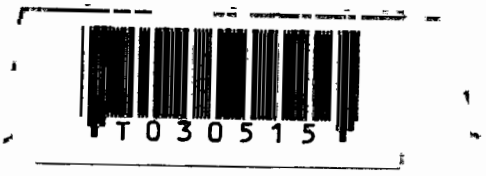
การออกแบบและพัฒนาไมโครโปรเซสเซอร์ 32 บิต

Designed & Development Microprocessor 32 Bit



โดย

นายกฤษณะ กันตังกุล

นางสาวปัญจภา เรืองสินทรัพย์

นางสาววรรณรัช สันติอมรทัต

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตร์บัณฑิต

ภาควิชาวิศวกรรมคอมพิวเตอร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2540

ปริญญานิพนธ์ปีการศึกษา 2540

ภาควิชา วิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง การออกแบบและพัฒนาไมโครโปรเซสเซอร์ 32 บิต

ผู้จัดทำ

1.	นายกฤษณะ	กันตังกุล	รหัสนักศึกษา	37014008
2.	นางสาวปัญจภา	เรืองสินทรัพย์	รหัสนักศึกษา	37014252
3.	นางสาววรรณรัช	สันติอมรทัต	รหัสนักศึกษา	37014373

อาจารย์ที่ปรึกษา

(อาจารย์อภิเนตร อูนากุล)

อาจารย์ที่ปรึกษา

(อาจารย์นวพร วรรณวิมลศรี)

ปีการศึกษา 2540

การออกแบบและพัฒนาไมโครโปรเซสเซอร์ 32 บิต

โดย

นายกฤษณะ	กันตังกุล	รหัสนักศึกษา	37014008
นางสาวปัญจภา	เรืองสินทรัพย์	รหัสนักศึกษา	37014252
นางสาววรรณรัช	สันติอมรทัต	รหัสนักศึกษา	37014373

อาจารย์ที่ปรึกษา

อ.อภิเนตร อูนากุล

อ.นวพร วรรณวิมลศรี

การออกแบบและพัฒนาไมโครโปรเซสเซอร์ 32 บิต

นายกฤษณะ กันตังกุล
นางสาวปัญจภา เรืองสินทรัพย์
นางสาววรรณรัช สันติอมรทัต
อ.อภิเนตร อูนากุล อาจารย์ที่ปรึกษา
อ.นวพร วรรณวิมลศรี อาจารย์ที่ปรึกษา
ปีการศึกษา 2540

บทคัดย่อ

โครงงานนี้เป็นการออกแบบไมโครโปรเซสเซอร์ 32 บิต ที่มีการทำงานเลียนแบบ ARM7 ซึ่งมีโครงสร้างการทำงานเป็นแบบ RISC (Reduced Instruction Set Computer) ด้วยภาษาอธิบายโครงสร้างฮาร์ดแวร์ VHDL โดยเริ่มจากการศึกษาและทำความเข้าใจในการทำงานของ ARM7 และเพื่อให้ง่ายต่อการออกแบบจึงทำการแบ่งออกเป็นส่วนย่อยๆ (Decomposition) แล้วจึงทำการทดสอบการทำงานเพื่อตรวจสอบความถูกต้อง โดยการใช้โปรแกรม V-System จากนั้นทำการรวมทุกๆ ส่วนเข้าด้วยกัน และทำการทดสอบการทำงาน จนกระทั่งได้ Soft Core ของไมโครโปรเซสเซอร์ที่สามารถจำลอง (Simulate) การทำงานได้

Designed & Development Microprocessor 32 bit

Mr. Kritsana	Kantangkul	
MissPhunjapa	Ruengsinsup	
MissWannarat	Suntiamorntut	
Mr.Apinetr	Unakul	Advisor
MissNavaporn	Wanvimonsri	Advisor
1997		

Abstract

This project is designing Microprocessor 32 bit, working similar with ARM7 which sturcture is RISC (Reduced Instruction Set Computer) by using VHDL hardware description language. The process of designing start with study ARM7 architecture, using decomposition method to design and verify correctness by testing each component with V-System. Then combine all component and testing until to obtain Soft Core of Microprocessor that can simulate.

สารบัญรูปภาพ

สารบัญตาราง

บทที่ 1
บทนำ

ปัจจุบันเทคโนโลยีได้ก้าวหน้าไปอย่างรวดเร็ว มีการนำไมโครโปรเซสเซอร์ (Microprocessor) ขนาด 32 บิตไปใช้ในอุปกรณ์อิเล็กทรอนิกส์ ที่ต้องการความสามารถ และความรวดเร็วในการประมวลผลสูงเป็นจำนวนมาก ส่งผลให้อุปกรณ์เหล่านั้นมีราคาที่สูงมาก เพราะต้องทำการสั่งซื้อไมโครโปรเซสเซอร์จากต่างประเทศ ซึ่งหากประเทศไทยสามารถที่จะออกแบบ และผลิตไมโครโปรเซสเซอร์เหล่านี้ได้เอง ก็จะทำให้ราคาของอุปกรณ์ต่างๆ ถูกลง

ไมโครโปรเซสเซอร์ มีโครงสร้างการทำงาน 2 แบบ คือ

1. Reduced Instruction Set Computer (RISC) จะมีชุดคำสั่งไม่มากนัก และเป็นคำสั่งที่ง่ายไม่ซับซ้อน ทำให้มีโครงสร้างภายในไม่ยุ่งยาก เช่น ARM7, MC88100
2. Complex Instruction Set Computer (CISC) จะมีชุดคำสั่งที่ซับซ้อน และมากกว่าแบบแรก เหมาะสำหรับการนำไปใช้ในอุปกรณ์ที่ต้องการความสามารถในการประมวลผลที่สูง โครงสร้างภายในไมโครโปรเซสเซอร์ประเภทนี้จะซับซ้อนกว่าแบบ RISC เช่น 32-bit Motorola MC68020

ขั้นตอนการออกแบบไมโครโปรเซสเซอร์มีส่วนคล้ายกับการออกแบบซอฟแวร์ หลังจากออกแบบแล้ว ถ้าเป็นซอฟแวร์ต้องใช้ภาษาต่างๆในการสร้างซอฟแวร์ขึ้นมา การออกแบบไมโครโปรเซสเซอร์ก็เช่นเดียวกัน ภาษาที่ใช้ในการอธิบายการทำงานของระบบ เราเรียกว่า Hardware Description Language (HDL) ซึ่งมีภาษาที่ใช้กันโดยทั่วไปก็คือ

1. Verilog มีโครงสร้างภาษาคล้ายภาษา C มักใช้ในงานการสร้างระบบในเชิงพาณิชย์
2. VHDL พัฒนามาจากภาษา ADA ซึ่งกระทรวงกลาโหมสหรัฐอเมริการเป็นผู้พัฒนา มีโครงสร้างทางภาษาคล้ายกับภาษา PASCAL ทุกหน่วยงานที่ต้องการที่จะออกแบบ หรือทำการพัฒนาไมโครโปรเซสเซอร์ให้กับกระทรวงนี้จะต้องใช้ภาษานี้ จึงทำให้ภาษานี้กลายเป็นภาษามาตรฐานในการนำไปออกแบบ

ภาษา VHDL มีลักษณะที่ผสมกันระหว่าง Object Oriented Language และ Concurrent Programming Language โดยมองส่วนต่างๆเป็นโมดูล (Module) หรืออ๊อปเจค (Object) และมีสัญญาณเชื่อมต่อระหว่างกัน การส่งสัญญาณต่างๆ ในระบบเกิดขึ้นพร้อมๆกันได้เช่นเดียวกับกระแสไฟฟ้าที่แยกไหลไปยังส่วนต่างๆของวงจร

หลังจากเขียนโปรแกรมอธิบายเรียบร้อยแล้ว ถ้าเป็นซอฟแวร์ต้องใช้คอมไพล์เลอร์ทำการคอมไพล์ เพื่อสร้างไบนารี่โค้ด (binary code) ที่ใช้ในการทำงาน การออกแบบไมโครโปรเซสเซอร์ก็มีคอมไพล์เลอร์ (Compiler) เช่นเดียวกัน แต่เมื่อคอมไพล์ (Compile) แล้วแทนที่จะได้วงจรทันทีจะได้ข้อมูลที่ใช้ในการจำลองการทำงานก่อน เพื่อตรวจสอบความถูกต้องของการทำงาน ซึ่งก็จะได้คอมโพเนนท์ที่สามารถจะนำมาใช้ได้อีก (Reuseable Component) หรือที่เราเรียกกันว่า **"Soft Core"** หลังจากนั้นจึงจะทำการสังเคราะห์ (Synthesize) สร้างวงจรต่อไป

1.1 ความสำคัญและความเป็นมาของปัญหา

ARM7 เป็นไมโครโปรเซสเซอร์ 32 บิต ของบริษัท ARM ที่มีโครงสร้างการทำงานเป็นแบบ RISC และมีโหมดการทำงานได้หลายโหมดด้วยกัน ซึ่งถูกนำไปใช้ในอุปกรณ์อิเล็กทรอนิกส์ที่ต้องการให้มีขนาดเล็กกะทัดรัด และประหยัดพลังงานเป็นจำนวนมาก เช่น ใช้ใน Smart Cards หากเราสามารถที่จะทำการออกแบบ และผลิตไมโครโปรเซสเซอร์นี้ได้เอง ก็จะทำให้ต้นทุนในการผลิตอุปกรณ์เหล่านี้ต่ำลง และเป็นการช่วยลดการขาดดุลได้อีกด้วย

1.2 วัตถุประสงค์

เป็นการออกแบบเพื่อให้ได้ Soft Core ของไมโครโปรเซสเซอร์ซึ่งสามารถทำงานเลียนแบบการทำงานของ ARM7 ใน User Mode ที่สามารถนำมาจำลองการทำงานได้ ด้วยการใช้ภาษา VHDL เป็นตัวอธิบายการทำงาน และการออกแบบ

1.3 ขั้นตอนในการทำโครงงาน

1. ศึกษาการใช้ภาษา VHDL ซึ่งเป็นตัวที่ใช้ในการอธิบายการทำงานของไมโครโปรเซสเซอร์ที่จะทำการออกแบบ รวมทั้งการใช้ V-System เพื่อการคอมไพล์และตรวจสอบสัญญาณ
2. ศึกษาการทำงานของรูปแบบคำสั่งและเวลาที่ใช้ในไมโครโปรเซสเซอร์ ARM7 อย่างละเอียด
3. ออกแบบภาพรวมของระบบ แล้วทำการเขียนการทำงานของระบบโดยการแบ่งเป็นส่วนต่างๆ ดังนี้คือ ALU, Shifter, Multiply, Register file และ Control Unit พร้อมทั้งทดสอบการทำงานโดยการใช้ Test Bench และ Memory ที่เขียนขึ้นเอง
4. ทำการรวมแต่ละส่วนเข้าด้วยกันแล้วทดสอบการทำงาน ด้วยการใช้ Test Bench ทดสอบความถูกต้องในการทำงานของคำสั่งต่างๆโดยดูจากกราฟที่ได้จาก V-System กับค่าที่ได้จาก Debugger ของ ARM 7 ตัวจริง

1.4 ประโยชน์ที่คาดว่าจะได้รับ

1. เข้าใจขั้นตอนการออกแบบ และโครงสร้างการทำงานของไมโครโปรเซสเซอร์ มากยิ่งขึ้น
2. เข้าใจการใช้งานภาษา VHDL และ V-System ได้เป็นอย่างดี
3. ได้ soft core ของระบบที่มีการทำงานเลียนแบบ ARM 7 เพื่อประโยชน์ในด้านการนำไปใช้งานและการผลิตไมโครโปรเซสเซอร์ 32 บิตต่อไป เนื่องจากราคาต้นทุนในการผลิตจะต่ำลง
4. รู้จักขั้นตอนการทำงาน การวางแผนงาน การตรวจสอบการทำงาน และแนวทางการแก้ไขปัญหาที่เกิดขึ้นได้

