Politecnico di Torino

ARM9TDMI Technical Report

Microelectronic Systems

A.A. 2015 - 2016

01NOYOQ

Canessa	Emanuele	S228234
Ciraci	Alessandro	S231552

Contents

1 Introduction to the ARM9TDMI Processor

1.1 About ARM9TDMI

The ARM9TDMI is an embedded general purpose processor; it belongs to the ARM9 processor family, based on the ARMv4T RISC Instruction Set Architecture.
Released in 1998, it supports both 32-bit ARM and 16-bit Thumb instruction sets, that allow a better trade-off between performance and code density.
The processor supports both bidirectional and unidirectional connections to external memory systems; it also includes hardware support for external coprocessors.

The ARM9TDMI is a five-stage pipelined processor consisting of:

- (F) Instruction Fetch
- (D) Instruction Decode / Register Read
- (E) Execute
- (M) Data Memory Access
- (W) Register Write

The memory scheme is based on the Harvard architecture, thus having separated memories (and interfaces) for instructions and data. These memory interfaces can be either connected to a cache or directly to an SRAM based system.

1.2 Processor Block Diagram

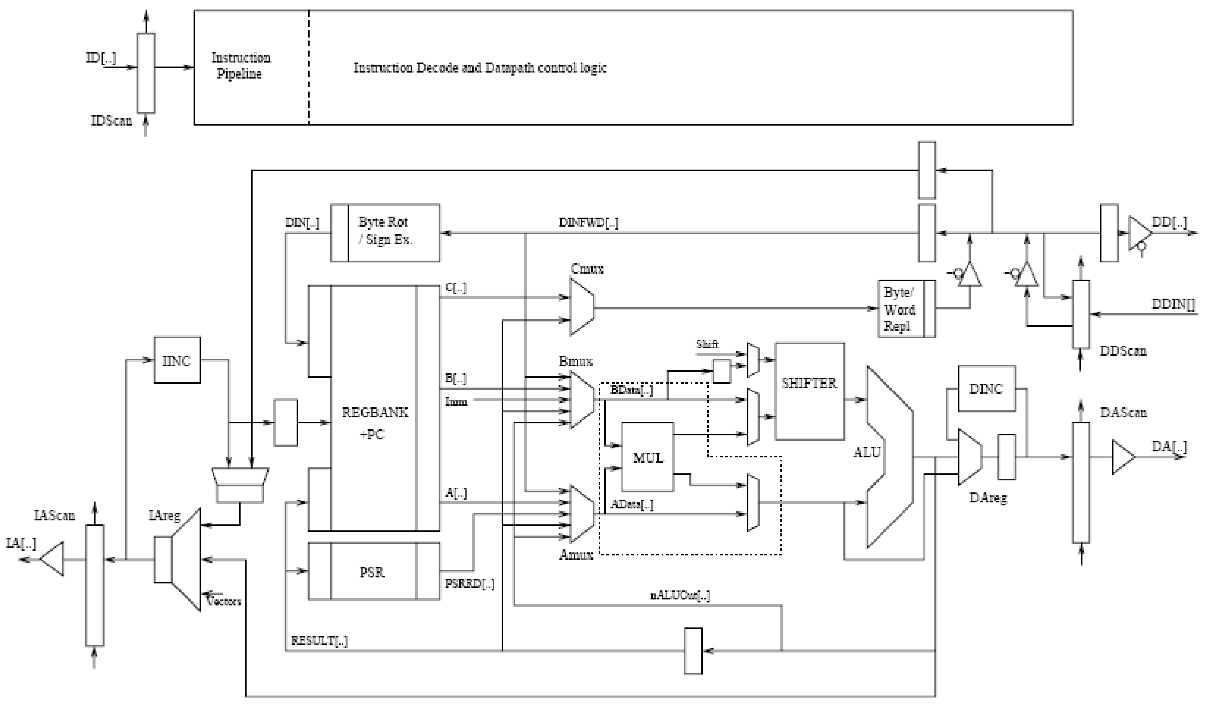

Figure 1: Block diagram of the ARM9TDMI processor

2 Instruction Fetch Stage

2.1 About the Instruction Memory Interface

The ARM9TDMI implements an Harvard bus architecture with separate instruction and data interfaces; it can also operate either in big-endian and little-endian memory configurations. All the handshaking signals, that define the type of access to the memory, are generated and read from `GCLK`.

