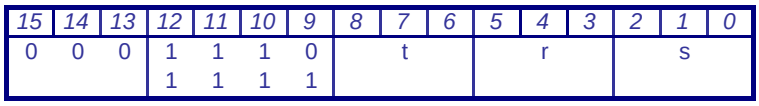
The `xs` instruction produces a value 0 in `rt` when `rs` contains a positive value (including zero); and produces a value -1 when `rs` in negative.

7.3.3 nML model

[See file `tmicro/lib/alu.n`, rule `alu_rr`]

7.4 Conditional move instructions

7.4.1 Format and syntax



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	1	1	0	t			r			s			`min rt,rr,rs`
			1	1	1	1										`max rt,rr,rs`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	0	0	1	t			r			s			`sel rt,rr,rs`

7.4.2 Description

The ALU supports the three conditional move operations: minimum value selection, maximum value selection and conditional move based in the `CND` bit. The operand registers are specified by the `r` and `s` fields, the result register is specified by the `t` field.

`min` : $\texttt{R[t]} \leftarrow \texttt{R[r]} < \texttt{R[s]} \,?\, \texttt{R[r]} : \texttt{R[s]}$

`max` : $\texttt{R[t]} \leftarrow \texttt{R[r]} > \texttt{R[s]} \,?\, \texttt{R[r]} : \texttt{R[s]}$

`sel` : $\texttt{R[t]} \leftarrow \texttt{CND} \,?\, \texttt{R[r]} : \texttt{R[s]}$

7.4.3 nML model

[See file tmicro/lib/alu.n, rule minmax_rrr,select_rrr]

7.5 The divide step instruction

7.5.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	1	0	0	r			xxx		

```
ds (r7,d6), rr
```

7.5.2 Description

The ALU supports the an instruction that implements a basic step of the non-restoring division algorithm. The denominator is located in the r field of the R register. The PL and PH registers are used to store the intermediate result of the division. Initially, the numerator is located in PL, and PH is set to zero. After 16 invocations of the ds instruction, the quotient is located in PL and the remainder is in PH.

7.5.3 nML model

[See file tmicro/lib/alu.n, rule div_r]

7.6 Definitions of the ALU primitive functions

[See file tmicro/lib/tmicro.p]

The bit true behaviour of the ALU primitives is defined in the primitive definition file tmicro.p. As a first example, consider the andw() primitive. It is defined by applying the bitwise AND operator & to the arguments a and b.

```
word andw(word a, word b) { return a & b; }
```

As a second example, the definition of the add() primitive is given. First, the 16 bit operands a and b are converted to 17 bit values aa and bb by applying a zero extension. Next, a 17 bit addition is done. The 16 least significant bits of this sum define the return value of the add() primitive. The most significant bit of the 17 bit sum is the carry output of the add() primitive.

```
word add(word a, word b, uint1& co)
{
    int17_t aa = (uint16_t)a;
    int17_t bb = (uint16_t)b;
    int17_t rr = aa + bb;
    co = rr[16];
    return rr[15:0];
}
```

8

Shift instructions

The TMICRO core has a 16 bit shift unit. The instructions that are executed on this unit are intended to execute ANSI C shift operations. All shift operations execute in the E1 stage. Figure 8.1 depicts the shifter data path. In this chapter, the different shift instructions are discussed.

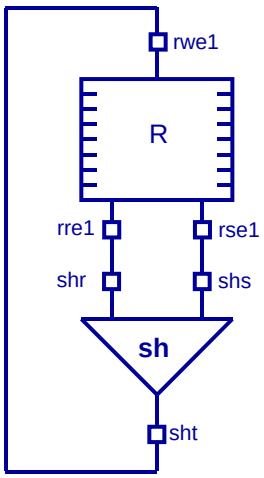


Figure 8.1: The shifter data path.

8.1 Shift instructions

8.1.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	t			r			s		
			1	0	0	1									
			1	0	1	0									

```
asr rt,rr,rs
lsr rt,rr,rs
lsl rt,rr,rs
```

8.1.2 Description

The shift unit supports the following operations: arithmetic shift right, logical shift right and logical shift left. The shift instructions are three register instructions; the operand register is specified by the r field, the shift factor register is specified by the s field, the result register is specified by the t field.

$$\texttt{asr}: \quad \texttt{R[t]} \leftarrow \texttt{R[r]}_{\text{signed}} >> \texttt{R[s]}_{3\ldots0}$$
$$\texttt{lsr}: \quad \texttt{R[t]} \leftarrow \texttt{R[r]}_{\text{unsigned}} >> \texttt{R[s]}_{3\ldots0}$$
$$\texttt{lsl}: \quad \texttt{R[t]} \leftarrow \texttt{R[r]}_{\text{unsigned}} << \texttt{R[s]}_{3\ldots0}$$

The `asr` instruction extends the result at the MSB side with sign bits. The `lsr` instruction extends the result at the MSB side with zero bits. The `lsl` instruction extends the result at the LSB side with zero bits. The actual shift factor corresponds to the 4 least significant bits of the value in `R[s]`. The range of the shift factor is therefore $[0\ldots15]$.

8.1.3 nML model

[See file `tmicro/lib/alu.n`, rule `shift_rrr`]

8.2 Definitions of the shift primitive functions

[See file `tmicro/lib/tmicro.p`]

The bit true models of the shift primitives are straightforward. The shift primitives are mapped onto the C shift operators of the corresponding sign encoding. The arithmetic shift primitive is mapped onto the C signed shift operator, the logical shift primitives are mapped onto the C unsigned shift operator. The unsigned shift operator is selected by converting the `a` operand to a 16 bit unsigned type. For the shift factor, the 4 least significant bits of the `b` operand are used.

```
word asr(word a, word f) { return a >> f[3:0]; }
word lsr(word a, word f) { return (uint16_t)a >> f[3:0]; }
word lsl(word a, word f) { return (uint16_t)a << f[3:0]; }
```

9

Multiply instructions

The TMICRO core has a 16 by 16 bit multiplier, which produces a 32 bit product. The instructions that are executed on this unit are intended to execute ANSI C integer multiplication operations. All multiplications execute in the E1 stage. The multiplier operates on the R register file, and produces a result in the PH and PL registers. Figure 9.1 depicts the multiplier data path. In this chapter, the different multiply instructions are discussed.

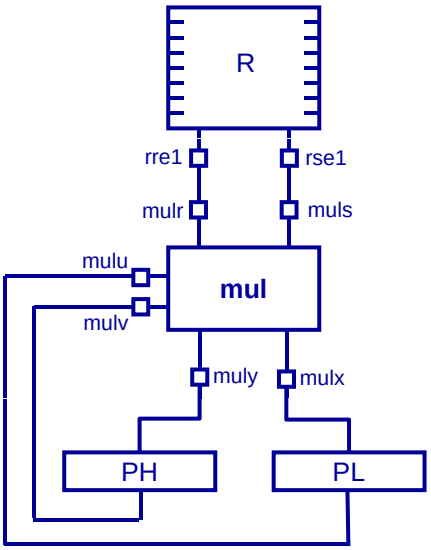


Figure 9.1: The multiplier data path.

9.1 Multiply instructions

9.1.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	0	0	r			s		
							0	0	1						
							0	1	1						

```
mulss (r7,r6),rr,rs
muluu (r7,r6),rr,rs
macl  (r7),rr,rs
```

9.1.2 Description

The multiplier unit supports signed and unsigned multiplications and multiply accumulate operations, as indicated in the table below. The 16 bit operands are read from the `r` and `s` fields of the `R` register file. The 32 bit product is stored is as two separate 16 bit words in the `PH` and `PL` registers.

`mulss`	$\texttt{(PH,PL)} \leftarrow \texttt{R[r]}_{\text{signed}} \times \texttt{R[s]}_{\text{signed}}$
`muluu`	$\texttt{(PH,PL)} \leftarrow \texttt{R[r]}_{\text{unsigned}} \times \texttt{R[s]}_{\text{unsigned}}$
`macl`	$\texttt{(PH)} \leftarrow \texttt{(PH)} + (\texttt{word})(\texttt{R[r]}_{\text{signed}} \times \texttt{R[s]}_{\text{signed}})$

The `mulss` and `muluu` instructions are intended to execute 16 bit signed and unsigned integer multiplications. The `macl` instruction is intended for the software emulation of 32 bit integer multiplication.

9.1.3 nML model

[See file `tmicro/lib/alu.n`, rule `mul_rr`]

9.2 Definitions of the multiply primitive functions

[See file `tmicro/lib/tmicro.p`]

The bit true models of the multiply primitives are straightforward. The multiply primitives are mapped onto the C multiply operators of the corresponding sign encoding.

```
void mulss(word a, word b, word& rl, word& rh)
{
    int32_t p = a * b;
    rl = p[15:0];
    rh = p[31:16];
}

void muluu(word a, word b, word& rl, word& rh)
{
    int32_t p = (uint16_t)a * (uint16_t)b;
    rl = p[15:0];
    rh = p[31:16];
}
```

The nML action attribute of the `macl` instruction combines the `mulss()` and `add()` primitives : the first output of the `mulss()` primitive is added to the `PH` register.