บทที่ 2
ARM7

ในบทนี้จะอธิบายถึงรายละเอียดต่างๆ ของ ARM7 เพื่อให้เข้าใจถึงสถาปัตยกรรม (Architecture), การทำงาน, ชุดคำสั่งต่างๆ ของ ARM7 ที่โครงงานนี้จะต้องทำการออกแบบไมโครโปรเซสเซอร์เลียนแบบการทำงาน ให้ได้

2.1 ภาพรวมของ ARM 7

ARM7 เป็นไมโครโปรเซสเซอร์ 32 บิต ที่ใช้พลังงานต่ำ มีขนาดเล็ก ซึ่งถูกนำไปใช้ใน ASICs (Application-specific integrated circuits) หรือ CSICs (Customer-specific integrated circuits) โครงสร้างของ ARM 7 เป็นแบบ RISC (Reduced Instruction Set Computer) คือ มีชุดคำสั่งที่ง่าย ไม่ซับซ้อนเหมือนแบบ CISC การตอบสนองสัญญาณอินเทอร์รัปท์ (Interrupt Respond) เกือบจะทันกาล (Real Time) เหมาะสำหรับการนำไปใช้กับอุปกรณ์ที่ต้องการให้มีขนาดเล็ก ประหยัดพลังงาน และ ความรวดเร็ว

Feature Summary : 32 bit RISC (data & address bus)
: 25 MHz (25 MIPS) , 3 V
: ใช้ไฟ 0.6 mA/MHz , 3 V fabric 0.8 m CMOS
: สนับสนุนการตอบสนองระบบอินเทอร์รัปท์ (Interrupt System Support)
: สนับสนุนการโปรแกรมโดยภาษาระดับสูง (High Level Language Support)
: ชุดคำสั่งง่าย แต่มีประสิทธิภาพ

ARM7 ถูกนำไปใช้ในด้านต่างๆ มากมาย ยกตัวอย่างเช่น

Telecommunication	GSM terminal controller
Data communication	Protocol conversion
Portable computing	Palmtop computer
Portable instrument	Handhold data acquisition unit
Automotive	Engine management unit
Information System	Smart cards
Imaging	JPEG controller

2.2 ARM7 Architecture

ARM7 มี Block Diagram เป็นดังรูป 2-1 ซึ่งสามารถทำงานได้ทั้งสิ้น 6 โหมดการทำงานด้วยกัน โดยที่โครงงานนี้จะทำการออกแบบไมโครโปรเซสเซอร์ที่สามารถทำงานได้เฉพาะ User Mode เท่านั้น โหมดการทำงานมีดังนี้

1. User (Unprivileged mode) เป็นโหมดการทำงานปกติ
2. FIQ จะเข้าสู่โหมดการทำงานนี้ เมื่ออินเทอร์รัปท์ (Interrupt) ที่มี priority สูงสุด เกิดขึ้น
3. IRQ จะเข้าสู่โหมดการทำงานนี้ เมื่ออินเทอร์รัปท์ที่มี priority ต่ำสุด เกิดขึ้น
4. Supervisor เมื่อเกิดการรีเซต (Reset) หรือมีการใช้คำสั่ง Software Interrupt ก็จะเข้าสู่โหมดการทำงานนี้
5. Abort โหมดการทำงานนี้จะใช้เมื่อเกิดการความผิดพลาดในการจัดการข้อมูลกับหน่วยความจำ เช่น อ้างอิงตำแหน่งในหน่วยความจำเกินตำแหน่งสูงสุด เป็นต้น
6. Undefined ใช้โหมดการทำงานนี้เมื่อเกิดการใช้คำสั่งที่ ARM7 ไม่รู้จัก

ARM7 มีรีจิสเตอร์ (Register) ทั้งสิ้น 37 ตัว โดยเป็นรีจิสเตอร์ขนาด 32 บิต ซึ่งในแต่ละโหมดการทำงานสามารถใช้รีจิสเตอร์แตกต่างกัน รีจิสเตอร์ทั้ง 37 ตัว มีดังนี้

- Program Counter 1 ตัว
- Current Program Status Register (CPSR) 1 ตัว
- Saved Program Status Register (SPSR) 5 ตัว
- รีจิสเตอร์ทั่วไป 30 ตัว

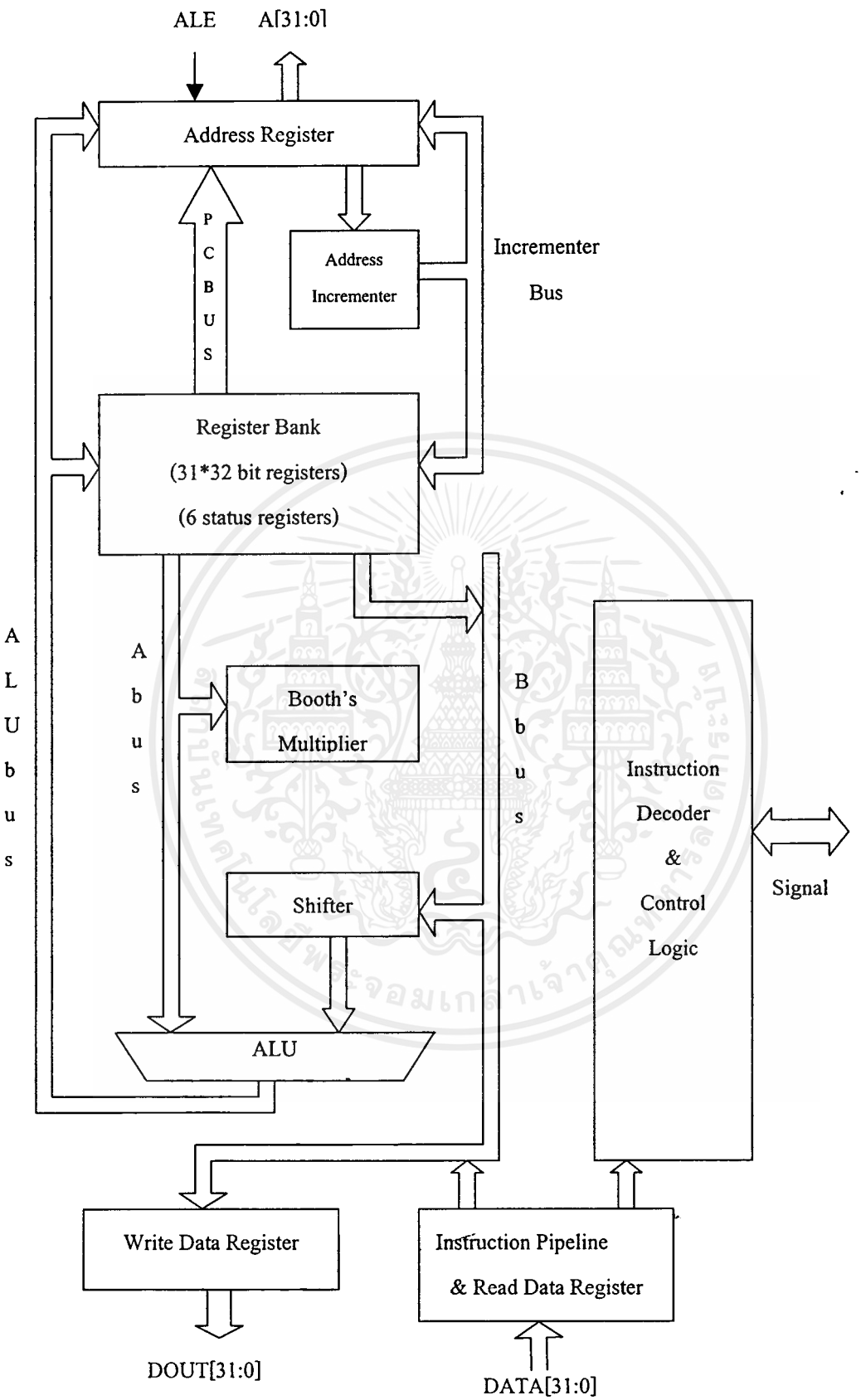
รายละเอียดของรีจิสเตอร์ที่ในแต่ละโหมดสามารถใช้ได้เป็นดังรูป 2-2 ซึ่งที่เป็นสีเข้มทั้งหมดรวมกันได้เป็น 37 ตัว

Condition Flag

ARM7 มี Flag ทั้งสิ้น 4 ตัวด้วยกัน ดังนี้

Flag	Logical Instruction	Arithmetic Instruction
Negative (N = '1')	ไม่เปลี่ยนแปลงค่า	บิตที่ 31 ของผลลัพธ์ที่ได้
Zero (Z = '1')	เมื่อผลลัพธ์เป็นศูนย์	ผลลัพธ์ที่ได้เป็นศูนย์
Carry (C = '1')	หลังการ Shift แล้วได้ค่า '1' เป็น Carry Out ออกมา	ผลลัพธ์ที่ได้มากกว่า 32 บิต
Overflow (V = '1')	ไม่เปลี่ยนแปลงค่า	ผลลัพธ์ที่ได้มากกว่า 31 บิต จะเป็นตัวบอกว่าเกิดการผิดพลาดใน Sign bit ของ Signed number

ตาราง 2-1 Condition Flag



รูปภาพ 2-1 Block Diagram ของ ARM7 Microprocessor

General Registers and Program Counter

User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8 FIQ	R8	R8	R8	R8
R9	R9 FIQ	R9	R9	R9	R9
R10	R10 FIQ	R10	R10	R10	R10
R11	R11 FIQ	R11	R11	R11	R11
R12	R12 FIQ	R12	R12	R12	R12
R13 (SP)	R13 FIQ	R13 SVC	R13 ABT	R13 IRQ	R13 UNDEF
R14 (LR)	R14 FIQ	R14 SVC	R14 ABT	R14 IRQ	R14 UNDEF
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

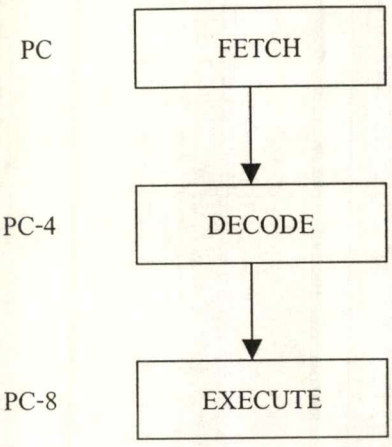
Program Status Registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_FIQ	SPSR_SVC	SPSR_ABT	SPSR_IRQ	SPSR_UNDE

รูปภาพ 2-2 Register Configuration

Instruction Pipeline

ARM7 มีการทำงานเป็นเป็นแบบไปป์ไลน์ (Pipeline) เพื่อเพิ่มความเร็วในการไหลของคำสั่งไปให้ยังตัวประมวลผล (Processor) คือ ในขณะที่กำลังทำการเอ็กซิคิวส์ (Execute) คำสั่งหนึ่งอยู่ ก็จะมีการถอดรหัส (Decode) คำสั่งถัดมาจากคำสั่งนั้น และมีการนำคำสั่ง (Fetch) ถัดจากคำสั่งที่กำลังถอดรหัสอยู่มาด้วย แต่ ARM7 จะแตกต่างไปเล็กน้อย คือ โดยเฉลี่ยแล้วแต่ละคำสั่งจะใช้ไซเคิลในการเอ็กซิคิวส์ มากกว่า 1 ไซเคิล ดังนั้นจะต้องเกิดการยืดไปป์ไลน์ หรือที่เรียกกันว่า " การ Store Pipeline " ซึ่งไปป์ไลน์ของ ARM7 นั้นมี 3 สเตต (State) ด้วยกันดังรูป 2-3