2.2 Wait States

For memory accesses which requires more than the canonic one cycle, the core can be put in a wait state using the signal `nWAIT`. This signal is inverted and ORed with `GCLK` in order to stretch the internal processor clock and allowing a slow memory to take some time to answer a request. The `nWAIT` signal can only change when `GCLK` is HIGH. For debug purpose, the internal core clock is exported to the output `ECLK`.

2.3 Instruction Memory Protocol

Each time a new instruction enters the Execute (E) stage of the pipeline, a new opcode is fetched from the instruction memory. Ideally, each fetch should last one cycle; however, it is possible to connect memories that answer differently in case of sequential or non-sequential accesses: for instance, a memory could take one cycle to provide data in case of a sequential access, while it could take more than one cycle in case of a non-sequential access.

All the instruction memory accesses are generated starting from the signals `GCLK`, `InMREQ` and `ISEQ`. An instruction fetch transaction starts by driving the `InMREQ` signal LOW. The memory reads the address on `IA[31:1]` and the signal `ISEQ` will indicate whether the access is sequential or not.

Depending on the combination of `InMREQ` and `ISEQ` the cycle type can be derived:

Table 1: `InMREQ` and `ISEQ` encoding

`InMREQ`	`ISEQ`	**Cycle Type**
`0`	`0`	Non-sequential
`0`	`1`	Sequential
`1`	`0`	Internal
`1`	`1`	*Reserved*

The ARM9TDMI performs 32-bit or 16-bit instruction fetches depending on whether the processor is in ARM or Thumb state. The processor state is determined by the value read on `ITBIT`.
Instruction fetches can be also marked as aborted by driving the signal `IABORT`.

2.3.1 Instruction Fetch Timing

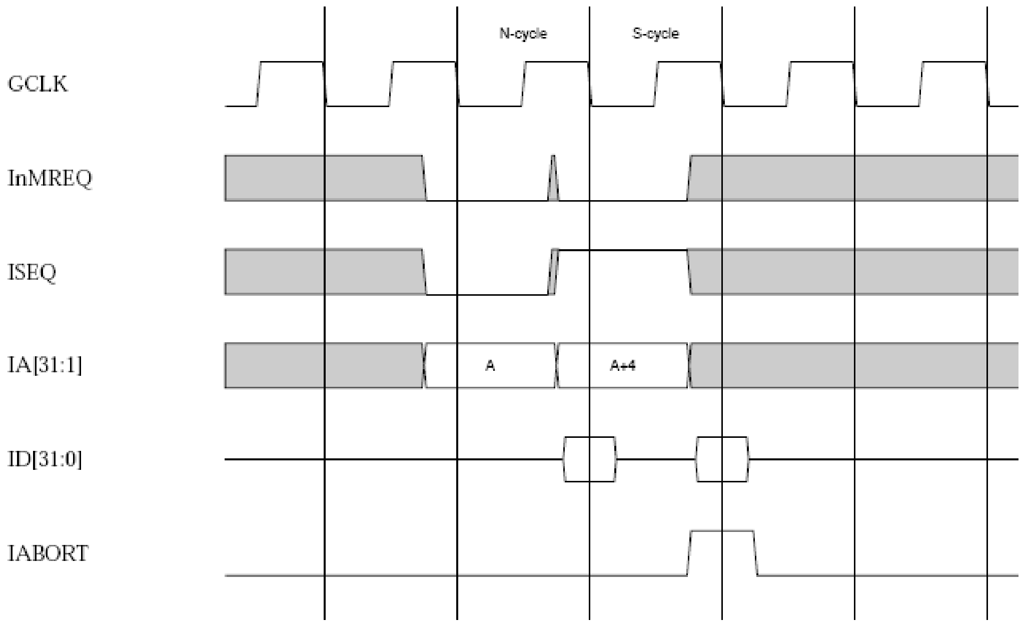

Figure 2: Instruction fetch timing diagram

2.4 Endian effects for Instruction Fetches

The ARM9TDMI supports both big-endian an little-endian memory fetches, resorting to an input signal `BIGEND`. This signal must not change at runtime. Depending on the configuration of `ITBIT` and `BIGEND`, the instruction fetch can change. When the processor is in ARM State (`ITBIT` LOW), the endian configuration does not affect the instruction fetches. When, instead, `ITBIT` is HIGH, the instructions read on `ID` are in different order, depending on the endian configuration.