10

Move instructions

In this chapter, the register move instructions and immediate load instructions of the TMICRO processor are discussed.

10.1 The word register move instruction

10.1.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	dst				src			

```
mv dst,src
```

.

10.1.2 Description

This instruction moves the content of one register of type `word` to another register of type `word`. The registers that are supported are listed in table 10.1. The register transfer takes place in pipeline stage `E1`.

wreg				register
0	r			R[r]
1	0	1	0	SP
1	0	1	1	LR
1	1	0	0	SR
1	1	0	1	ILR
1	1	1	0	ISR

Table 10.1: Possible word registers.

10.1.3 nML model

[See file `tmicro/lib/move.n`, rule `mv_wreg`]

10.2 The load word immediate instruction

10.2.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	0	0	0	1	wreg			
imm															

```
mvi wreg,imm
```

10.2.2 Description

This instruction loads a register of type `word` with a signed immediate value in the range $[-32768 \ldots 32767]$. The registers that are supported are listed in table 10.1. The register is written in pipeline stage `E1`.

This instruction is a two word and two cycle instruction. The first instruction word consists of op code bits and a field that selects the destination register. The second instruction word is the 16 bit immediate value. The pipeline diagram below depicts the two cycle behaviour of the `mvi` instruction. [h]

	cycle	1	2	3	4	5	6	7	8
	issue_sig	1	1	0	1	1	1		
	mword_sig	0	0	1	0	0	0		
	R[0]	?	?	?	?	imm	imm		
PC	Instruction								
5	**A**	IF	ID	E1					
6	**mvi r0,<imm>**		IF	ID	E1				
7	**<imm>**			IF	-	-			
8	**B**				IF	ID	E1		

10.2.3 nML model

[See file `tmicro/lib/move.n`, rule `mvi_wreg_word`]

10.3 The load byte immediate instruction

10.3.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	imm								wreg			

```
mvib wreg,imm
```

10.3.2 Description

This instruction loads a register of type `word` with a signed immediate value of type `byte` which has a range $[-128 \ldots 127]$. The value of type `byte` is sign extended to obtain a value of type `word`. The registers that are supported are listed in table 10.1. The register is written in pipeline stage `E1`.

10.3.3 nML model

[See file `tmicro/lib/move.n`, rule `mvi_wreg_byte`]

11

Load and store instructions

The TMICRO core has two separate memories: the data memory DM and the program memory PM. The first set of load and store instructions operate on DM. In addition, there is another set of load and store instructions for the program memory PM.

The TMICRO core has one address generation unit (see figure 11.1). Indirect addresses are taken from the register file R. The address generation unit can increment or decrement the address using post modification.

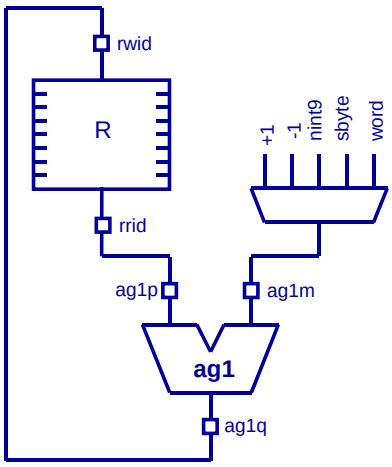


Figure 11.1: The address generation unit.

11.1 Timing of load and store instructions

The load and store instructions of the DM memory memory have a fixed timing, wait states are not supported. The following table is an overview of the ports of the memory interface. The port names correspond to the names that are generated by the GO tool.

dm_read_r	input	data port for load.
dm_read_extad	output	address port, shared for load and store.
dm_read_r_cntrl	output	control signal for load.
dm_write_w	output	data port for store.
dm_write_w_cntrl	input	control signal for store.

The timing diagram of the load instruction is depicted in figure 11.2. A load instruction starts in the ID stage. The address computation unit places the address on the dm_read_extad port and the

dm_read_r_cntrl signal is asserted. At the end of the ID stage, the address is latched into the memory. At the end of the E1 stage, the data from the dm_read_r port of the memory is latched in the destination register.

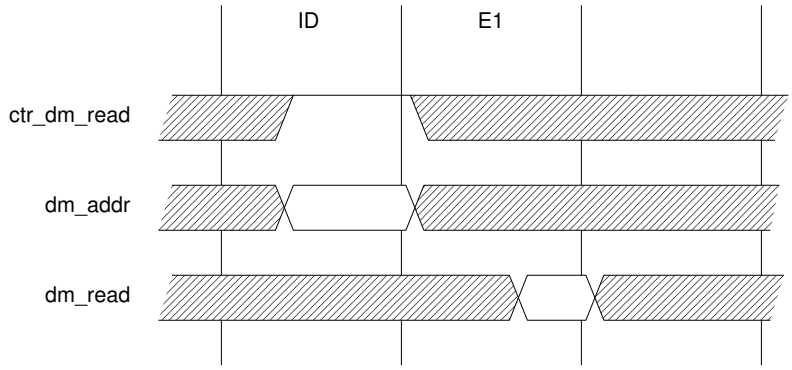


Figure 11.2: Timing diagram of a load instruction.

The timing diagram of a store instruction is depicted in figure 11.3. . The address computation also takes place in the ID stage. Since the address is only needed in the E1 stage it is latched into the dm_addr_pipe pipeline register. In the E1 stage the address is put on the dm_read_extad port, the data is put on the dm_write_w port and the dm_write_w_cntrl signal is asserted. At the end of the E1 stage these values are latched into the memory.

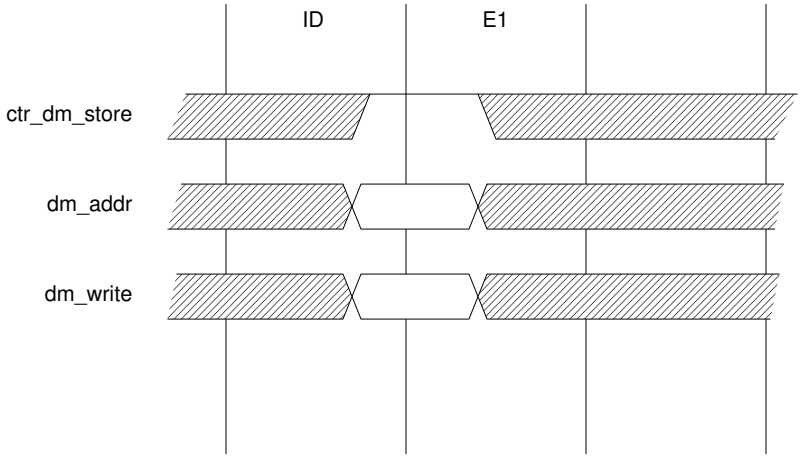


Figure 11.3: Timing diagram of a store instruction.

11.2 SP indexed load/store instructions

11.2.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0 1	offs								wreg			

```
ld wreg,dm(sp+offs)
st wreg,dm(sp+offs)
```

.

11.2.2 Description

These instructions load or store a register of type `word` from or to the `DM` memory. The registers that are supported are listed in table 10.1. The address is given by the value of the stack pointer plus a negative offset. The index addition is executed on the AGU.

`ld` : `wreg` ← `DM[SP+offs]`, with `offs` in the range $[-256 \ldots -1]$
`st` : `DM[SP+offs]` ← `wreg`, with `offs` in the range $[-256 \ldots -1]$

These instructions are intended for the C compiler, to save registers on the stack frame as part of the context switch of a function call or interrupt. For more information of the stack frame, consult the Chess manual [5].

11.2.3 nML model

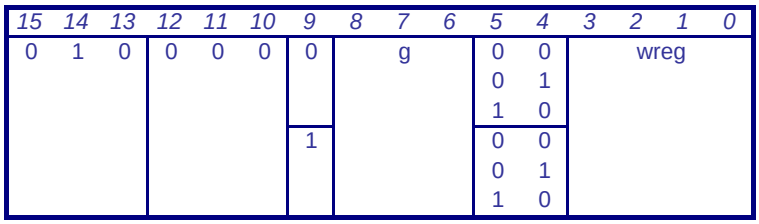
[See file `tmicro/lib/load_store.n`, rule `load_store_wreg_sp_indexed`]

Notice that only the lower 8 bits of the 9 bit `offs` field are encoded in the instruction. The ninth bit is assumed to be always one.

```
image : ls::offs[one 7..0]::rr;
```

11.3 Load/store instructions with linear addressing

11.3.1 Format and syntax



11.3.2 Description

These instructions load or store a register of type `word` from or to the `DM` memory. The registers that are supported are listed in table 10.1. The address is taken from a field of the `R` register file. Optionally, this field is post-incremented or post-decremented.