รูปภาพ 2-3 Instruction Pipeline

2.3 ชุดคำสั่ง (Instruction Set)

คำสั่งที่มีการใช้งานใน ARM7 แบ่งกลุ่มเป็น 6 ชุดคำสั่ง ซึ่งแต่ละชุดคำสั่งสามารถมีเงื่อนไข (condition) ได้ โดย Condition Field มีทั้งสิ้น 16 กรณี และมีรายละเอียดของแต่ละคำสั่งดังนี้

Condition Field

รูปภาพ 2-4 Condition Field

"0000" = EQ : Z set (equal)

"0001" = NE : Z clear (not equal)

"0010" = CS : C set (unsigned higher or same)

"0011" = CC : C clear (unsigned lower)

"0100" = MI : N set (negative)

"0101" = PL : N clear (positive or zero)

"0110" = VS : V set (overflow)

"0111" = VC : V clear (no overflow)

"1000" = HI : C set and Z clear (unsigned higher)

"1001" = LS : C clear or Z set (unsigned lower or same)

"1010" = GE : N set and V set ,or N clear and V clear (greater or equal)

"1011" = LT : N set and V clear ,or N clear and V set (less than)

"1100" = GT : Z clear and either N set and V set ,or N clear and V clear (Greater than)

"1101" = LE : Z set ,or N set and V clear ,or N clear and V set (less than)

"1110" = AL : Always นั่นคือ คำสั่งจะทำงานโดยไม่คำนึงถึงค่า flag เลย

"1111" = NV : Never

หมายเหตุ

ในคำสั่ง NOP จะเหมือนกับคำสั่ง MOV Ro,Ro Bit condition field จะเป็น "AL"

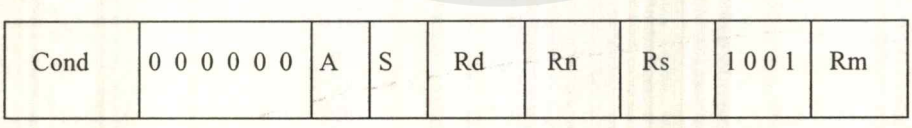
1. Branch and Branch with link

สามารถกระโดดไปทำงานได้ +/- 32 Mbytes

รูปภาพ 2-5 Branch Instruction

การทำคำสั่ง BL ค่าเก่าใน PC จะถูกนำไปเก็บใน R14 ก่อนที่ PC จะเปลี่ยนไปเป็นค่าใหม่ที่ต้องการกระโดดไปทำงาน โดยที่ค่าของ OFFSET นั้นตัวคอมไพล์เลอร์จะทำการคำนวณจากการนำค่าตำแหน่งที่ต้องการกระโดดไป ลบด้วยตำแหน่งของคำสั่งนั้น แล้วลบออกอีก 8 (ตำแหน่งใหม่ – ตำแหน่งปัจจุบัน – 8) ส่วนการคืนค่ากลับไปยังที่ตำแหน่งเดิมเมื่อเป็นคำสั่ง BL นั้นจะใช้คำสั่ง MOV PC,R14

2. Multiply and Multiply-accumulate



รูปภาพ 2-6 Multiply Instruction

A : Accumulate

"0" = multiply only

"1" = multiply and accumulate

S : Set condition code

"0" = do not alter condition codes

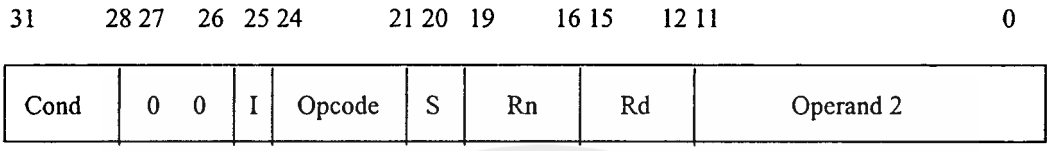
"1" = set condition code

Rd : Destination Register

Rn ,Rs ,Rm : Operand Register

เป็นการนำโอเปอร์แลนด์ Rs มาทำการคูณกับโอเปอร์แลนด์ Rm แล้วบวกด้วย Rn หากเป็นการทำคำสั่ง Multiply-accumulate ซึ่ง ARM7 นั้นใช้การคูณด้วยวิธี Booth's Algorithm ที่จะกล่าวถึงรายละเอียดในภายหลัง

3. Data Operation



รูปภาพ 2-7 Data Operation Instruction

I : Immediate Operand

"0" = operand 2 is a register value

"1" = operand 2 is an immediate value

Opcode : operation Code

"0000"	AND
"0001"	EOR
"0010"	SUB
"0011"	RSB
"0100"	ADD
"0101"	ADC
"0110"	SBC
"0111"	RSC
"1000"	TST
"1001"	TEQ
"1010"	CMP
"1011"	CMN
"1100"	ORR
"1101"	MOV
"1110"	BIC
"1111"	MVN

S : Set condition codes

"0" = do not alter condition codes

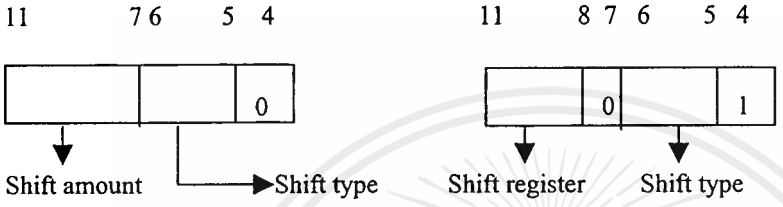
"1" = set condition codes

Rn : 1st operand register

Rd : Destination register

เป็นการทำ Arithmetic & Logic Operation โดยที่ Operand2 นั้นสามารถที่ทำการ Shift หรือ Rotate Right ก่อนที่นำมาทำการคำนวณต่างๆ ได้ ซึ่งจะใช้เพียงคำสั่งเดียวก็สามารถที่จะทำถึง 2 คำสั่งได้ ทำให้เวลาในการทำงานรวดเร็วขึ้นได้

Operand2 (ARM Shift Operation)



รูปภาพ 2-8 ARM Shift Operations

Shift amount : เป็นจำนวนเต็มบวก 5 บิต

Shift type :

"00" logical left

"01" logical right

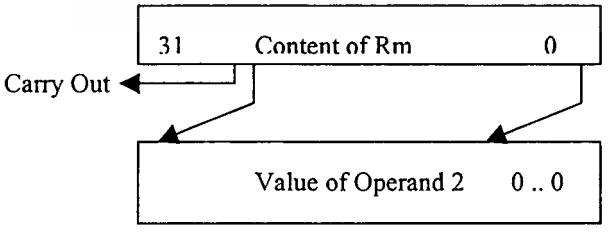
"10" arithmetic right

"11" rotate right

Shift register : จำนวนครั้งในการ Shift หรือ Rotate ใช้ค่า 8 บิตล่างของค่าในรีจิสเตอร์นั้นๆ

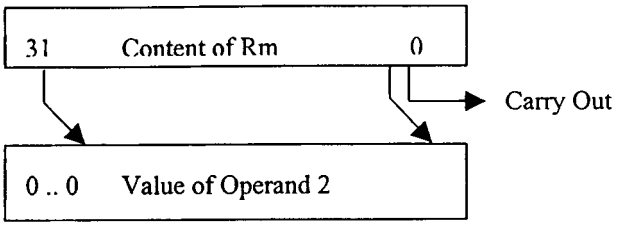
รายละเอียดของ Shift type มีดังนี้

1. Logical Shift Left (LSL)



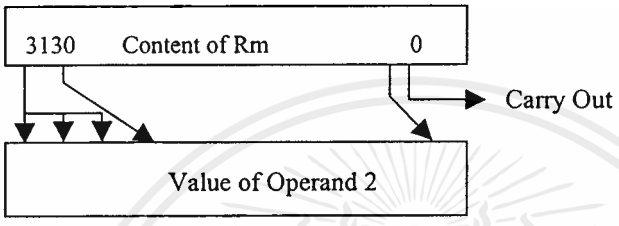
รูปภาพ 2-9 Logical Shift Left

2. Logical Shift Right (LSR)



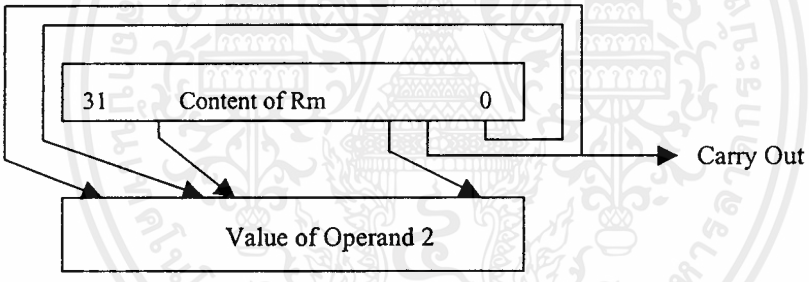
รูปภาพ 2-10 Logical Shift Right

3. Arithmetic Shift Right (ASR)



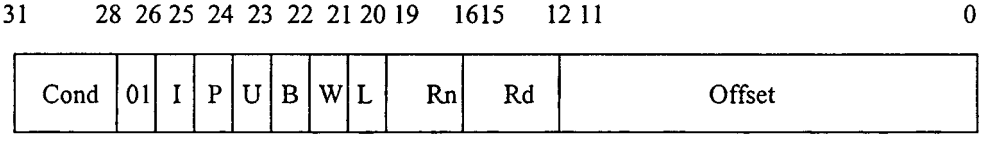
รูปภาพ 2-11 Arithmetic Shift Right

4. Rotate Right



รูปภาพ 2-12 Rotate Right

4. Single Data Transfer



รูปภาพ 2-13 Single Data Transfer

Rd : Source/Destination register

Rn : Base register

L : Load/Store bit

"0" = Store

"1" = Load

W : Write-back bit

"0" = no write-back

"1" = write address into base

B : Byte/Word bit

"0" = transfer word quantity

"1" = transfer byte quantity

U : Up/Down bit

"0" = down ;subtract offset from base

"1" = up ; add offset before transfer

P : Pre/Post indexing bit

"0" = post ; add offset after transfer

"1" = Pre ; add offset to base

I : Immediate offset

"0" = offset is an immediate value

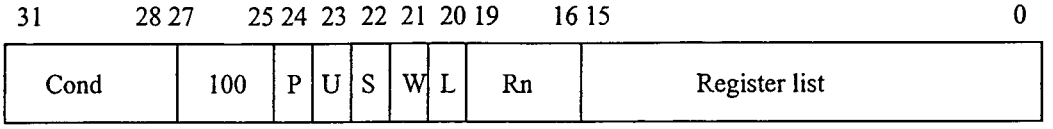
"1" = offset is a register

เป็นการย้ายข้อมูลระหว่างรีจิสเตอร์กับค่าในหน่วยความจำเท่านั้น โดยที่สามารถที่จะทำการนำค่าจากหน่วยความจำไปเก็บลงในรีจิสเตอร์ (Load Instruction) หรือนำค่าจากรีจิสเตอร์ไปเก็บลงยังหน่วยความจำ (Store Instruction) โดยที่ค่าของ OFFSET นั้นก็จะเหมือนกับค่าของ Operand2 คือ สามารถที่จะเป็นการ Shift หรือ Rotate Right ได้ก่อนที่จะทำการย้ายค่าข้อมูลไปยังตำแหน่งของผลลัพธ์นั้น แต่ว่าจะไม่สามารถทำการ Shift หรือ Rotate Right โดยใช้จำนวนครั้งในการ Shift (Shift amount) จากค่า 8 บิตล่างของค่าในรีจิสเตอร์