Table 2: Endian effect on instruction position

`IA[1]`	**Little Endian BIGEND = 0**	**Big Endian BIGEND = 1**
0	`ID[15:0]`	`ID[31:16]`
1	`ID[31:16]`	`ID[15:0]`

3 Instruction Decode / Register Read Stage

The Instruction Decode unit has two main roles in the processor:

- Decode the instructions (operation actually performed by the Control Unit);
- Read the data from the Register File.

The register read process is performed during the *negative* clock edge, to avoid Read-after-Write (RAW) data hazards. More information on the Register File in chapter 6.

4 Execute Stage

The Execute Stage is composed by three main components, the ALU, the Barrel Shifter and the High-Speed Multiplier.

4.1 ALU

The implementation of the ALU has been derived from the logic unit of the Sun Microsystems' UltraSPARC T2 'Niagara 2' and the adder from Intel's Pentium 4 'Prescott'.
The ARM9TDMI ALU supports the following operations:

- Addition
- Subtraction
- Logic-OR
- Logic-AND
- Logic-XOR
- Logic-NOT
- Bit Clear
- Register Swap

To allow these operations to be performed, the following control signals are needed:

- `LOGIC_OP`, which selects the operation to be performed by ALU, between arithmetical (if all zeros) and logical;
- `CARRY_OP`, which selects the value of the carry input for the arithmetic unit;
- `A_INVERT`, which logically negates the value on bus A;
- `B_INVERT`, which logically negates the value on bus B;
- `A_PASS`, which connects directly the output of the ALU to bus A;
- `B_PASS`, which connects directly the output of the ALU to bus B.

4.2 Barrel Shifter

The Barrel Shifter is a special kind of shifter able to shift a number by a variable amount of positions in only a clock cycle. The ARM9TDMI's Barrel Shifter is used to both pre-process data and to decode immediates.

An interesting feature of the ARM processors is the ability to perform a shift operation in the same clock cycle of an arithmetical or logical operation, thus allowing a much higher performance.

ARM's Barrel Shifter allows to perform several operations:

- (LSL) Logical Shift Left
- (LSR) Logical Shift Right
- (ASR) Arithmetic Shift Right
- (ROR) Rotate Right
- (RRX) Rotate Right with Extend

The shift amounts for the aforementioned operations can be provided using either an immediate, that has a particular encoding, or a register. The ARM9TDMI supports shift amounts greater than 32, providing always the correct result.

The implementation of these operations is naive and relies on functions that implements the behavior described in the ARM9TDMI ISA Reference Manual. All the functions are defined into a VHDL Package called `SHIFT_FUNCTION_pack.vhd`.

4.2.1 Immediate Encoding

The 12 bits reserved for immediates do not store directly the 12-bit number to be processed. This field ([11:0]) is splitted in two parts: the first field ([11:8]) is called `rotate_imm`, while the second field ([7:0]) stores `immed_8`. The immediate is then formed by rotating right the `immed_8` by two times the unsigned amount specified in `rotate_imm`.
This means that:

- Not all 32-bit immediates are legitimate.
- Some values of `immediate` have more than one possible encoding.

4.3 High-Speed Multiplier

The High-Speed Multiplier, implemented in several ARM processors, is based on the Booth's multiplication algorithm; it supports both 32-bit and 64-bit signed/unsigned multiplication result. The multiplier can also perform multiply and accumulate (MAC) operations.

This multiplier, as can be seen in chapter 4.3.2, uses a carry-save array with 4 layers of adders (with a data width of 34 bit), each handling two bit multiplication, so the array can multiply eight bit per clock cycle.The array is cycled up to four times, using early termination (signal `EARLY_FINISH`) to complete the instruction in fewer clock cycles if the multiplier Rs has a sufficient number of zeros in its top bit.
For further informations on the clock cycle count of the multiplication operations, see chapter 4.3.1.