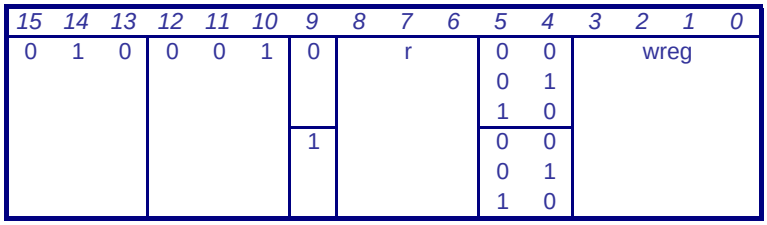
`ld` : `wreg` ← `DM[R[r]]`, optionally `R[r]` ← `R[r]+1` or `R[r]` ← `R[r]-1`
`st` : `DM[R[r]]` ← `wreg`, optionally `R[r]` ← `R[r]+1` or `R[r]` ← `R[r]-1`

11.3.3 nML model

[See file `tmicro/lib/load_store.n`, rule `load_store_wreg_indirect`]

11.4 Load/store instructions on the program memory

11.4.1 Format and syntax



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	1	0	r			0	0	wreg				ld wreg,pm(rr)
										0	1					ld wreg,pm(rr++)
										1	0					ld wreg,pm(rr--)
						1				0	0					st wreg,pm(rr)
										0	1					st wreg,pm(rr++)
										1	0					st wreg,pm(rr--)

11.4.2 Description

These instructions load or store a register of type `word` from or to the `PM` memory. The registers that are supported are listed in table 10.1. The address is taken from a field of the `R` register file. Optionally, this field is post-incremented or post-decremented.

```
ld    :  wreg ← PM[R[r]],        optionally R[r] ← R[r]+1 or R[r] ← R[r]-1
st    :  PM[R[r]] ← wreg,        optionally R[r] ← R[r]+1 or R[r] ← R[r]-1
```

Since the program memory is a single ported memory, it is impossible to execute a `PM` load or store, and simultaneously fetch the next instruction.

A `PM` load occupies the `pm_addr` bus in the `ID` stage and the `pm_read` bus in the `E1` stage. This conflicts with a subsequent instruction fetch, as shown on the pipeline diagram below.

	cycle	1	2	3	4	5	6	7	8
	issue_sig	1	0	1	1	1	1		
PC	Instruction								
5	`ld r1,pm(r1)`	IF	ID	E1					
6	`A`	(PF)	IF	-	ID	E1			
7	`B`		~~(PF)~~	~~IF~~	IF	ID	E1		
8	`C`					IF	ID	E1	

`(PF)` indicates the cycle in which the PCU send the address of the next instruction to `PM`; `IF` indicates the cycle in which the next instruction is available on the `pm_read` bus.

When the `PM` load is in the `ID` stage, instruction 7 would normally be fetched. In order to avoid the simultaneous access conflict, the instruction fetch must be postponed for one cycle. This is achieved by making the `PM` load a `cycles(2)` instruction.

A `PM` store occupies the `pm_addr` bus in the `E1` stage. This also conflicts with a subsequent instruction fetch, as shown on the pipeline diagram below.

	cycle	1	2	3	4	5	6	7	8
	issue_sig	1	0	0	1	1	1		
PC	Instruction								
5	`st r1,pm(r1)`	IF	ID	E1					
6	`A`	(PF)	IF	-	ID	E1			
7	`B`		~~(PF)~~	~~(PF)~~	~~IF~~	IF	ID	E1	
8	`C`						IF	ID	E1

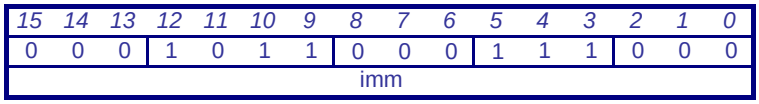
When the `PM` store is in the `E1` stage, instruction 8 would normally be fetched. In order to avoid the simultaneous access conflict, the instruction fetch must be postponed for two cycles. This is achieved by making the `PM` store a `cycles(3)` instruction.

11.4.3 nML model

[See file `tmicro/lib/load_store.n`, rule `load_store_pm_wreg_indirect`]

11.5 The stack pointer addition

11.5.1 Format and syntax




```
addw sp,imm
```


```
addb sp,imm
```

11.5.2 Description

The `add` instruction adds a 16 bit signed immediate value to stack pointer register. This instruction consists of two words. The first word specifies the instruction, the second word holds the immediate value. The `addb` instruction adds an 8 bit signed immediate value to stack pointer register. This is a single word instruction. The additions are executed on the linear AGU, in the `ID` stage.

Both instructions are used to allocate a stack frame in the context of a subroutine call. For more information on the stack frame, consult the Chess modelling manual [5].

11.5.3 nML model

[See file `tmicro/lib/load_store.n`, rule `add_sp_word,add_sp_byte`]

12

Control instructions

Ordinary instructions increment the program counter so that the next instruction is fetched from the program memory. Control instructions change the flow to another part of the program. They include jumps, subroutine call and return, hardware loops and interrupts. The TMICRO control instructions are described in the file `control.n`. The nML model of the control flow instructions is an abstract, property based, model [5]. In some cases, the nML model consists of two views: a `__programmers_view__`, which is sufficient for the compiler, linker and assembler tools; and a more detailed view, with some additional register transfer operations, and which is used for the ISS and RTL generation. In addition to the nML code of the control flow instructions, there is also a behavioral model of the program control unit (PCU) in the file `tmicro_pcu.p`.

12.1 Description of the PCU

The purpose of the program control unit is to advance the flow of the program. This involves fetching instructions from the program memory and issuing these instructions in a pipelined way – a new instruction is fetched while the previous instruction is decoded – as was shown in figure 5.1. In this section, the operation of the PCU is described. More information the modeling of the PCU can be found in [4].

The TMICRO PCU is depicted in figure 12.1. The two functions that are depicted, `user_issue()` and `user_next_pc()` are defined in the `tmicro_pcu.p` file. The main registers that are affected by the PCU are the program counter register `PC`, the fetch buffer register `reg_f_instr`, and the instruction register `IR`. There are actually two versions of the instruction register, that hold the instructions that are present in the pipeline. The program memory `PM` of TMICRO is a synchronous memory. This is indicated in figure 12.1 by the `pm_addr` register. At the end of one cycle a new address is registered in `pm_addr` and in the next cycle the instruction at that address appears on the `pm_read` port.

The default behavior of the PCU is that each cycle the following actions take place. This situation occurs when the processor executes a straight line sequence of single cycle instructions.

- An instruction is fetched from `PM`. This instruction is stored in `IR(ID)` and in `reg_f_instr`.
- The instructions that are present in the `IR` registers are decoded. For each version of `IR`, the control signals that enable the actions of the corresponding pipeline stage are decoded.
- The program counter value is incremented. The new value is stored in the `PC` and `pm_addr` registers.

Deviations from this basic behavior occur when multi cycle instructions, delay slot instructions, multi word instructions or control flow instructions are executed; when an interrupt occurs; or when a debug request arrives.

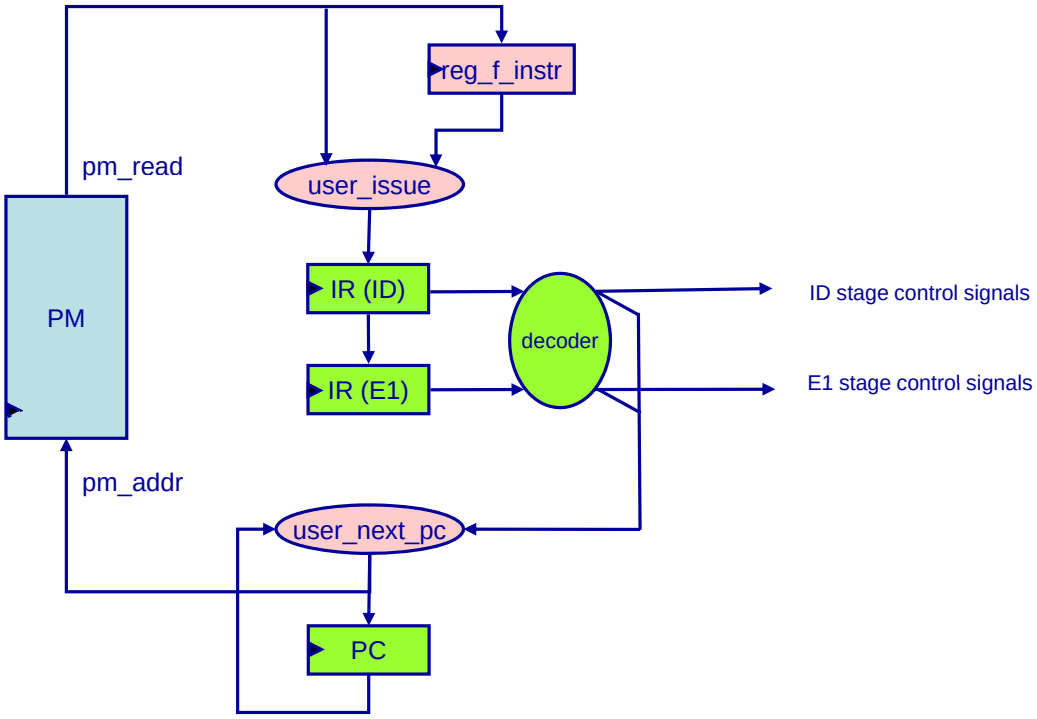


Figure 12.1: The program control unit (PCU).