หากเป็นการ Load แบบเป็นไบต์ ค่าที่โหลดมาได้จากหน่วยความจำจะถูกนำไปใส่ไว้ใน 8 บิตล่างของรีจิสเตอร์ และบิตที่เหลือจะถูกเติมด้วยค่าศูนย์ ส่วนหากเป็นการ Store แบบเป็นไบต์ ค่าที่จะนำไปเก็บยังหน่วยความจำจะเป็นค่า 8 บิตล่างของรีจิสเตอร์นั้น โดยที่จะนำค่านี้ใส่ลงไปทั้ง 4 ไบต์ของ Data Out ซึ่งก็คือ ทำการก็อปปี้ค่า 8 บิตล่างนี้ไปไว้ในทั้ง 4 ไบต์ของ Data Out นั่นเอง

การย้ายข้อมูลสามารถที่จะเป็นแบบ Pre-Index คือ ทำการเปลี่ยนตำแหน่งก่อนที่จะย้ายค่าข้อมูลจากรีจิสเตอร์ หรือนำค่าจากรีจิสเตอร์ไปเก็บ ส่วนอีกแบบก็คือ Post-Index จะเป็นการนำค่าข้อมูลไปเก็บยังหน่วยความจำ หรือนำค่าจากหน่วยความจำตำแหน่งนั้นมาไว้ในรีจิสเตอร์ก่อนที่จะทำการเปลี่ยนค่าตำแหน่ง โดยที่สามารถจะเป็นการเปลี่ยนค่าตำแหน่งแบบถอยหลังลงไป (ลดค่า) หรือเป็นแบบก้าวหน้าขึ้นไป (เพิ่มค่า) ก็ได้ และสามารถที่จะกำหนดให้นำค่าตำแหน่งที่เปลี่ยนแปลงแล้วไปเขียนลงรีจิสเตอร์ที่เป็นตัวชี้ตำแหน่งหรือไม่ก็ได้ด้วยเช่นกัน

5. Block Data Transfer



รูปภาพ 2-14 Block Data Transfer

Rn : Base register

L : Load/Store bit

"0" = Store to memory

"1" = Load from memory

W : write-back bit

"0" = no write-back

"1" = write address into base

S : PSR & Force user bit

"0" = do not load PSR or force user mode

"1" = load PSR or force user mode

U : Up/Down bit

"0" = down ;subtract offset from base

"1" = up ; add offset to base

P : Pre/Post indexing bit

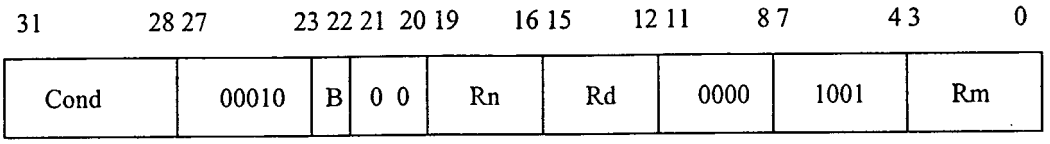
"0" = post ; add offset after transfer

"1" = pre ; add offset before transfer

เป็นการย้ายข้อมูลระหว่างรีจิสเตอร์ และหน่วยความจำเท่านั้นเช่นกัน แต่จะสามารถทำการย้ายได้มากกว่า 1 รีจิสเตอร์ หรือ 1 ตำแหน่ง โดยสามารถที่จะทำการย้ายค่าในรีจิสเตอร์หลายตัวไปไว้ในหน่วยความจำ ณ ตำแหน่งที่ต้องการไล่มาเรื่อยๆ จนกระทั่งครบจำนวนที่ต้องการย้าย (Load Multiple) หรือ ทำการย้ายค่าจากในหน่วยความจำไปไว้ในรีจิสเตอร์ที่ต้องการไล่มาเรื่อยๆ จนกระทั่งครบ (Store Multiple) ก็ได้เช่นกัน โดยหากต้องการใช้รีจิสเตอร์ตัวใด บิตนั้นๆ ใน Register List ก็จะถูกทำให้มีค่าเป็น 1

คำสั่งนี้ทำให้สามารถทำการย้ายค่าได้หลายๆ ค่าภายในคำสั่งเดียวเช่นกัน เป็นการประหยัดเวลาในการทำงานลงได้ แทนที่จะต้องสั่งคำสั่ง Load หรือ Store ไล่มาเรื่อยๆ จนกว่าจะครบ

6. Data Swap



รูปภาพ 2-15 Data Swap

B : Byte/Word bit

"0" = transfer word quantity

"1" = transfer byte quantity

เป็นการสลับค่าข้อมูลระหว่างรีจิสเตอร์กับหน่วยความจำ โดยที่ระหว่างการทำงานคำสั่งนี้ อุปกรณ์อื่นๆ จะไม่สามารถทำการจัดการใดๆ (Access) กับหน่วยความจำได้ เพราะจะทำให้เกิดความผิดพลาดได้ โดย ARM7 จะมีการส่งสัญญาณ LOCK ไปบอก ตัวควบคุมหน่วยความจำ (Memory Controller) ซึ่งคำสั่งนี้เป็นการทำคำสั่ง Load แล้วตามด้วยคำสั่ง Store นั่นเอง ดังนั้นหากเป็นการทำการ Swap แบบเป็นไบต์ ก็จะเหมือนกับการ Load เป็นไบต์ แล้วตามด้วยการ Store เป็นไบต์นั่นเอง

2.4 Multiply Booth

Booth's Algorithm เป็นรูปแบบของการคูณที่ใช้การสร้าง Booth Digit จากตัวคูณ ในที่นี้เป็น Booth 2 Encoding โดยวงรอบของการคูณที่มากที่สุดจะเท่ากับ จำนวนบิตของตัวคูณ หาร 2 ซึ่งใน ARM7 เป็น 32 บิต ดังนั้นวงรอบการคูณที่มากที่สุดจะเท่ากับ 16 จำนวนวงรอบที่ใช้จะแปรผันตามค่าของตัวคูณ ดังตาราง 2-2

การสร้าง Booth Digit จะใช้การตัดตัวคูณทีละ 3 บิตโดยในครั้งแรกจะมีการเพิ่มบิตพิเศษเข้าไปต่อหลังบิตต่ำสุดและให้ค่าเป็นศูนย์ 3 บิตถัดไปจะตัดเอาบิตบนของการจับครั้งแรกมาเป็นบิตล่าง และจะทำไปเรื่อยจนกว่าบิตที่เหลือเป็นศูนย์หมดหรือตัดครบ 16 ครั้ง โดยในแต่ละ 3 บิตที่ตัดมาได้จะมีความหมายดังตาราง 2-3

Min	Max	Speed
0	1	1S + 1I
2	7	1S + 2I
8	31	1S + 3I
32	127	1S + 4I
128	511	1S + 5I
512	2047	1S + 6I
2048	8191	1S + 7I
8192	32767	1S + 8I
32768	131071	1S + 9I
131072	524287	1S + 10I
524288	2097151	1S + 11I
2097152	8388607	1S + 12I
8388608	33554431	1S + 13I
33554432	134217727	1S + 14I
134217728	536870911	1S + 15I
536870912	2147483648	1S + 16I

ตาราง 2-2 ค่าความเร็วที่ใช้ในการคูณ

Multiplier Bits	Selection
000	+ 0
001	+ Multiplicand
010	+ Multiplicand
011	+ 2 x Multiplicand
100	- 2 x Multiplicand
101	- Multiplicand
110	- Multiplicand
111	- 0

ตาราง 2-3 Multiplier bit และความหมาย

2.5 Pipeline Hazards

ปัญหาที่เกิดขึ้นกับการรบกวนการทำงานของไปป์ไลน์ เราจะเรียกว่า Pipeline Hazards เช่น การเกิดความจำเป็นที่ต้องมีการใช้หน่วยความจำในช่วงของการเฟสคำสั่ง เพื่อเก็บผลลัพธ์ที่ได้จากช่วงการเก็บค่าในเวลาเดียวกัน อาจพบความยุ่งยากถ้าหากคำสั่งของอีกตัวขึ้นอยู่กับคำสั่งของผลอีกคำสั่งหนึ่ง หรือการกระโดดข้ามการทำงาน เราจำเป็นต้องทำการอินเตอร์รัปท์ (Interrupt) การไหลของคำสั่งในไปป์ไลน์ และเปลี่ยนไปทำงานของคำสั่งที่กระโดดข้ามไป Pipeline Hazards แบ่งได้เป็น 3 ชนิด

2.5.1 Structural Hazard : เป็นปัญหาที่เกิดขึ้นเมื่อไม่สามารถสนับสนุนการทำงานของคำสั่งในเวลาเดียวกันได้

2.5.2 Data Hazard : เมื่อคำสั่งขึ้นกับผลของคำสั่งถัดไป อาจจำแนกออกได้เป็น (ให้คำสั่ง i เกิดก่อนคำสั่ง j)

1. RAW – Read After Write คำสั่ง j พยายามที่จะอ่าน Source ก่อนหน้าที่คำสั่ง i จะเขียน
2. WAR – Write After Read คำสั่ง j พยายามที่จะเขียนลงในปลายทางก่อนที่คำสั่งจะถูกอ่าน
3. WAW – Write After Write คำสั่ง j พยายามที่จะเขียนลงในโอเปอร์แลนด์ (Operand) ก่อนหน้าที่จะถูกเขียนโดยคำสั่ง i (สูญเสียการเขียน) สิ่งนี้สามารถเกิดขึ้นได้ในไปป์ไลน์ที่มีการเขียนอยู่มากกว่า 1 สเตต สามารถแก้ไขได้ถ้าอนุญาตให้เกิดการเขียนในช่วงของการเขียนข้อมูลกลับเท่านั้น

2.5.3 Control Hazard : คำสั่งใดๆที่ผลต่อการเปลี่ยนคำสั่งในไปป์ไลน์ หรือการทำคำสั่งการกระโดดไปทำงานยังตำแหน่งอื่น (Branch หรือ Jump) ซึ่งจะก่อให้เกิดปัญหาเนื่องจากว่า การทำงานแบบไปป์ไลน์ จะมีการนำคำสั่งถัดมาเข้ามาไว้ล่วงหน้า ดังนั้นหากมีการกระโดดไปยังตำแหน่งอื่น ก็จะก่อให้เกิดข้อผิดพลาดขึ้นได้ ซึ่งวิธีการแก้ปัญหานั้นมีด้วยกัน 3 วิธีคือ

1. Branch Prediction ในบางครั้งเราสามารถที่จะทำการคาดคะเนตำแหน่งที่จะกระโดดไปทำงานได้ ยกตัวอย่างเช่น หากเป็นการทำงานแบบวนลูป (Loop) ซึ่งตำแหน่งที่จะกระโดดไปทำงานอีกครั้งเมื่อครบลูป คือตำแหน่งเริ่มต้นของลูป ดังนั้นทุกๆ ครั้งที่จะต้องทำการกระโดดไปทำงาน ก็สามารถที่จะรู้ได้ทันทีว่าตำแหน่งที่จะกระโดดไปนั้นเป็นตำแหน่งใด ยกเว้นเมื่อเป็นการเริ่มต้นทำงานลูป และสิ้นสุดการทำงานของลูปเท่านั้นที่เราไม่สามารถคาดเดาได้ เป็นต้น
2. Branch History เป็นการเก็บสถิติว่า หากเป็นตำแหน่งเดิมที่ได้ทำการกระโดดไปแล้วในครั้งก่อน หรือหากตำแหน่งนี้ถูกกระโดดไปทำงานติดต่อกันหลายๆ ครั้ง แล้ว ก็น่าที่จะเป็นตำแหน่งนี้อีกครั้งที่จะกระโดดไปทำงาน เป็นต้น