The partial sum and carry are stored in two 64-bit registers; the two registers are cleared at the start of the instruction, but the sum register can be initialized to an initial value in case of a MAC operation.
As the multiplier is shifted 8 bit right every clock cycle in the `Rs` register, the intermediate result registers are rotated right by 8 bit each cycle; this is used to allow a much regular structure among the carry-save adders and the removal of any need for the factor `Rm` to be shifted.

To complete the multiplication, at the end of the cycles on the carry-save array, the partial results are summed into the ALU to produce the final result; since the ALU can only perform 32 bit operations, in case of a 64-bit multiplication, a further ADC operation is performed to evaluate the upper 32 bit of the result.

The algorithm of the High-Speed Multiplier is a close derivation of Booth's algorithm. For each clock cycle, the multiplier takes 8 bit from the factor Rs and divides them in 4 groups of 2 bit, used to choose among 4 possible values to be summed to the partial results:

- If $Rs_{2bit} =$ `00` then adds `0`
- If $Rs_{2bit} =$ `01` then adds Rm
- If $Rs_{2bit} =$ `10` then adds $2 \times Rm$ if unsigned, $-Rm$ if signed

- If Rs_{2bit} = `11` then adds $3 \times Rm$ if unsigned, $-2 \times Rm$ if signed

The choice of a 34-bit datapath throughout the multiplier was forced by the necessity of representing any value on 32 bit multiplied by 3, thus forcing the use of two extra bits.

4.3.1 Multiplier Cycle Count

The number of cycles that a multiply instruction takes to complete depends on which instruction it is, and on the value of the multiplier-operand. The multiplier-operand is the contents of the register specified by bits [8:11] of the ARM multiply instructions, or bits [2:0] of the Thumb multiply instructions.

- For ARM `MUL`, `MLA`, `SMULL`, `SMLAL`, and Thumb `MUL`, it takes:

  1 if bits `[31:8]` of the multiplier operand are all zero or one
  2 if bits `[31:16]` of the multiplier operand are all zero or one
  3 if bits `[31:24]` of the multiplier operand are all zero or one
  4 otherwise

- For ARM `UMULL` and `UMLAL`, it takes is:

  1 if bits `[31:8]` of the multiplier operand are all zero
  2 if bits `[31:16]` of the multiplier operand are all zero
  3 if bits `[31:24]` of the multiplier operand are all zero
  4 otherwise