12.1.1 Multi cycle instructions

A multi cycle instruction is indicated in nML by means of the instruction property `cycles(n)`. When a multi cycle instruction is issued, the next instruction is issued n cycles later. The PCU controls the issuing of multi cycle instructions by means of the boolean signal `issue_sig()`. When executing single cycle instructions, this signal is always 1. When a multi cycle instruction is decoded, `issue_sig()` signal becomes 0. When the specified number of cycles is passed, the signal becomes 1 again. The signal will be low for $n-1$ cycles, starting from the cycle where the instruction is in the decode stage. An instruction that was fetched may only be issued when this signal is 1. When `issue_sig()` is 0 the instruction that was fetched in the previous cycle is not issued but is kept in the fetch buffer register `reg_f_instr`.

An example of a multi cycle instruction is the `jr` instruction, which takes two cycles. The timing diagram of this instruction is shown in section 12.3.

12.1.2 Delay slot instructions

When a jump instruction is executed, it typically takes a few cycles before the target instruction has been fetched. In these cycles other instructions could be executed. However, this is not so when using the `cycles(n)` property. An alternative approach is to indicate that cycles after a jump instruction can be used to executed other instructions. This can be specified in nML by the `delay_slots(n)` instruction property. For more information on this property, see [5]. No extra signals are needed in the PCU. Instruction execution is continued in a normal way while the jump is taken.

An example of an instruction with delay slots is the `jrd` instruction. The timing diagram of this instruction is shown in section 12.2.

12.1.3 Multi word instructions

A multi word instruction consists of multiple 16 bit instruction words. On the TMICRO core, all multi word instructions are two word instructions with all encoding bits in the first word and an immediate value in the second word.

A multi word instruction is fetched in parts. The individual instruction words are loaded in consecutive cycles. The PCU controls the issuing of multi word instructions by means of the boolean signal `mword_sig`. This signal is 1 when the second word of a two-word instruction is fetched. The PCU must not issue this second instruction word as a new instruction, but must attach it as an immediate value to the current instruction.

Note that the nML processor property `default_cycles: words` is specified. As a result, all multi-word instructions are automatically defined as multi-cycle as well.

An example of a multi word instruction is the `mvi` instruction, which has two instruction words. The timing diagram of this instruction is shown in section 10.2.

12.1.4 Control flow instructions

The behavior of the PCU when control flow instructions are executed is discussed in the remaining sections of this chapter.

12.1.5 Interrupts

The behavior of the PCU when interrupts occur is discussed in [6].

12.2 The `jrd` instruction

12.2.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	0	of							

```
jrd of
```

12.2.2 Description

The `jrd` instruction is a unconditional relative jump with one delay slot. The instruction unconditionally transfers control from the current `PC` to a new address at `PC + of`, where `of` is a signed 8 bit offset.

		cycle	1	2	3	4	5	6	7	8
		issue_sig	1	1	1	1				
		diid	0	1	0	0				
		next PC	6	10	11	12				
PC	Instruction									
5	`jrd T`		IF	ID						
6	`ds1`			IF	ID	E1				
7										
8										
9										
10	`T: t1`				IF	ID	E1			

Due to the pipeline, the change of flow is not immediate. As shown in the pipeline diagram, there is an intermediate cycle between the issue of the jump instruction and the issue of the first instruction `t1` at the

jump target T. This is because the jump is executed in the ID stage. In this stage, the target address PC + of is written to the PC, and the next cycle (cycle 3) the target instruction is fetched. In the mean time one more instruction, ds1, has been fetched. In the case of a jrd jump, ds1 is executed as a delay slot instruction.

12.2.3 nML model

[See file tmicro/lib/control.n, rule jrd]

The jrd instruction is modeled by the jrd rule. The action attribute states that the jump primitive is executed in stage ID, and that the offset parameter of is assigned to the offs transitory.

```
action { stage ID: jump(offs=of); }
```

In the image attribute, some additional properties are given.

```
image  : of, delay_slots(1), chess_pc_offset(1), class(jump);
```

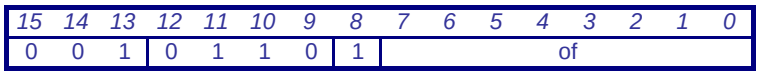
The delay_slots property indicates that the instruction that comes after jrd in the program, must be issued right after jrd is issued. The CHESS compiler will schedule 1 instruction after the jump. If there is no useful instruction in the program, then a nop instruction must be inserted in the code.

By default, the CHESS compiler assumes that the jump is relative to the address of the instruction. When the addition PC + of takes place in the ID stage, the PC has advanced by one. The chess_pc_offset property indicates to the compiler that it must take this offset of 1 into account in the immediate of value. For the example shown in the timing diagram, of must get the value 4.

The instruction class annotation jump will be used in the definition of hazard rules that involve this jump instruction (see section 13.2.3).

12.3 The jr instruction

12.3.1 Format and syntax



12.3.2 Description

The jr instruction is a unconditional relative jump without delay slot. The instruction unconditionally transfers control from the current PC to a new address at PC + of, where of is a signed 8 bit offset.

Due to the pipeline, the change of flow is not immediate. As shown in the pipeline diagram, there is an intermediate cycle between the issue of the jump instruction and the issue of the first instruction t1 at the jump target T. This is because the jump is executed in the ID stage. In this stage, the target address PC + of is written to the PC, and the next cycle (cycle 3), the target instruction is fetched. In the mean time one more instruction, x1, has been fetched. In the case of a jr jump, this instruction is not executed. Therefore, no instruction is issued in cycle 2.

12.3.3 nML model

[See file tmicro/lib/control.n, rule jr]

The jr instruction is modeled by the jr rule. This operation rule is similar to that of the jrd instruction, except for the image attribute.

```
image  : of, cycles(2), chess_pc_offset(1), class(jump);
```

		cycle	1	2	3	4	5	6	7	8
		issue_sig	1	0	1					
		next PC	6	10	11	12				
PC	Instruction									
5	**jr T**		IF	ID	E1					
6	**x1**			IF	-	-	-			
7										
8										
9										
10	**T: t1**				IF	ID	E1			

The `cycles` property indicates that `jr` is a two cycle instruction, no instruction is issued in the cycle after `jr` is issued.

12.4 The `jcr` instruction

12.4.1 Format and syntax

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	
0	0	1	0	1	1	1	1	of								`jcr of`

12.4.2 Description

The `jcr` instruction is a conditional relative jump. The instruction conditionally transfers control from the current `PC` to a new address at `PC` + `of`, where `of` is a signed 8 bit offset. The jump is taken when the `CND` register contains a 1. Because the `CND` register is written in the `E1` stage, the conditional jump instruction also read this register in `E1`. The pipeline diagram for a taken jump is as follows.

		cycle	1	2	3	4	5	6	7	8
		issue_sig	1	1	1	1	1			
		tcc	0	0	1	0	0			
		jcr	0	0	1	0	0			
		next PC	6	7	10	11	12			
PC	Instruction									
5	**jcr T**		IF	ID	E1					
6	**n1**			IF	~~ID~~	-				
7	**n2**				IF	-				
8	**n3**									
9	**n4**									
10	**T: t1**					IF	ID	E1		

In cycle 2, the jump instruction is decoded. At the same time the next instruction `n1` is fetched. Even though the condition is not yet known, the next instruction is speculatively issued. In cycle 3, the condition is checked and seen to be true. At this point, instruction `n1` is killed: the actions that are executed in the `ID` stage are not allowed to update the processor state. In cycle 3, the address of the target instruction is computed and written to the `PC` and placed on the program memory address bus. In cycle 4, the target instruction is fetched.

The jump is not taken when the `CND` register contains a 0. The pipeline diagram for a not taken jump is as follows.

In cycle 2 the operation is the same as for the taken case: instruction `n1` is fetched and issued In cycle 3, the condition is seen to be false. Instruction `n1` is now not killed. Instruction `n2` is fetched and issued.

		cycle	1	2	3	4	5	6	7	8
		issue_sig	1	1	1	1	1			
		tcc	0	0	0	0	0			
		jcr	0	0	1	0	0			
		next PC	6	7	8	9				
PC	Instruction									
5	**jcr T**		IF	ID	E1					
6	**n1**			IF	ID	E1				
7	**n2**				IF	ID	E1			
8	**n3**					IF	ID	E1		
9	**n4**									
10	**T: t1**									

12.4.3 nML model

[See file `tmicro/lib/control.n`, rule `jcr`]

This instruction is modeled by the `jcr` rule. In the action attribute, two transitories are set, these are used in the behavioral model of the PCU.