3. Delay Branch การทำงานแบบไปป์ไลน์นั้นทำให้เกิดการนำคำสั่งถัดมาเข้ามาไว้แล้ว ดังนั้นจะต้องไม่ทำการถอดรหัส หรือเอ็กซิคิวส์คำสั่งถัดมา หรือก็คือคำสั่งที่ได้นำเข้ามารอไว้ หากคำสั่งที่กำลังทำการเอ็กซิคิวส์อยู่นั้นเป็นคำสั่ง Branch นั่นก็คือการตัดหรือทิ้งคำสั่งเหล่านั้นไป หรือที่เราเรียกว่า " การ Store Pipeline " นั่นเอง

บทที่ 3
หลักการออกแบบระบบดิจิตอล (digital) ด้วยภาษา VHDL

ในบทนี้ได้อธิบายถึงขั้นตอนการออกแบบระบบด้วยภาษา VHDL เพื่อให้เข้าใจขั้นตอนการทำงาน รวมถึงรูปแบบการเขียน Test Bench ที่นำมาใช้ทดสอบระบบที่ได้ออกแบบ เพื่อตรวจสอบความถูกต้อง

3.1 การออกแบบระบบดิจิตอล

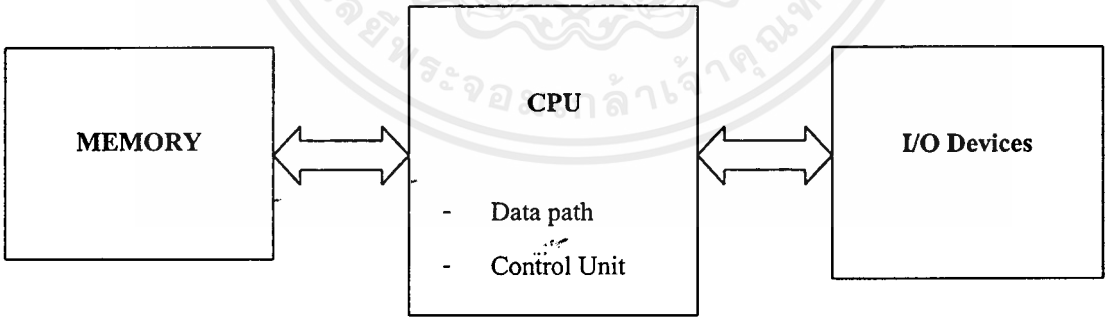
3.1.1 ระบบดิจิตอลคอมพิวเตอร์ (Digital Computer Systems)

ในโลกปัจจุบันนี้เรานำดิจิตอลมาช่วยในการอำนวยความสะดวก และเพิ่มความปลอดภัยให้กับมนุษย์มากมาย เช่น

- ระบบควบคุมการบิน (Aircraft Control)
- โรงงานพลังงานนิวเคลียร์ (Nuclear Plants)
- เครื่องตรวจจับการเต้นของหัวใจ (Heart Pacemakers)
- ระบบที่เกี่ยวกับอาวุธ (Weapon Systems)
- การทำงานในธนาคาร และประกัน (Banking and Insurance Operations)

โครงสร้างของคอมพิวเตอร์ แบ่งเป็นส่วนหลักๆได้ดังนี้

- หน่วยประมวลผล (Control Process Unit(CPU) , or Processor)
- ระบบหน่วยความจำ (Memory System)
- อุปกรณ์รับและแสดงผลข้อมูล (I/O Devices)



รูปภาพ 3-1 โครงสร้างของระบบคอมพิวเตอร์

คำสั่งที่ใช้ในการสั่งให้ตัวประมวลผล (Processor) ทำงานนั้นจะเก็บอยู่ในหน่วยความจำ (Memory) พร้อมกับข้อมูล (data items) ที่จะใช้ในการทำงาน (operate)

ในส่วนของหน่วยประมวลผลประกอบด้วย

1. ส่วนของการจัดการข้อมูล (Data Path)
   - จะมีริจิสเตอร์ เพื่อเก็บข้อมูล

- หน่วยที่ใช้ในการทำงานเกี่ยวกับข้อมูลเช่น ALU

2. หน่วยควบคุม (Control Unit)
   - นำคำสั่งถัดไปมาจากหน่วยความจำ
   - ถอดรหัส คำสั่งให้เป็นความหมายที่เครื่องสามารถเข้าใจได้
   - ประมวลผลคำสั่ง
   - ส่งสัญญาณไปควบคุมการทำงานของคอมโพเนนท์ (Component) อื่นๆ

3.1.2 หลักการในการออกแบบวงจรลอจิก (Logic Design Methodologies)

แบ่งเป็นขั้นตอนหลักดังนี้

3.1.2.1 การกำหนดรายละเอียดระบบ(System Specification)

- ภาษา (language),กราฟ (graph)
- การกำหนดขั้นพฤติกรรม (Behavioral Specification)
- การกำหนดขั้นการทำงาน (Functional Specification)
- การกำหนดในขั้นโครงสร้าง (Structural Specification)

3.1.2.2 การแบ่งระบบให้เป็นสัดส่วน (System Partitioning)

3.1.2.3 การจำลองการทำงานของระบบ (Simulation)

- การจำลองการทำงานของหน้าที่(Functional Simulation)
- การจำลองการทำงานในขั้นรีจิสเตอร์ (Register Transfer Simulation)
- การจำลองการทำงานในระดับเกต (Gate Level Simulation)
- การจำลองการทำงานในความถูกต้องของระบบ (Fault Simulation)
- การวิเคราะห์วงจร (Circuit Analysis)
- การจำลองการทำงานของเครื่องมือ (Device Simulation)
- การจำลองการทำงานของโปรเซส (Process Simulation)

3.1.2.4 การที่สามารถทดสอบระบบได้ (Testability)

- การทดสอบโดยอัตโนมัติ (Automatic Test Generation)
- การวิเคราะห์การทดสอบ (Testability Analysis)
- ออกแบบระบบให้ทดสอบได้ (Testable Design Rule Enforcement)

3.1.2.5 การสังเคราะห์ระบบ (System Synthesis)

- การมินิไมส์ลอจิก(Logic minization)

3.1.2.6 การวางโครงสร้างระดับกายภาพ (Physical Layout)

- อินเตอร์แอคทีฟกราฟฟิคส์ (Interactive Graphics)
- โครงสร้างอัตโนมัติ (Automated Layout)

3.1.2.7 การตรวจสอบความถูกต้องของการออกแบบ(Design Verification)

- ตรวจสอบการออกแบบ (Design Rule Check (DRC))
- ตรวจสอบทางไฟฟ้า (Electrical Rule Check (ERC))
- ตรวจสอบการเชื่อมต่อ (Connectivity Check)
- ตรวจสอบเครื่องมือ (Device Recognition)
- เอาพารามิเตอร์ทางไฟฟ้าออก (Electrical Parameter Extraction)

3.1.2.8 การทำเอกสาร (Documentation)

3.1.2.9 การจัดการข้อมูล (Data Management)

- ฐานข้อมูล (Database)
- ระบบการจัดการระบบฐานข้อมูล (Database Management System)
- การจัดการคอนฟิกกิวเรชั่น (Configuration) ของข้อมูล (Data Configuration Management)
- การพัฒนาผลิตภัณฑ์ (Production Development)
- ระบบการจัดการข้อมูล (Management Information Systems)

3.1.3 <u>การกำหนดรายละเอียดของระบบ</u> (System Specifications)

3.1.3.1 การอธิบายรายละเอียดของภาษาที่ใช้ในการออกแบบ (Language Description)

การกำหนดรายละเอียดของระบบเป็นพื้นฐานของการออกแบบระบบเพื่อจะนำไปใช้งาน (Implement) สิ่งที่สำคัญที่สุดในการนำคอมพิวเตอร์มาช่วยออกแบบระบบคือ ต้องสามารถระบุปัญหาภายใต้สถานการณ์ต่างๆได้ ภาษาที่ใช้ออกแบบระบบดิจิตอลเรียกว่า <u>ฮาร์ดแวร์ เดสคริพชั่น แลงเกวช</u> (Hardware Description Language (HDL)) แบ่งได้เป็น 2 ประเภทใหญ่ดังนี้

3.1.3.1.1 <u>ภาษาระดับโครงสร้าง</u> (Structural Language) เป็นการระบุส่วนประกอบของลอจิก (Logic Elements)และการเชื่อมต่อของส่วนประกอบ

1. ภาษาที่เป็นข้อความ(Textural Description Language)
   - ระบุข้อมูลในการเชื่อมต่อของส่วนต่างๆในวงจรและพารานิเตอร์ที่เกี่ยวข้องกับส่วนต่างๆ
2. ภาษาที่เป็นรูปภาพ (Graphical Language)
   - แสดงการออกแบบด้วยรูปภาพ

3.1.3.1.2 <u>ภาษาระดับพฤติกรรม</u> (Behavioral languages) ระบุการทำงานของแต่ละส่วน (Element) ในวงจรและกำหนดรายละเอียดเกี่ยวกับไทม์มิ่ง(Timing)ของแต่ละส่วนในวงจร

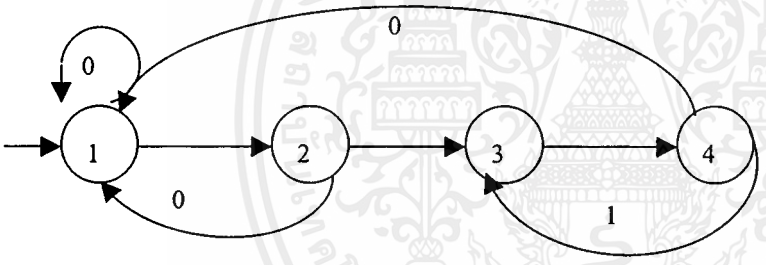
แบบที่เป็นกราฟ : สเตตไดอะแกรม (State Diagram),ลอจิกไดอะแกรม(Logic Diagram),วงจรลอจิก(Logic Schematics) ตัวอย่างเช่น

1. ไฟไนต์สเตตแมชีน (Finite State Machine (FSM)) ประกอบด้วยตารางของสเตต (State table), เรกูลาร์เอกซ์เพรสชั่น (Regular Expression), โฟลว์ชาร์ต(Flowchart) เช่น เรกูลาร์เอกซ์เพรสชั่นเป็น 11(01)* หมายความว่า เซตของอินพุทที่ขึ้นต้นด้วย 11 และประกอบด้วย 01 กี่ตัวก็ได้, S={ 11,1101,110101,... }

2. วิธีการใช้ทฤษฎีของกราฟ (Graph-theoretic methods) ยกตัวอย่างเช่น ทรานสิชั่น (Transition Graph), เพทรีเนทส์ (Petri Nets)

	0	1
A	B	C
B	C	D
C	A	D
D	D	C

ตาราง 3-1 ตารางสถานะ (State Table)



รูปภาพ 3-2 State Machine

3.1.3.2 การอธิบายในระดับพฤติกรรม (Behavioral Description) : การไหลของข้อมูล (Information Flow)

ระบบจะถูกมองเสมือนการเชื่อมต่อของฟังก์ชั่นแนลโมดูล(Dunctional Modules)ที่ถูกระบุโดยคุณสมบัติของอินพุทเอาท์พุท

3.1.3.3 การอธิบายในระดับฟังก์ชั่น (Functional Description) : การไหลของดาต้า (Data Flow)