4.3.2 Block diagram of the High-Speed Multiplier

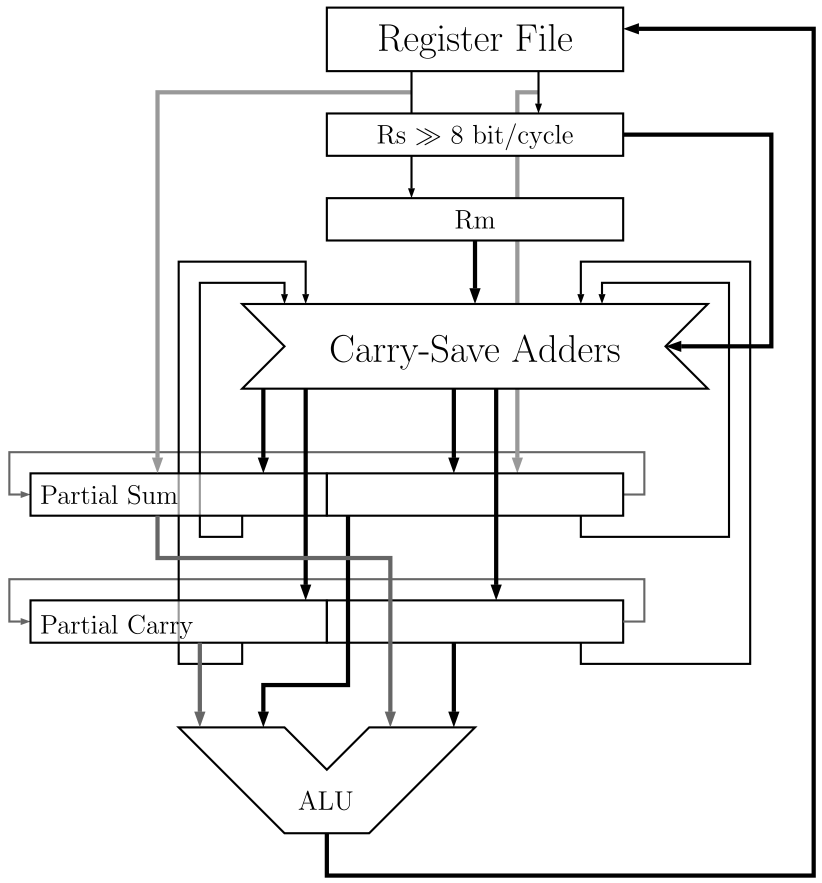

Figure 3: High-Speed Multiplier block diagram

5 Data Memory Access Stage

5.1 About the Data Memory Interface

The ARM9TDMI implements an Harvard bus architecture with separate instruction and data interfaces; it can also operate either in big-endian and little-endian memory configurations. All the handshaking signals, that define the type of access to the memory, are generated and read from `GCLK`.

5.2 Wait States

For memory accesses which requires more than the canonic one cycle, the core can be put in a wait state using the signal `nWAIT`. This signal is inverted and ORed with `GCLK` in order to stretch the internal processor clock and allowing a slow memory to take some time to answer a request. The `nWAIT` signal can only change when `GCLK` is HIGH. For debug purpose, the internal core clock is exported to the output `ECLK`.

5.3 Data Memory Access Protocol

The protocol used for Data Memory access is very similar to the one described for the Instruction Memory access.

Similarly, all Data Memory accesses are generated starting from the signals `GCLK`, `DnMREQ`, and `DSEQ`. A data fetch transaction starts by driving the `DnMREQ` signal LOW. The memory reads the address on `DA[31:0]` and the signal `DSEQ` will indicate whether the access is sequential or not.

Depending on the combination of `DnMREQ` and `DSEQ` the cycle type can be derived:

Table 3: `DnMREQ` and `DSEQ` encoding

DnMREQ	DSEQ	**Cycle Type**
0	0	Non-sequential
0	1	Sequential
1	0	Internal
1	1	Coprocessor transfer

Data Memory fetches can be also marked as aborted by driving the signal `DABORT`.

Differently from the Instruction Fetch stage, there are additional signals available. First of all, `DnRW` indicates the direction of the transaction: if LOW indicates a read from the Data Memory, if HIGH indicates a write to the Data Memory.

- For all the read operations from the data memory, the signal `DDIN` is driven by the Data Memory. The data is sampled during the falling edge of `GCLK`.
- For all the write operations to the data memory, the signal `DD` is driven during the rising edge of `GCLK` an will be sampled by data memory during the falling edge of `GCLK`.

Second, the size of a transfer (read or write) can be defined using the signal `DMAS[1:0]`. The following table represents the encoding of this signal:

Table 4: DMAS[1:0] encoding

DMAS[1:0]	**Transfer Size**
00	Byte
01	Half word
10	Word
11	Reserved

Finally, a special signal DMORE is used to indicate the memory the sequentialness of a transfer one cycle ahead of DSEQ. The purpose of this signal is to give more time to the memory logic to decode the information.