`tcc` The value of the condition register `CND` is assigned to the `tcc` transitory. The PCU used this information to determine if the jump is taken or not.

`jcr` The *jump conditional relative* transitory is set to one. This signal controls the killing of `n1` in cycle 3 in the taken case. The condition

$$\neg(\texttt{jcr} \wedge \texttt{tcc})$$

overrules the `issue_sig`, which is always 1 for the `jcr` instruction. (In case of a `cycles(m|n)` annotation, `issue_sig` is 0 for $n-1$ cycles.)

In the image attribute, the `cycles` and `chess_pc_offset` properties are set as follows.

```
image : of, cycles(3|1), chess_pc_offset(2), class(jump);
```

The `cycles(3|1)` annotation indicates that the instruction takes 3 cycles when the jump is taken, and 1 cycles otherwise. As can be seen in the above pipeline diagrams, the `PC` is incremented twice, when `jcr` is in the `IF` and `ID` stages, hence the `chess_pc_offset(2)` annotation.

12.5 The `j` instruction

12.5.1 Format and syntax

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	
0	0	1	0	1	1	1	0	1	0	0	0	0	0	0	0	`j tg`
tg																

12.5.2 Description

The `j` instruction is a unconditional absolute jump. This instruction unconditionally transfers control from the current `PC` to a new address given by the `tg` parameter. It is a two word instruction: the first word encodes the instruction, the second word contains the target address.

Due to the pipeline, the change of flow is not immediate. As shown in the pipeline diagram, there is one cycle between the jump instruction and the first instruction `t1` at the jump target `T`. This is because the jump is executed in the `ID` stage. In this stage, the target address `tg` is written to the `PC`, and the next cycle (cycle 3), the target instruction is fetched. In the intermediate cycle, the target address is fetched.

	cycle	1	2	3	4	5	6	7	8
	issue_sig	1	0	1					
	next PC	6	10	11					
PC	Instruction								
5	**j T**	IF	ID	E1					
6	**<T>**		IF	-	-				
7	**n1**								
8	**n2**								
9	**n3**								
10	**T: t1**			IF	ID	E1			

12.5.3 nML model

[See file `tmicro/lib/control.n`, rule `j`]

This instruction is modeled by the `j` rule. The image attribute indicates that the jump takes 2 cycles.

```
image  : tg, cycles(2), class(jump);
```

12.6 The `jc` instruction

12.6.1 Format and syntax

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	
0	0	1	0	1	1	1	0	1	0	0	1	0	0	0	0	`jc tg`
tg																

.

12.6.2 Description

The `jc` instruction is a conditional absolute jump. This instruction conditionally transfers control from the current `PC` to a new address given by the `tg` parameter. It is a two word instruction: the first word encodes the instruction, the second word contains the target address. The pipeline diagram for a taken jump is the following.

	cycle	1	2	3	4	5	6	7	8
	issue_sig	1	0	1	1	1			
	tcc	0	0	1	0	0			
	jc	0	0	1	0	0			
	next PC	6	7	10	11	12			
PC	Instruction								
5	**jc T**	IF	ID	E1					
6	**<T>**		IF	-	-	-			
7	**n1**			IF	-	-	-		
8	**n2**								
9	**n3**								
10	**T: t1**				IF	ID	E1		

The pipeline diagram for a not taken jump is the following.

12.6.3 nML model

[See file `tmicro/lib/control.n`, rule `jc`]

	cycle	1	2	3	4	5	6	7	8
	issue_sig	1	0	1	1	1			
	tcc	0	0	0	0	0			
	jc	0	0	1	0	0			
	next PC	6	7	8	9	10			
PC	Instruction								
5	**jc T**	IF	ID	E1					
6	**<T>**		IF	-	-	-			
7	**n1**			IF	ID	E1			
8	**n2**				IF	ID	E1		
9	**n3**					IF	ID	E1	
10	**T: t1**								

This instruction is modeled by the `jc` rule. In the action attribute, the two transitories `tcc` and `jc` are set (see `jcr` instruction). The image attribute indicates that the jump takes 2 cycles.

12.7 The `ji` instruction

12.7.1 Format and syntax

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0	0	1	0	1	1	1	0	1	0	1	0	1	r		

`ji r`

12.7.2 Description

The `ji` instruction is a unconditional indirect jump. This instruction unconditionally transfers control from the current `PC` to a new address specified in the `R[r]` register.

Due to the pipeline, the change of flow is not immediate. As shown in the pipeline diagram, there are two cycles between the jump instruction and the first instruction `t1` at the jump target `T`. This is because the jump is executed in the `E1` stage. In this stage, the target address is read from `R` and written to the `PC`. In the next cycle (cycle 4), the target instruction is fetched. In the intermediate cycles, no instructions are fetched or issued.

	cycle	1	2	3	4	5	6	7	8
	issue_sig	1	0	1					
	next PC	6	10	11					
PC	Instruction								
5	**ji r0**	IF	ID	E1					
6	**n1**		IF	-	-				
7	**n2**								
8	**n3**								
9	**n4**								
10	**T: t1**				IF	ID	E1		

12.7.3 nML model

[See file `tmicro/lib/control.n`, rule `ji`]

This instruction is modeled by the `j` rule. The image attribute indicates that the jump takes 2 cycles.

```
image  : tg, cycles(2), class(jump);
```

12.8 The `cl` instruction

12.8.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	1	1	1	0	1	0	0	1	1	0	0	0	`cl tg`
tg																

12.8.2 Description

The `cl` instruction is a direct call to subroutine. This instruction unconditionally transfers control from the current `PC` to a new address given by the `tg` parameter. It is a two word instruction: the first word encodes the instruction, the second word contains the target address. The `cl` instruction saves `PC` value to the link register `LR` in the `ID` stage.

The pipeline diagram is shown below. The transfer of control takes place in stage `ID`. This means that there is one cycle between the call instruction and the first instruction of the subroutine.

12.8.3 nML model

[See file `tmicro/lib/control.n`, rule `cl`]

This instruction is modeled by the `cl` rule.

12.9 The `clid` instruction

12.9.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	1	1	1	0	1	0	1	0	0	r			`clid r`

12.9.2 Description

The `clid` instruction is a indirect call to subroutine. This instruction unconditionally transfers control from the current `PC` to a new address specified in the `R[r]` register. The new `PC` value is written in the `E1` stage. The `clid` instruction saves `PC` value to the link register `LR` in the `E1` stage.

The pipeline diagram is shown below. The transfer of control takes place in stage `E1`. This means that there are two cycles between the call instruction and the first instruction of the subroutine. These cycles are used as delay slots.

12.9.3 nML model

[See file `tmicro/lib/control.n`, rule `clid`]

This instruction is modeled by the `clid` rule.

12.10 The rt instruction

12.10.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	1	1	1	0	1	0	1	1	1	0	0	0	rt

12.10.2 Description

The rt instruction is a return from subroutine. This instruction unconditionally transfers control from the current PC to a new address given by link register plus one (LR + 1).

The transfer of control takes place in stage E1. This means that there are two intermediate cycles between the return instruction and the instruction at the return address. The instruction that is fetched in cycle 2 is not executed.

12.10.3 nML model

[See file tmicro/lib/control.n, rule rt]

This instruction is modeled by the rt rule. Note that the LR + 1 addition is not modeled in nML. It is part of the behavioral PCU model.

12.11 The rtd instruction

12.11.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	1	1	1	0	1	1	0	0	0	0	0	0	rtd

12.11.2 Description

The rtd instruction is a return from subroutine. This instruction unconditionally transfers control from the current PC to a new address given by link register plus one (LR + 1).

This instruction is similar to the rtinstruction, but has two delay slots. The transfer of control takes place in stage E1. This means that there are two intermediate cycles between the return instruction and the instruction at the return address. During these cycles, the delay slot instructions ds1 and ds2 are executed.

12.11.3 nML model

[See file tmicro/lib/control.n, rule rtd]

This instruction is modeled by the rtd rule. Note that the LR + 1 addition is not modeled in nML. It is part of the behavioral PCU model.

12.12 The `nop` instruction

12.12.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0

```
nop
```

12.12.2 Description

The effect of a `nop` is that for 1 cycle, no actions are issued.

12.12.3 nML model

[See file `tmicro/lib/control.n`, rule `nop`]

This instruction is modeled by the `nop` rule.