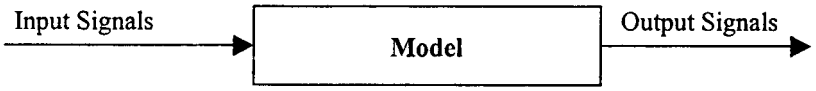
ระบบจะถูกแบ่งเป็นระบบย่อย(Subsystem)และอัลกอริทึมของลอจิกที่จะทำงานต่างๆจะอยู่ในรูปของการเคลื่อนย้ายสมการบูลีน (Boolean Equations), ตารางระบุสเตต (State Tables), ไทม์มิ่งไดอะแกรม(Timing Diagrams) และโฟลว์ชาร์ต(Flowcharts)

3.1.3.4 การอธิบายในระดับการโครงสร้าง : การนำไปใช้(Implementation)

จะอธิบายระบบในรายละเอียดของเกต(Gate), ไบสเตเบิล(Bistables) เป็นต้น

3.1.4 การจำลองการทำงานของระบบ (Simulation)

เป็นการใช้คอมพิวเตอร์มาช่วยทำนายผลของการทำงานและประสิทธิภาพในโมดูล ทำให้สามารถประหยัดเวลา และค่าใช้จ่ายในการออกแบบได้ เนื่องจากว่า เมื่อนำไปผลิตออกมาเป็นชิพ (Chip) ก็จะไม่ต้องแก้ใข เพราะสามารถมั่นใจได้ว่า การทำงานถูกต้องสมบูรณ์แล้ว จากการจำลองการทำงาน เพื่อตรวจสอบความถูกต้องที่ได้กระทำมา



รูปภาพ 3-3 ความสัมพันธ์ระหว่างอินพุทและเอาท์พุทซิกแนล

สัญญาณ(Signal) : สัญญาณไฟฟ้า – โวลท์เตจ, กระแสไฟฟ้า, ประจุไฟฟ้า เป็นต้น
ลอจิก – 0,1,X
ฟังก์ชั่นแนล(Functional) – คุณสมบัติ, แพคเกตของข้อมูล (Data Packet), เวคเตอร์ เป็นต้น

3.1.4.1 การจำลองการทำงานในระดับของระบบ(System level)

- เพื่อประเมินคุณสมบัติของระบบ
- เพื่อนำไปวิเคราะห์ไทม์มิ่ง (Timing Analysis)

3.1.4.2 การจำลองการถ่ายโอนข้อมูลระหว่างรีจิสเตอร์ (Register Transfer Level)

- เพื่อประเมินการออกแบบหน้าที่ของระบบ
- การไหลของข้อมูลจะถูกกำหนดในขั้นการถ่ายโอนข้อมูลระหว่างรีจิสเตอร์

3.1.4.3 การจำลองการทำงานในระดับเกตหรือลอจิก (Gates or Logic Level)

- เพื่อตรวจสอบความถูกต้องของลอจิก และไทม์มิ่ง
- ลอจิกเกตที่แท้จริง และการเชื่อมต่อ และหน้าที่จะถูกกำหนด
- จำกัดให้สัญญาณเป็นไบนารี (Binary Signal) เท่านั้น
- มีการกำหนดเกตดีเลย์(Gate-Propagation Delays)

3.1.4.4 การจำลองการทำงานในระดับวงจร(Circuit Level)

- เพื่อตรวจสอบประสิทธิภาพของวงจร
- ส่วนประกอบในวงจร (เช่นไดโอด, ทรานซิสเตอร์, รีจิสเตอร์)และการเชื่อมต่อของส่วนประกอบเหล่านั้นจะถูกกำหนดขึ้น
- จำกัดขนาดของโวลท์เตจ และกระแสไฟฟ้าที่เหมาะสม

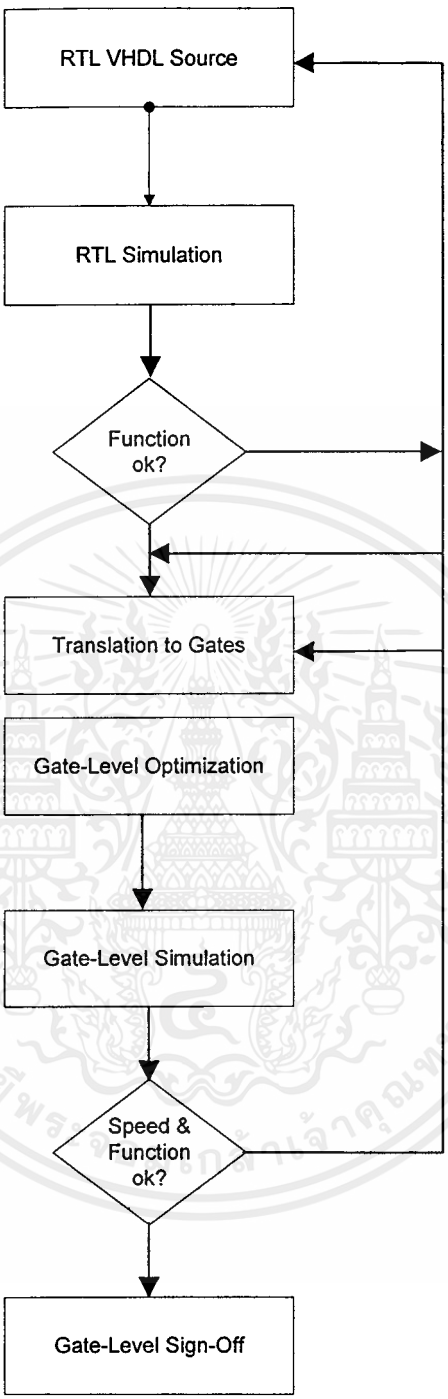
3.2 ขั้นตอนการออกแบบ

3.2.1 ขั้นตอนการเขียน RTL source code

จากการออกแบบที่ได้มาจะทำให้เราสามารถเขียนการทำงานของระบบได้โดยการมองให้เป็นลักษณะของคอมโพเนนท์ ซึ่งการเขียนนี้จะใช้การประกาศเป็นเอนติตี้ (Entity), การทำงานของแต่ละโปรเซส (Process) หรือการใช้แพคเกท (Package) โดยจะมีรูปแบบของการเขียนดังนี้

1. Package
2. การประกาศเรียกใช้Library
3. Entity
4. โครงสร้างการทำงานภายในซึ่งจะถูกเขียนภายใน Architecture

ในที่นี้ใช้โปรแกรม Code Wright ในการเขียนโปรแกรม เนื่องจากซอฟแวร์ตัวนี้มีความสามารถในการตรวจสอบรูปแบบการใช้งานของภาษาVHDLได้ นอกจากนี้ยังมีความสามารถในการเขียนโปรแกรมภาษาอื่นๆ เช่น แอสแซมบลี (Assembly) หรือภาษาซี (C)

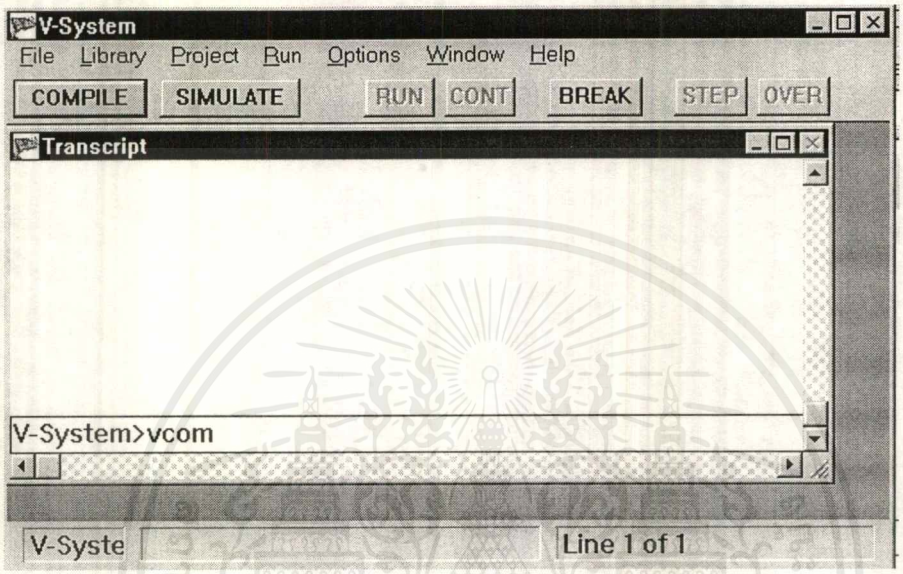


รูปภาพ 3-4 ขั้นตอนการออกแบบโดยใช้ HDL

3.2.2 ขั้นตอนในการ จำลองการทำงาน

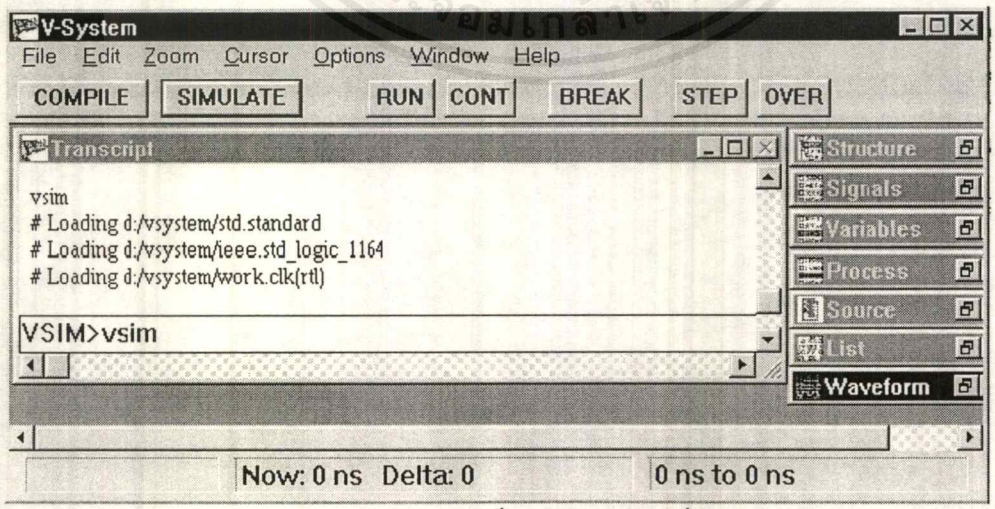
เมื่อทำการคอมไพล์ ด้วยโปรแกรมที่ชื่อว่า V-System แล้วก็จะเริ่มการตรวจสอบการทำงานของระบบ โดยใช้การจำลองการทำงาน บนโปรแกรม V-System ตัวเดียวกัน โดยมีขั้นตอนการทำงานดังนี้

1. เมื่อได้ตัวโปรแกรมที่ได้จากการเขียนเรียบร้อยแล้วจะทำการคอมไพล์ โดยการใช้คำสั่ง Vcom หรือเรียกใช้เมนูจากหน้าจอของโปรแกรม



รูปภาพ 3-5 การใช้งาน V-System ในการ Compile

2. เมื่อโปรแกรมที่อธิบายการทำงานของระบบไม่มีปัญหาหรือไม่มีข้อผิดพลาดในเรื่องของรูปแบบก็จะทำการ จำลองการทำงาน โดยการเรียกใช้ Vsim ซึ่งจะเกิดหน้าจอเพื่อให้เลือกเอาตัวเอนติตี้ (Entity) ที่เราต้องการในการจะจำลองการทำงาน มาดังนี้



รูปภาพ 3-6 ผลที่ได้จากการใช้คำสั่ง Vsim

3. จากนั้นก็ทำการป้อนค่าสัญญาณต่างๆที่ต้องการแล้วดูผลลัพธ์ที่ได้ว่าตรงกับความต้องการและการทำงานของระบบจริงหรือไม่ โดยการป้อนค่าจะเขียนโปรแกรม VHDL ขึ้นมาเพื่อทำงานแทนการป้อนค่าถ้าหากมีข้อผิดพลาดต้องทำการเปลี่ยนแปลงโปรแกรมใหม่ แล้วเริ่มการคอมไพล์ใหม่อีกครั้งจนกว่าจะได้ค่าของสัญญาณตามที่ต้องการ