5.3.1 Data Access Timing

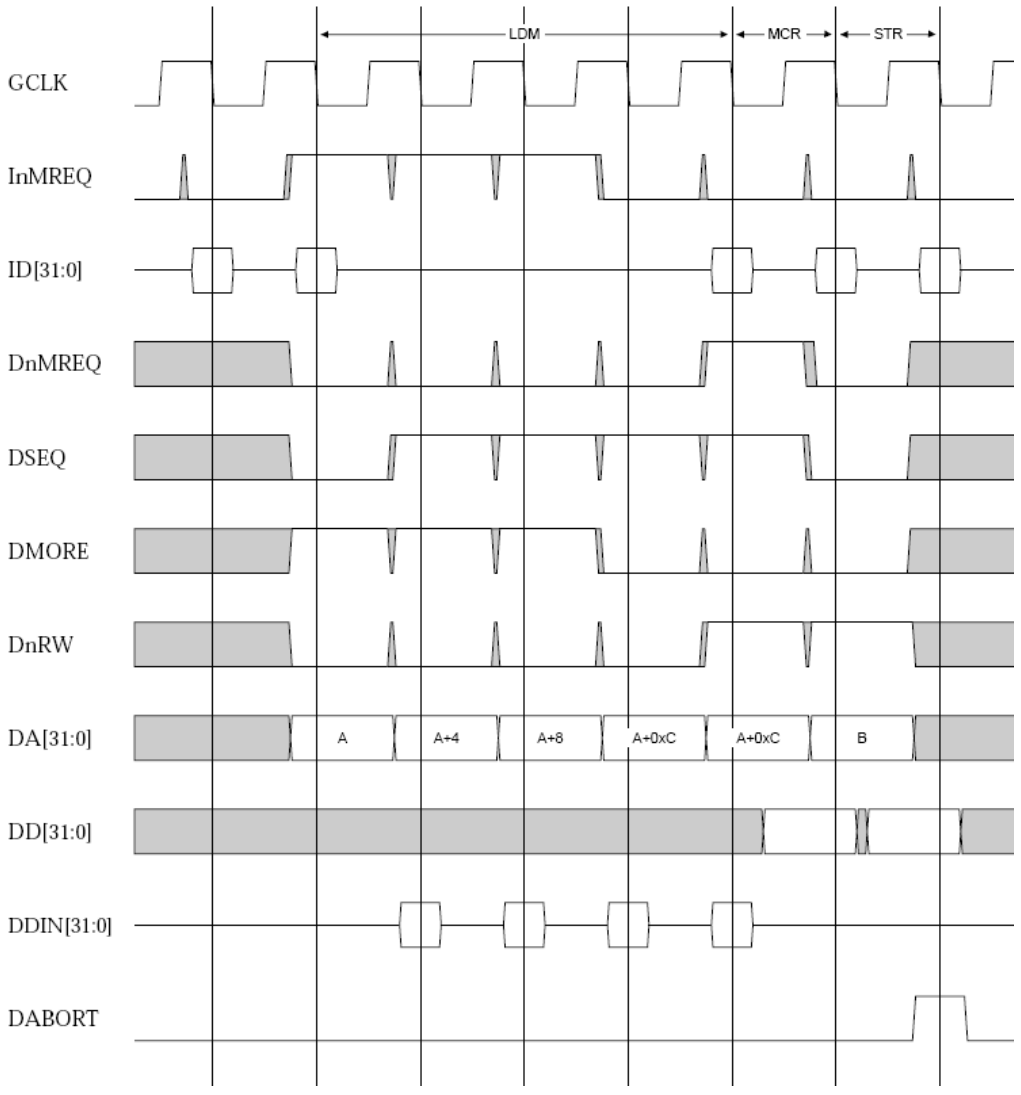

Figure 4: Data access timing diagram

5.4 Endian effects for Data Transfers

The ARM9TDMI supports both big-endian an little-endian memory fetches, resorting to an input signal BIGEND. This signal must not change at runtime. Depending on the configuration of DnRW, DA[1:0], DMAS[1:0], and BIGEND, there are different scenarios.

If DMAS indicates a word read or write, the signal BIGEND does not affect the memory access.
If DMAS instead indicates a byte/half-word:

- In case of read: one specific portion of DDIN is read. The tables 5 and 6 better explain the correct behavior.
- In case of write: the data on DD is replicated. In case of half-word writes, two copies of the same data appears on DD[31:16] and DD[15:0]. In case of byte writes, four copies of the same data appears on DD[31:24], DD[23:16] DD[15:8], and DD[7:0].

Table 5: Endian effects for 16-bit data fetches

	Little Endian	**Big Endian**
DA[1:0]	**BIGEND = 0**	**BIGEND = 1**
00	DDIN[15:0]	DDIN[31:16]
10	DDIN[31:16]	DDIN[15:0]

Table 6: Endian effects for 8-bit data fetches

	Little Endian	**Big Endian**
DA[1:0]	**BIGEND = 0**	**BIGEND = 1**
00	DDIN[7:0]	DDIN[31:24]
01	DDIN[15:8]	DDIN[23:16]
10	DDIN[23:16]	DDIN[15:8]
11	DDIN[31:24]	DDIN[7:0]

6 Register File

6.1 About the Register File

The ARM family of processors include 16 32-bit registers that are always visible. The first 13 (R0...R12) are general purpose registers, R13 is the Stack Pointer (SP), R14 is the Link Register (LR), and R15 is the Program Counter. In addition to that, there is the Current Program Status Register (CPSR), plus other 'shadow' registers that are used depending on the current Processor Mode.