12.13 The `do` and `doi` instructions

12.13.1 Format and syntax

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0	1	1	0	1	1	g		
last															

```
do gg,last
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0	1	1	1	0	0	0	0	0
last															
count															

```
doi count,last
```

12.13.2 Description

The TMICRO core supports zero overhead hardware controlled loops. This allows fast looping over a block of instructions. Once the loop is set up with the `do` instruction, there are no additional execution cycles needed to control the looping. The loop is executed a pre-specified number of iterations and is controlled by a dedicated hardware block. The TMICRO core supports three levels of hardware loops. Typically, two levels are used in general code, and the third level is reserved for interrupt service routines.

The status of the hardware loops is stored in a set of dedicate register.

`LS` The *loop start address* registers store the address of the first loop instruction. `LS` is a register file with three fields, one for each loop level.

`LE` The *loop end address* registers store the address of the last loop instruction. `LE` is a register file with three fields, one for each loop level.

`LC` The *loop count* registers store the remaining number of iterations of the loop. `LC` is a register file with three fields, one for each loop level.

LF The *loop flags* register keeps track of the number of hardware loops that are active and addresses the LS, LE and LC registers.

The LF register has an initial value of 3. This value indicates that there are no hardware loops active. When there LF register contains 0, this indicates that there is one active hardware loop, with loop parameters in LS[0], LE[0] and LC[0]. When the LF register contains 1, there are two active hardware loops; an outer loop with parameters in LS[0], LE[0] and LC[0]; and an inner loop with parameters in LS[1], LE[1] and LC[1]. An LF value of 2 indicates that there are three active hardware loops.

The do instruction initiates a zero overhead loop. The r parameter specifies the loop count register. The second instruction word specifies the loop end address. There is one delay slot.

The doi instruction also initiates a zero overhead loop. The second instruction word specifies the loop end address, and the third instruction word specifies the loop count.

The pipeline diagrams below show the startup phase of a hardware loop.

The following actions take place during loop startup.

In stage E1

- LF register is incremented. When no other hardware loops are active, LF wraps from a value of 3 to 0. When other loops are active, LF increments from 0 to 1 or from 1 to 2.
- The loop end address is fetched from PM and is stored in LC.
- The current PC value plus 1 is written to LS.
- The loop count is fetched from PM of from R and is stored in LC.

12.13.3 nML model

[See file tmicro/lib/control.n, rule do]

[See file tmicro/lib/control.n, rule doi]

These instructions are modeled by the do and doi rules. Note that the functional unit dlflg is used to increment the LF register and that the functional unit dlls to compute the loop start address.

12.14 The dlf instruction

12.14.1 Format and syntax

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	
0	0	1	0	1	1	1	0	1	1	1	0	1	0	0	0	dlf

12.14.2 Description

The dlf instruction decrements the content of the loop flag register LF. In case LF contains 0, the new value becomes 3. The purpose of this instruction is to decrement the loop flag register after a jump out of a zero overhead loop. Note that such a jump can be programmed in assembly code but will not be generated when compiling C code.

12.14.3 nML model

[See file `tmicro/lib/control.n`, rule `dlf`]

This instruction is modeled by the `dlf` rule.

12.15 Behavioral model of the PCU

[See file `tmicro/lib/tmicro_pcu.p`]

The behavior of control instructions is for a large part modeled in PCU. More information on controller modeling can be found in [7].

The behavioral model of the PCU consists of a number of user defined functions. The main functions are:

```
void tmicro::user_issue();
void tmicro::user_next_pc();
```

The `user_issue()` function captures the instruction that was fetched from `PM` and determines if this must be issued. The `user_next_pc()` function is responsible for computing the next `PC` value and initiating `PM` fetches. These functions are executed every clock cycle. Furthermore, an auxiliary function for checking the interrupt request inputs and for determining if a pending interrupt can be accepted is defined (`check_interrupts()`).

12.15.1 PCU storages

The following storages are declared and used locally in the PCU:

```
pcu_storages {
    reg reg_booting<bool>;     // is set during first cycle after reset
    reg reg_halted<bool>;      // is set when processor is in halted state
    reg reg_fetch<bool>;       // delayed  fetch signal
    reg reg_f_instr<iword>;    // delayed fetched instruction
    trn trn_interrupted<bool>; // interrupt is accepted in current cycle
    trn debug_mode<bool>;      // set when debug request is accepted
    reg reg_debug_mode<bool>;  // delayed debug_mode signal
}
```

12.15.2 Fetching

Whether an instruction is fetched in a particular cycle depends on the previous instruction. The default behavior is that an instruction is fetched every cycle. This is indicated by setting the signal `fetch` to 1 at the beginning of `user_next_pc()`. The cases where no instruction must be fetched are

- Cycles during which the processor accepts an interrupt (see [6]).
- When the processor is halted.
- When the processor is in debug mode (see [8]).
- When a multi cycle or multi word instruction is being executed.

Under these conditions, `fetch` is set to 0. When `fetch` is set, a new instruction is fetched from `PM`. This instruction fetch is defined in the `user_next_pc()` function.

```
if (fetch)
    pm_read‘1‘ = PM[pm_addr = nextpc]‘0‘;
```

12.15.3 Issuing

Depending on the `fetch` value of the previous cycle, either a new instruction is obtained from the program memory, or the instruction that was stored in `ref_f_instr` is selected for issuing.

```
iword f_instr = reg_f_instr;
if (reg_fetch)
    f_instr = pm_read;
```

The default issuing behavior is that the `f_instr` instruction is inserted in the instruction register. The following code models the issuing and is part of the `user_issue()` function (the case that supports interrupts will be explained in [6]).

```
if (trn_interrupted) {
    iword swi_instr = pdg_encoding_swi(i_vector);
    issue_instr(0,0,swi_instr)
}
else if (mword_sig())
    add_mword(f_instr);
else if (issue_ins && !stop_issue)
    issue_instr(pcr = PC,1,f_instr);
```

In the case of a multi-word instruction, the fetched instruction word must not be issued as a new instruction, but must be added as an extension to the previous instruction.

Issuing is done whenever the `issue_sig()` signal is set (see [7]). As explained in section 12.4.3, issuing can be overruled by the `jcr` and `jc` signals. In the PCU model, a variable called `issue_ins` signals is computed as follows.

```
bool jcr_taken = jcr && tcc;
bool jc_taken  = jc  && tcc;

// force issue_ins low when jcr or jc  is taken
bool issue_ins =  issue_sig() && !jcr_taken && !jc_taken;
```

During the booting cycle there is not yet an instruction that has been fetched, therefore none can be issued. This is modeled by means of the `stop_issue` variable. Note that other conditions contribute to `stop_issue`. These are explained elsewhere (see [6] and [8]).

```
bool stop_issue = reg_booting || ...;
```

12.15.4 Killing

In case a `jcr` instruction is taken, the next instruction must be killed. This is done by calling the `kill_instr()` function (see [7]) when the `jcr` jump is taken.

```
if (jcr_taken)
     kill_instr();
```

When an instruction was fetched but that instruction is not issued, it is saved on the `reg_f_instr` register. When that instruction is then issued in a subsequent cycle, it does not need to be fetched again, but is read from `reg_f_instr`. This mechanism applies to multi cycle instructions such as the `PM` load and store instructions. Even though `reg_f_instr` is normally not used, the saving of `f_instr` is done unconditionally.

```
reg_f_instr = f_instr;
```

12.15.5 Next PC computation

The `user_next_pc()` function computes a new `PC` value based on the current `PC` value, based on possible control flow instructions that are present in the pipeline, and based on information from the end of loop detection.

```
addr nextpc = pcr = PC;

if (trn_interrupted || reg_halted || debug_mode) {
    fetch = 0;
}
else if (reg_booting || reg_debug_mode)  { // start after reset or debug
    fetch = 1;
    reg_booting = 0;
}
else if (jump_offs_sig)
    nextpc = pcr + offs;          // relative jump
else if (jump_trgt_sig)
    nextpc = trgt;                // absolute jump
else if (issue_sig() || mword_sig())  {
    lp_update = 1;   // no LC or LF update for previous cases
    if (lp_jump)
        nextpc = tlsr;
    else
        nextpc = pcr + 1;
}
else
    fetch = 0;
```

12.15.6 End of loop test

The end of loop test is modeled in `user_next_pc()`. As much operations as possible are programmed early on in the `user_next_pc()` function, so that the resulting logic can operate concurrently to the next PC computation. These operations for the reading of the loop control registers are conditional on a valid index being present in LF.

```
word tlcr = 0;
word tlsr = 0;
word tler = 0;
lfra = LF;
if (lfra < 3) {
    tlcr = lcr = LC[lfra];
    tlsr = lsr = LS[lfra];
    tler = ler = LE[lfra];
    if (pcr == (uint16_t)tler) {
        if(lcr == 1)
            lp_done = 1;
        else
            lp_jump = 1;
    }
}
```

The end of loop test is active when the value in LF is less than 3 and when the PC (`pcr`) equals the loop end address of the active loop. When the count of the active loop reaches 1, the loop is terminated, `lp_done` is set to one. Otherwise, `lp_jump` is set to indicate that a jump to the loop start address is needed.

The actual jump back is implemented in the next PC computation (see section 12.15.5).