4. ทำการทดสอบการทำงานของทุกคำสั่งที่มีการใช้งานใน ARM7 โดยการเขียนโปรแกรมทดสอบให้ครอบคลุมทุกคำสั่งที่มีการใช้งานเพื่อตรวจสอบให้ได้มากกรณีที่เป็นไปได้ของคำสั่งทั้งหมด ในที่นี้ทางกลุ่มได้ทดสอบการทำงานของทุกคำสั่งโดยจะแสดงผลลัพธ์ของสัญญาณที่เกิดขึ้นในแต่ละคำสั่งในบทต่อไป

3.3 การสร้าง Test Bench

Test bench ใช้ตรวจสอบความถูกต้องในการทำงานของระบบที่สร้างโดย VHDL การทดสอบ VHDL code ที่เขียนขึ้นโดยทั่วไปมี 2 วิธีคือ

1. ทำการจำลองการทำงานโดยการกำหนดค่า

วิธีนี้ต้องมีการกำหนดค่าให้กับสัญญาณ (force input signal) ทุกครั้งมีการทำการจำลองการทำงาน ซึ่งในบาง tool อนุญาตให้สร้าง macro file เพื่อเก็บรวบรวมคำสั่งที่ใช้ในการจำลองการทำงานทำให้สามารถเก็บรูปแบบของ input ที่ซ้ำกันไว้ได้ และในบาง tool อนุญาตให้ทำการ test pattern file เพื่อนำมาใช้ประกอบการจำลองการทำงานได้ แต่มักเป็น tool ราคาแพง

การใช้สิ่งที่ tool มีให้ในการจำลองการทำงานมีข้อเสียคือ ต้องยึดติดกับ tool นั้นๆเกินไปเมื่อต้องการ test ด้วย tool อื่น อาจต้องแก้ไขใหม่เนื่องจาก แต่ละ tool มักมีความแตกต่างกัน

2. ทำการจำลองการทำงานด้วย VHDL code ที่เขียนขึ้นเพื่อทดสอบโดยเฉพาะ

สิ่งที่ทุกๆ tool มีเหมือนกันคือ ต้องรับ VHDL code ได้ดังนั้น test program ที่สร้างขึ้นจะมองระบบที่สร้างขึ้นเป็นคอมโพเนนท์หนึ่งในโปรแกรมและหน้าที่ของโปรแกรมคือการป้อน test pattern ไปยังคอมโพเนนท์ที่ทำการทดสอบแล้วตรวจสอบ output ที่ออกมาว่ามีค่าตรงกันที่ควรจะเป็นหรือไม่

3.3.1 รูปแบบของ test program โดยทั่วไป

3.3.1.1 Entity

Test program ไม่จำเป็นต้องมี I/O port ดังนั้น entity จึงมีลักษณะดังนี้

```
Entity testbench is
End ;
```

3.3.1.2 Architecture

3.3.1.3 Declarative region

3.3.1.4 คอมโพเนนท์ ที่ต้องการ test เช่น

```
Component
    port(clk        : in std_ulogic;
         mul_in1    : in std_logic_vector(31 downto 0);
```

```
        mul_in2     : in std_logic_vector(31 downto 0);
        work        : in std_ulogic;
        done        : out std_ulogic;
        mul_out     : out std_logic_vector(31 downto 0));
    end component;
```

3.3.2 Input/Output Signal ที่เกี่ยวข้อง

เป็นสัญญาณต่างๆ ที่ต้องการใช้กับคอมโพเนนท์ที่ต้องการทดสอบสำหรับคอมโพเนนท์ ใน 3.3.2 จะมี signal ดังนี้

```
signal clk          : std_ulogic := '0';
signal mul_in1      : std_logic_vector(31 downto 0);
signal mul_in2      : std_logic_vector(31 downto 0);
signal work         : std_ulogic;
signal done         : std_ulogic;
signal mul_out      : std_logic_vector(31 downto 0);
```

3.3.3 โครงสร้างข้อมูลของ Test pattern

ใช้ชนิด (type) ข้อมูลแบบ record ของ VHDL ในการเก็บค่าสัญญาณที่ใช้ใน test pattern สำหรับ คอมโพเนนท์ ในข้อ 3.3.2 จะได้ record ดังนี้

```
type test_record is record
   mul_in1      : std_logic_vector(31 downto 0);
   mul_in2      : std_logic_vector(31 downto 0);
   mul_out   : std_logic_vector(31 downto 0);
end record;
```

3.3.4 Test Pattern

ทำได้โดยการให้ค่าเริ่มต้น (initial value) กับสัญญาณต่างๆ เป็นแอเรย์ (Array)ใน record สำหรับ record ในข้อ 3.3.3 จะได้ test pattern ดังนี้

```
type test_array is array(positive range<>) of test_record;
constant test_pattern : test_array := (
(mul_in1 => "00000000000000000000000000000000",
 mul_in2 => "00000000000000000000000000000000",
 mul_out => "00000000000000000000000000000000"),
(mul_in1 => "00000000000000111111111111000000",
 mul_in2 => "00000000000000000000000000000000",
```

```
mul_out => "00000000000000000000000000000000"),
(mul_in1 => "00000000000000000000000000000000",
 mul_in2 => "00000110000000000011000001100001",
 mul_out => "00000000000000000000000000000000"),
 mul_in1 => "00000000000000000000000000000000",
 mul_in2 => "00000000000000000000000000000000",
 mul_out => "00000000000000000000000000000000"),
(mul_in1 => "00000000000000000000000000000000",
 mul_in2 => "00000000000000000000000000000000",
 mul_out => "00000000000000000000000000000000"));
```

3.3.5 Architecture contents

3.3.5.1 Instantiate component ดังนี้

```
use_m:mul port map(clk => clk,
                   mul_in1 => mul_in1,
                   mul_in2 => mul_in2,
                   work => work,
                   done => done,
                   mul_out => mul_out);
```

3.3.5.2 Testing process

เนื่องจาก test pattern เป็นแอเรย์ของ record ซึ่งการป้อนค่าจาก Test pattern ไปยัง คอมโพเนนท์ ทำได้โดยใช้ for loop ซึ่งใน for loop นี้ เมื่อป้อนค่าแล้ว หลังจากรอ output ออกมาจาก คอมโพเนนท์ แล้วก็ทำการตรวจสอบจาก output pattern ที่ควรเป็น การรายงานความถูกต้องของผลลัพธ์ ทำได้ด้วย Function assert ของ VHDL ดังตัวอย่าง

```
test : process
    variable vector : test_record;
    variable ferror : boolean := false;
    begin
        for i in test_pattern'range loop
        work <= '1';
        vector := test_pattern(i);
-- apply the simuls
        mul_in1 <= vector.mul_in1;
        mul_in2 <= vector.mul_in2;
-- wait for the outputs to settle
```

```
                wait until done = '1';
        -- check the results
                if (mul_out /= vector.mul_out) then
                    assert false
                     report "Result not true.";
                     ferror := true;
                end if;
                wait for 10 ns;
                  work <= '0';
                  wait for 20 ns;
            end loop;
            assert not ferror
             report "There were ERRORS in the test."
            severity note;
            assert ferror
            report "Test complete with no errors."
            severity note;
            wait;
end process;
```

บทที่ 4
การออกแบบระบบดิจิตอลเลียนแบบการทำงานของ ARM 7

ในบทนี้ได้อธิบายถึงการทำงานต่างๆ ที่ได้ทำในโครงงานนี้ รายละเอียดของแต่ละคอมโพเนนท์ ที่ได้ทำการศึกษา เพื่อให้เข้าใจถึงการทำงานของแต่ละคอมโพเนนท์ รวมทั้งแนวทางการทดสอบที่โครงงานนี้ได้ทำการกำหนดขึ้นมา เพื่อทดสอบการทำงานของไมโครโปรเซสเซอร์ที่ได้ออกแบบมา

4.1 ขั้นตอนในการออกแบบ ARM 7 ไมโครโปรเซสเซอร์

- Preliminary Design
  - Development Plan วางแผนในการออกแบบ
  - Design Partition แบ่งการออกแบบเป็นส่วนๆ
  - Design Specification ระบุรายละเอียดของที่จะออกแบบ(ARM7)

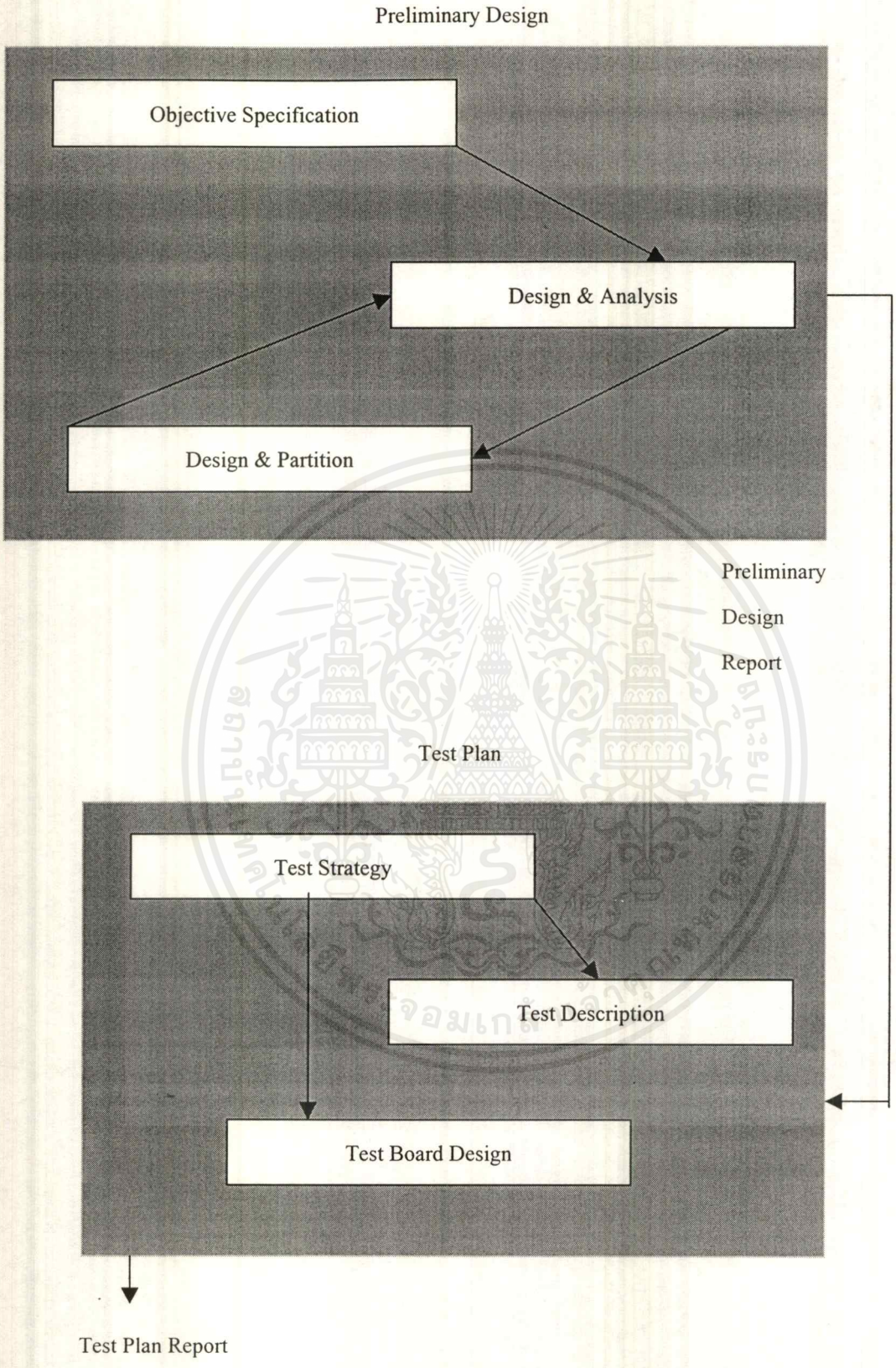
- Test Plan
  - Test Strategy วางแผนการตรวจสอบ