The ARM9TDMI integrates a 3R2W Register File. Three concurrent data reads and two concurrent data writes (plus the program counter) are possible.
All read operations take place on the rising edge of the clock, while all write operations are set on the falling edge of the clock. This solves all the Read After Write (RAW) hazards.

6.2 Current Program Status Register

The Current Program Status Register (CPSR) is a 32-bit register that holds:

- Condition Code flags `[31:28]`:
  - (N) Negative/Less Than flag
  - (Z) Zero flag
  - (C) Carry flag
  - (V) Overflow flag
- Control Bits `[7:0]`:
  - (I) IRQ Disable
  - (F) FIQ Disable
  - (T) State Bit (ARM/Thumb)
  - (M4...M0) Mode Bits (Processor Modes)

As the name should suggest, it represents the current state of the processor. In particular, changing the Control Bits changes the current state of the processor. For instance, the State Bit (also called Thumb Bit) defines if the processor is in ARM Mode or Thumb Mode: flipping its value will change the state with immediate effect on the next instruction fetch performed.

6.3 Processor Modes

The ARM9TDMI has six different Processor Modes:

- Normal (User/System)
- Fast Interrupt Request (FIQ)
- Supervisor (Protected mode for Operating Systems)
- Abort (Instruction or Data abort)
- Interrupt Request (IRQ)
- Undefined (Undefined execution state)

Each of them share some general purpose registers, while has its own set of shadow registers that will mask the normal ones. For instance, IRQ shares all the registers except for R13 (SP) and R14 (LR).

Each time that the processor mode changes, the current CPSR is saved into a special register called SPSR. Each processor mode (except Normal) has its own SPSR that holds the previous value of CPSR.

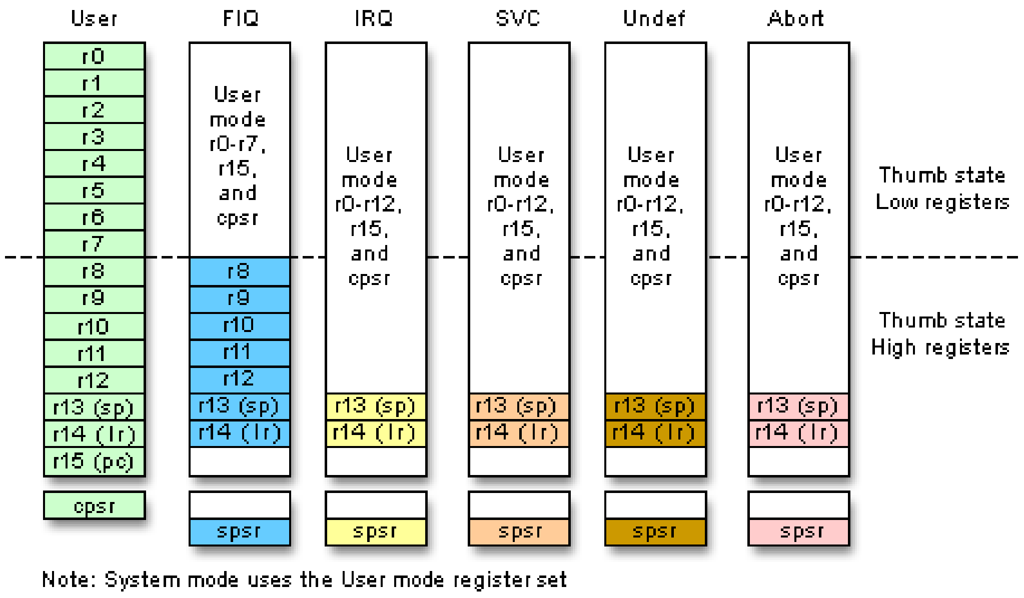
User
FIQ
IRQ
SVC
Undef
Abort
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
User mode r0-r7, r15, and cpsr
User mode r0-r12, r15, and cpsr
spsr
Thumb state Low registers
Thumb state High registers
Note: System mode uses the User mode register set

Figure 5: Register File organization of the ARM9TDMI processor

7 Control Unit

The Control Unit is in charge of performing the actual decode of the fetched instructions and sending the correct signals to the whole processor. This unit is mainly composed by two decode tables, which transform the incoming instructions into actual control signals, and several finite state machines (FSM).