```
if (lp_jump)
    nextpc = tlsr;
```

Note that this assignment is at the end of the if-then-else statement where the updating of PC based on control flow instructions is checked first. This allows that a control flow instruction that is scheduled near the end of the loop takes precedence over the end of loop test (see also section 13.2.3).

The loop control registers must be updated only when there was no control flow instruction near the end of the loop that took precedence, when there was no interrupt, and the normal fetch state is active. The

lp_update control signal is then set, and the following actions are executed.

- When the loop has terminated, the loop flag register is decremented.

```
if (lp_done)
    LF = lfw = lfra - 1;
```

- When the loop has not terminated, the loop count is decremented.

```
if (lp_jump)
    LC[lfra] = lcw = lcr - 1;
```

The end of loop test code supports the testing of a single loop. It is therefore not allowed to have two nested hardware loops with the same loop end address (see section 13.2.3).

12.15.7 Booting

In the first cycle after the reset signal is de-asserted, it is not possible to issue an instruction because none has been fetched. Initially (after a reset) the booting cycle, the reg_booting register is set.

```
hw_init reg_booting = 1;
```

During this booting cycle, the PCU must only drive the PC value onto the PM address bus and assert PM read control signal (fetch must be set to 1). Furthermore, reg_booting can be cleared.

```
else if (reg_booting || reg_debug_mode)  { // start after reset or debug
    fetch = 1;
    reg_booting = 0;
}
```

	cycle	1	2	3	4	5	6	7	8
	issue_sig	1	0	1	1	1			
	LR	?	?	7	7				
	next PC	6	10	11					
PC	Instruction								
5	`cl T`	IF	ID	E1					
6	`<T>`		IF	-	-				
7	`n1`								
8	`n2`								
9	`n3`								
10	`T: t1`			IF	ID	E1			

	cycle	1	2	3	4	5	6	7	8
	issue_sig	1	0	1	1	1			
	diid	0	1	0	0	0			
	die1	0	0	1	0	0			
	next PC	6	7	10	11				
PC	Instruction								
5	**clid r0**	IF	ID	E1					
6	**ds1**		IF	ID	E1				
7	**ds2**			IF	ID	E1			
8	**x1**								
9	**X2**								
10	**T: t1**				IF	ID	E1		

	cycle	1	2	3	4	5	6	7	8
	issue_sig	1	0	0	1	1			
	LR	10	10	10	10				
	next PC	6	6	10	11				
PC	Instruction								
5	`rt`	IF	ID	E1					
6	`n1`		IF	-	-	-			
7	`n2`								
8	`n3`								
9	`n4`								
10	`T: t1`				IF	ID	E1		

	cycle	1	2	3	4	5	6	7	8
	issue_sig	1	1	1	1	1			
	LR	10	10	10	10				
	next PC	6	7	10	11				
PC	Instruction								
5	**rtd**	IF	ID	E1					
6	**ds1**		IF	ID	E1				
7	**ds2**			IF	ID	E1			
8	**n2**								
9	**n3**								
10	**T: t1**				IF	ID	E1		

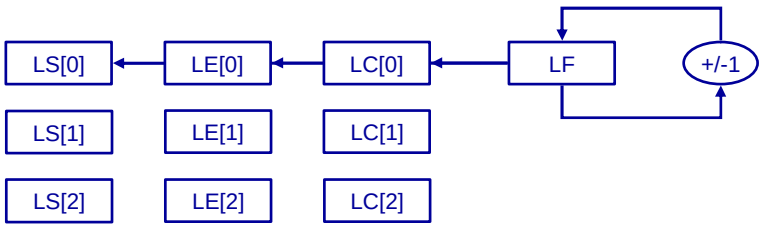


Figure 12.2: The hardware loop registers.

		cycle	1	2	3	4	5	6	7	8
		issue_sig	1	0	1	1	1			
		die1	0	0	1	0	0			
		LF	3	3	3	0	0			
		LS	?	?	?	8	8			
		LE	?	?	?	10	10			
		LC	?	?	?	cnt	cnt	cnt	cnt-1	cnt-1
		next PC	6	7	8	9	10	8	9	10
PC	Instruction									
5	`do r0,L`		IF	ID	E1					
6	`<L>`			IF	-	-				
7	`ds1`				IF	ID	E1			
8	`lp1`					IF	ID	E1		
9	`lp2`						IF	ID	E1	
10	`L: lp3`							IF	ID	E1

		cycle	1	2	3	4	5	6	7	8
		issue_sig	1	0	0	1	1			
		LF	3	3	3	0	0			
		LS	?	?	?	8	8			
		LE	?	?	?	10	10			
		LC	?	?	?	cnt	cnt	cnt	cnt-1	cnt-1
		next PC	6	7	8	9	10	8	9	10
PC	Instruction									
5	`doi cnt,L`		IF	ID	E1					
6	`<L>`			IF	-	-				
7	`<cnt>`				IF	-	-			
8	`lp1`					IF	ID	E1		
9	`lp2`						IF	ID	E1	
10	`L: lp3`							IF	ID	E1

13

Hazards

Hazards are artifacts of the pipelined execution of instructions. Following hazards are commonly identified in literature [9]:

- Structural hazards: occur when a hardware resource (a transitory in the nML model) is used in different pipeline stages by different instructions.
- Data hazards: occur when a register is accessed in different stages by different instructions. A typical example is the read-after-write (RAW) hazard that occurs when a register is written in a late stage and that register is also read (by another instruction) in an early stage.
- Control hazards: occur when a jump instruction is issued that results in a change of the program counter in a later pipeline stage.

On TMICRO, all hazards are solved by organizing the software in such a way that the hazard does not occur. This approach is called software stalling, and is indicated in nML by means of `sw_stall` rules.

The complete list of structural and data hazards can be generated automatically by running the nML analysis tool ANIMAL with the `-h` option. You can run ANIMAL on the command line, as follows:

```
cd tmicro/lib
animal -h -d3 -Iisg +wisg -D__checkers__ tmicro
```

Alternatively, you can specify following setting in the CHESSDE. Open `tmicro.prx` in CHESSDE. Open the *Project settings* window, select *Processor* in the left pane and *nML front end* in the center pane, and specify following setting :

- `Add SW stalls & print hazards (-h)` : *On*.

and optionally

- `Extra arguments` : *-d 3*.

This will produce a file called `isg/tmicro_iss_hzd.rpt`. Note that the hazard report is generated for the Simulation view. This view contains additional register transfers compared to the Compiler view, and these are best also included in the hazard report.

For each hazard, this hazard report file will contain a description of the hazard and a reference to the nML source lines that caused the hazard. This description is in comments and is followed by a software stall rule for the hazard. The hazard report file will contain more context information when a higher debug level such as `-d7` or `-d9` is selected.

The `isg/tmicro_iss_hzd.rpt` lists the following hazards:

```
Structural hazard for cycle offset [1,1] on dm_addr
Structural hazard for cycle offset [1,1] on offs
RAW data hazard for cycle offset [1,1] on R
WAW data hazard for cycle offset [1,1] on R
WAW data hazard for cycle offset [1,1] on LR
RAW data hazard for cycle offset [1,1] on SP
```

```
WAW data hazard for cycle offset [1,1] on SP
RAW data hazard for cycle offset [1,1] on ILR
WAW data hazard for cycle offset [1,1] on ILR
```

As all hazards will be solved by software stalling, the `tmicro_iss_hzd.rpt` file can be added directly to the TMICRO nML model. It can be renamed to `hazards.n`

```
mv tmicro\_hzd.rpt hazard.n
```

and included in the `tmicro.n` file.

13.1 Structural hazards

A structural hazard occurs when a transitory is used in more than one stage. The following structural hazards are present on the TMICRO core.

- `Structural hazard for cycle offset [1,1] on dm_addr`: The address bus `dm_addr` is used by load instructions in the `ID` stage; and is used by store instructions in the `E1` stage. The `1,1` in the description line specifies the range of cycle offsets for which the hazard occurs.
- `Structural hazard for cycle offset [1,1] on offs`: The jump offset transitory `offs` is used by the `jrd` and `jr` instructions in the `ID` stage; and is used by the `jcr` instruction in the `E1` stage.
- A hazard also occurs when a program memory store is programmed as the first delay slot instruction of a control flow instruction with two delay slots. The `PM` load drives the `pm_addr` bus in the `ID`. At the same time, the control flow instruction is in the `EX` stage, where it assigns the next fetch address to `pm_addr`.

13.2 Data hazards

Two categories of data hazards are present on the TMICRO core: read-after-write hazards between `ID` and `E1`, and write-after-write hazards between `ID` and `E1`.

13.2.1 Read-after-write hazards between ID and E1

This hazard occurs when a register is written by one instruction in a late stage (stage `E1`), and is read by another instruction in an early stage (stage `ID`). Figure 13.1(a) shows an `add` instruction, followed by a `ld` instruction. The `add` writes to register `R[0]` in the same cycle the `ld` reads `R[0]`. In case there is a data dependency, this hazard must be resolved by inserting at least one intermediate instruction (see figure 13.1(b)).

The following read-after-write hazards between `ID` and `E1` are present on the TMICRO core.