7.1 Components of the Control Unit

The two decode tables are the following:

- ARM Decode Table, which takes 12 bit of any ARM instruction (`31-20, 7-4`) and generates all the control signals to be sent to the datapath;
- Thumb Decode Table, which takes a Thumb instruction and converts it into the corresponding ARM instruction, that then is sent to the first table to be decoded.

The Thumb instruction Branch and Link BL, since it has no corresponding instruction in the ARM instruction set, has been translated into a specific ARM instruction that can be executed only if the processor is in Thumb mode; in this way the behavior of the processor corresponds to the one described in the ARM Instruction Set Architecture datasheet (pages from A7-26 to 28).

The Control Unit also contains two FSMs, which are in charge of executing the multi-cycle operations that can be performed by the processor:

- Multiplier FSM, which controls the Decode and Execute stages during a multiplication;
- Load/Store Multiple (LDM/STM) FSM, which controls the Register Read, Execute, Data Memory Access and Register Write operations during a LDM/STM operation; this unit is TO BE DONE.

7.2 Control Signals

The `TMP.vhd` file contains an INCOMPLETE ARM decode table. This table generates the following control signals:

- Internal signals:
  - **Undefined Instruction** [U], which defines if the instruction is not recognized by the processor (1 if not recognized);
  - **Thumb-only Instruction** [T], which defines if the instruction can be performed only in Thumb mode and is true only for the `T_BL` instruction, which implements the Thumb branch and link instruction;
  - **Load/Store Multiple Instruction** [M], which defines if it is a LDM/STM instruction and enables the corresponding FSM;
  - **Multiply Instruction** [X], which defines if the instruction is a multiplication and enables the corresponding FSM;
- Immediate Selection signals:
  - **ALU Operand Immediate** [I], which selects as second operand of the ALU the immediate value;
  - **Shifter Operand Immediate** [i], which selects as shift value the immediate value;

- Execute Stage signals:
  - **ALU Control signals** [LLLL], which define the operation the ALU has to perform
  - **A/B Bus Pass signals** [a/b], which select the value on bus A or B as result of the Execute unit;
  - **A/B Bus Invert signals** [A/B], which select whether the values on bus A and B have to be logically negated;
  - **Carry-In Enable signal** [c], which defines if the carry is considered during the arithmetic operation;
  - **Update CPSR signal** [C], which signals to save the current flags into CPSR;
  - **Update SPSR signal** [S], which signals to save the current flags into SPSR;
- Data Memory Access Stage signals: TO BE DEFINED
- Register Write signals:
  - **Write Enable 1 signal** [1], which enables a write operation on Register File's port 1;
  - **Write Enable 2 signal** [2], which enables a write operation on Register File's port 2.

7.3 Multiplier Finite State Machine

The Multiplier Finite State Machine controls the High-Speed Multiplier of the ARM9TDMI processor. This FSM has 8 different states, each controlling the multiplier in a specific clock cycle.

- `IDLE` is the idle state, in which the multiplier is deactivated; when receiving a reset signal, the FSM enters this state;
- `DECODE` is the state in which the partial sum and carry registers are cleared through a reset;
- `DECODE_MAC` is the state in which the partial sum is initialized for a MAC operation; this state is entered as an alternative to the `DECODE` state only during MAC instructions;
- `MULT1` is the state in which the first cycle of multiplication is performed; in case of early finish the FSM could either go to `IDLE` or `SUM_LO` depending if it is a 32-bit or 64-bit multiplication;
- `MULT2` is the state in which the second cycle of multiplication is performed; in case of early finish the FSM could either go to `IDLE` or `SUM_LO` depending if it is a 32-bit or 64-bit multiplication;
- `MULT3` is the state in which the third cycle of multiplication is performed; in case of early finish the FSM could either go to `IDLE` or `SUM_LO` depending if it is a 32-bit or 64-bit multiplication;
- `MULT4` is the state in which the fourth cycle of multiplication is performed;
- `SUM_LO` is an optional state, entered only if the multiplication yields a 64-bit result; in this state, the FSM drives the ALU in order to perform the addition of the lower 32-bit of the partial sum and carry.