```
RAW hazard for cycle offset [1,1] on R
```

```
RAW hazard for cycle offset [1,1] on SP
```

```
RAW hazard for cycle offset [1,1] on ILR
```

All hazards of this kind are resolved in software. This is expressed by adding `sw_stall` rules to the nML model (see file `hazard.n`). The following example treats the `R` register file. It states that there is a hazard condition when one instruction writes to a certain field of `R` in stage 2 (the `E1` stage), and the next instruction reads from the same file in stage 1 (the `ID` stage).

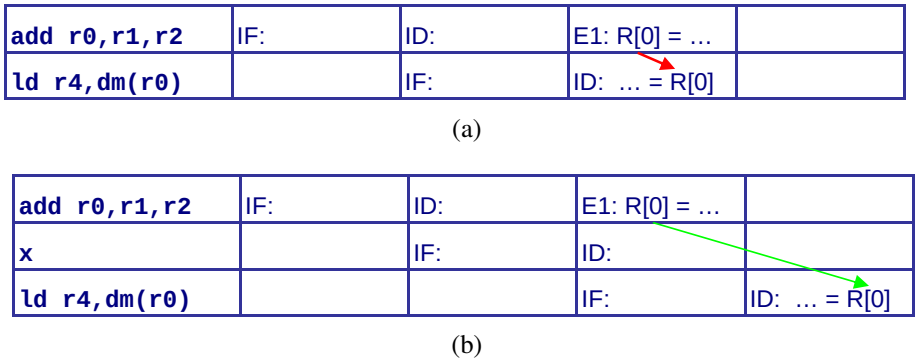


Figure 13.1: Example of a read-after-write hazard (a), and a reordered program without the hazard (b).

Figure 13.2: Example of a write-after-write hazard.

```
sw_stall 1 cycles () class(read_after_write_R) {
    stage E1: R[#] = ...;
}
-> {
    stage ID: ... = R[#];
}
```

Note that this stall rule was given a class name `read_after_write_R`, which is used by the tools to refer to this stall rule.

13.2.2 Write-after-write hazards between ID and E1

This hazard occurs when a register is written by one instruction in a late stage (stage `E1`), and is also written by another instruction in an early stage (stage `ID`). Figure 13.2(a) shows a `mv` instruction, followed by a `cl` instruction. The `mv` writes to register `LR` in the same cycle as the `cl`. By default, the tools avoid this hazard by preferring the update of the second instruction over that of the first instruction.

13.2.3 Data hazards caused by the end of loop test

The end of loop test, which takes place for the last instruction of a hardware loop, will update the `PC` when the loop has not completed. This update may conflict with the `PC` update that is part of a jump instruction. To avoid the hazard, the jump must not be scheduled near the end of the loop.

Figure 13.3(a) shows a program where a jump `jr` is at address 9 and the last instruction in the loop is at address 11. This is a safe situation as the jump updated the `PC` in cycles 11, whereas the end of loop test would update the `PC` in cycle 12.

Figure 13.3(b) shows a program where the jump is at address 10 and the last instruction in the loop is at address 11. This situation is still safe because the PCU is implemented in such a way that the jump takes precedence over the loop test.

Figure 13.3(c) shows a program where the jump is at the same address as the last instruction in the loop. This situation is unsafe because the jump would take place after the loop back to the loop start address. To

avoid this conflict, the jump must be positioned at least one instruction before the loop end address. This is achieved as follows:

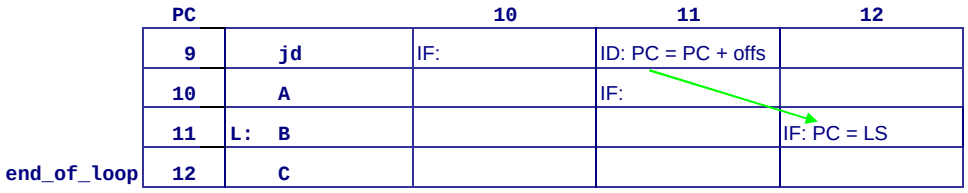
- First, each jump instruction is tagged with the `class(jump)` instruction property. For example, for the `jr` instruction:

```
opn jr(of: c8s)
{
    ...
    image  : of, cycles(2), chess_pc_offset(1), class(jump);
}
```

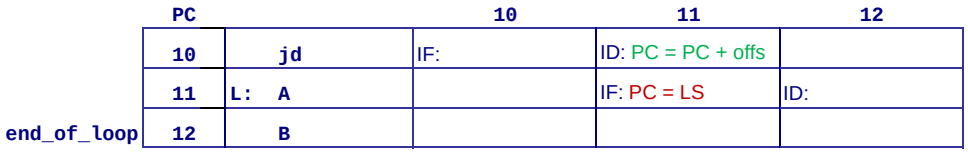
- Then, a software stall rule is defined.

```
sw_stall 1 words () class(jump_to_loop_end) {
    class(jump);
} -> {
    special(end_of_loop);
}
```

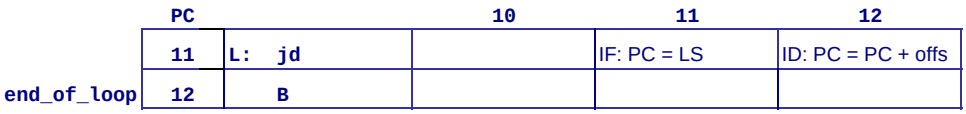
This rule specifies that there is a hazard when a jump instruction is located in the last instruction word of a do-loop. Notice that `special(end_of_loop)` denotes the first instruction after the loop.



(a)



(b)



(c)

Figure 13.3: Example of a hazard caused by the end of loop tests.

Another problem arises when a `clid` instruction is executed near the end of the loop. When the delay slots are executed, the PC is incremented. Suppose it reaches the loop end address in the cycle where the call is executed. Then, the call takes precedence and the end of loop test is not executed. The called function then returns to PC + 1, which is after the loop. This hazard is avoided through the following stall rule.

```
sw_stall 1..3 words () class(bsr_to_loop_end)  {
    stage E1: lnk_pf = bsr(trgt); delay_slots(2);
} -> {
    special(end_of_loop);
}
```

In order to avoid that two nested hardware loops terminate at the same address (see section 12.13.2), following stall rule is defined.

```
sw_stall 0 words () class(between_loop_ends) {
    special(end_of_loop);
} -> {
    special(end_of_loop);
}
```

13.3 Control hazards

Control hazards are identified in nML by means of the `cycles` and `delay_slots` image attributes. This topic is discussed in detail in chapter 12.

A

Some useful commands

Commands to build the processor model

The following commands are executed in the `lib` directory.

1. To build the processor model.

```
chessmk tmicro.prx
```

2. To build the processor specific software library (this library is required to build the application program).

```
chessmk libtmicro.prx
```

3. To build the C runtime library (standard C functions and hosted IO functions).

```
cd runtime
chessmk libc.prx
```

Commands to build the instruction set simulator

Execute the following command in the `iss` directory.

1. To build the cycle accurate ISS.

```
chessmk tmicro_ca.prx
```

2. To build the instruction accurate ISS.

```
chessmk tmicro_ia.prx
```

Commands to generate the RTL model

The following commands are executed in the `hdl` directory.

1. To generate the VHDL model.

```
chessmk tmicro_vhdl.prx
```

2. To generate the Verilog model.

```
chessmk tmicro_vlog.prx
```

Notice that in both cases, the GO configuration file `go_options.cfg` is used.

In case the `on_chip_debugging` option is specified, a shared object with the JTAG emulator code must be created before the RTL simulator can be started. This is done as follows

1. For VHDL simulation.

```
cd jtag_emulator_vhdl
Edit the Makefile and adapt the MTI_HOME path.
make
```

2. To make JTAG emulator for Verilog simulation.

```
cd jtag_emulator_vlog
make
```

Example software project

You can find an example of a software project in the `sort` directory. Open the `sort.prx` file in CHESSDE and build and simulate the project.

Bibliography

[1] *Chess Compiler User manual*. Synopsys, September 2016. Version L-2016.09.

[2] *The nML Processor Description Language*. Synopsys, September 2016. Version L-2016.09.

[3] *Checkers ISS Interface manual*. Synopsys, September 2016. Version L-2016.09.

[4] *Go User manual, nML to HDL translation*. Synopsys, September 2016. Version L-2016.09.

[5] *Chess Compiler Processor Modeling manual*. Synopsys, September 2016. Version L-2016.09.

[6] *Implementing Interrupts on the Tmicro Core*. Synopsys, March 2015. Version J-2015.03.

[7] *Primitives Definition and Generation manual*. Synopsys, September 2016. Version L-2016.09.

[8] *Implementing On Chip Debugging on the Tmicro Core*. Synopsys, March 2015. Version J-2015.03.

[9] *Computer Architecture: A Quantitative Approach, 2nd Edition*. John L. Hennessy and David A. Patterson. Morgan Kaufmann, 1996.