- RTL Block Coding
  - เขียน code ภาษา VHDL ของแต่ละ block ที่ได้ทำการแบ่งเอาไว้ก่อนหน้านี้

- Test Program Design
  - เขียน Code ภาษา VHDL ตรวจสอบแต่ละ Block และ Subsystem

- Subsystem Simulation
  - ทำการ Simulate และตรวจทุก Block ว่าทำงานได้อย่างถูกต้อง

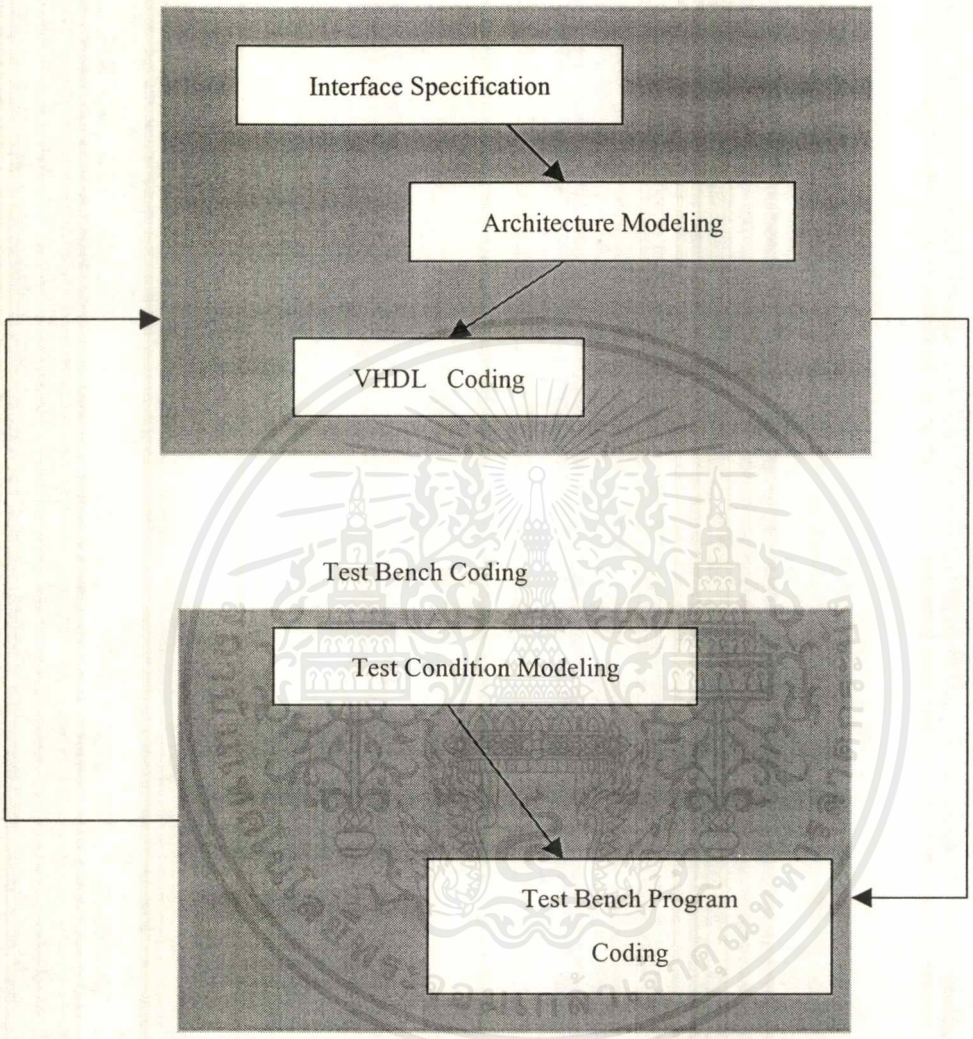
- Project Development Process
  - Preliminary Design
    - Objective Specification — ระบุจุดหมายของโครงงาน
    - Design & Analysis — วิเคราะห์และวางแผนการออกแบบ
    - Design Partition — แบ่งส่วนในการออกแบบออกเป็นส่วนๆ
  - Test Plan
    - Test Strategy — วางแผนการตรวจสอบ
    - Test Description — อธิบายวิธีการตรวจสอบ



รูปภาพ 4-1 Project Development process

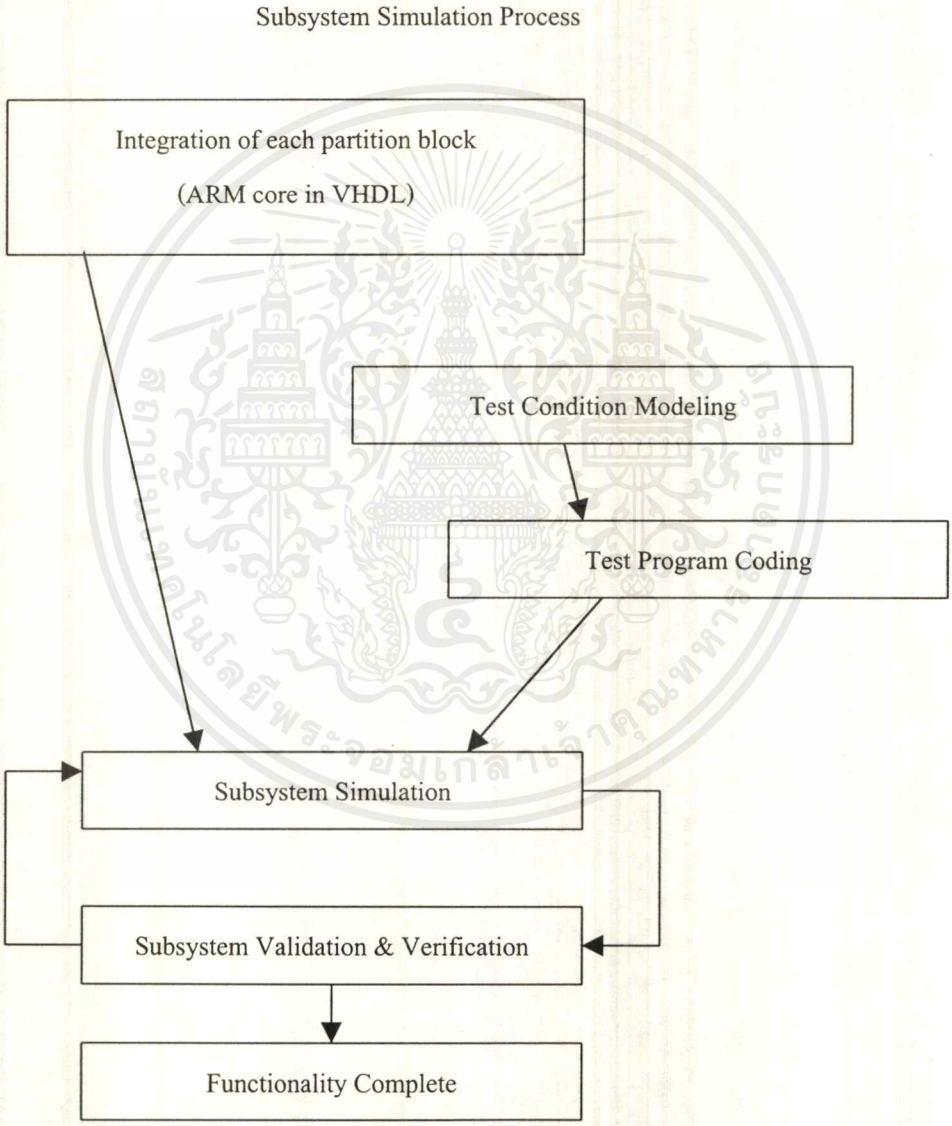
Block Simulation Process

RTL Block Coding



รูปภาพ 4-2 Block Simulation Process

- RTL Block Coding
  - Interface Specification ระบุความต้องการของการเชื่อมต่อ
  - Architecture Modeling วิเคราะห์ และออกแบบแต่ละ block ในระดับ Architecture
  - VHDL Coding เขียน Code ภาษา VHDL
- Test Bench Coding
  - Test Condition Modeling ออกแบบสถานการณ์ในการตรวจสอบในลักษณะต่างๆ
  - Test Bench Program Coding เขียน Code ด้วยภาษา VHDL เพื่อใช้ในการตรวจสอบ



รูปภาพ 4-3 Subsystem Simulation Process

4.2 การสร้างระบบดิจิตอลเลียนแบบการทำงานของ ARM 7 โดยใช้ VHDL

ขั้นตอนในการออกแบบระบบนี้จะเหมือนกับการออกแบบโดยทั่วไป ซึ่งกล่าวไว้ในบทที่2 แต่ส่วนสำคัญที่ต่างกันก็คือ " ไม่ได้สร้างระบบขึ้นมาใหม่แต่สร้างระบบที่มีการทำงานเหมือนกับระบบเก่าให้มากที่สุด " จาก ดาต้าชีท (Data Sheet) ต้องทำการศึกษาการทำงานของคำสั่ง (Instruction Set) และไทม์มิ่งของคำสั่ง(Instruction Cycle) ให้เข้าใจการทำงานแล้วจึงนำมาออกแบบโครงสร้างของระบบของที่จะทำการออกแบบ จากนั้นจึงเขียนไทม์มิ่ง ขึ้นมาเองใหม่อีกครั้งแล้วทำการเขียน Code ในแต่ละส่วนการทำงานด้วยภาษา VHDL แล้วทำการทดสอบการทำงานโดยใช้ V-System ในการตรวจสอบ

ขั้นตอนการออกแบบระบบ

จากการวิเคราะห์การทำงานของ ARM 7 พบว่า Instruction Cycles แบ่งออกได้เป็น 3 สเตตๆ ละ 2 เฟส คือ

1. สเตตที่ 1 : Instruction Fetch (IF)

   การทำงาน :

   นำ คำสั่งจากหน่วยความจำเข้ามารอทำการถอดรหัสเพื่อใช้ในการทำงานต่อไป

2. สเตตที่ 2 : Instruction Decode(ID)

   การทำงาน :

   ถอดรหัสคำสั่งที่เก็บใน IR ออกมาทำการถอดรหัสคำสั่งเพื่อให้สามารถส่งสัญญาณควบคุมส่วนต่างๆได้เช่นการควบคุม MUX, ALU, MULTIPLY, SHIFTER เตรียมค่าโอเปอร์แลนด์เพื่อนำไปใช้ในการทำงานต่อไป ส่วนนี้จะเป็นการแยกให้ทราบว่าอยู่ในชุดคำสั่งใดทำให้สามารถทำการเอ็กซิคิวส์ได้ถูกต้องในสเตตต่อไป

3. สเตตที่ 3 : Instruction Execute (IE)

   การทำงาน :

   แยกการทำงานของแต่ละคำสั่งอย่างชัดเจนซึ่งจะมีจำนวนรอบของเวลา (Cycle Time) ในการทำงานในแต่ละคำสั่งไม่เท่ากัน แล้วก็ทำการเขียนผลลัพธ์ลงสู่ที่หมายปลายทางที่ต้องการในขอบขาขึ้นของเฟสที่ 2 ทั้งนี้ข้อควรระวังตรงที่จะเกิดปัญหาเนื่องมาจากที่นำค่าของรีจิสเตอร์ที่ไม่เป็นค่าล่าสุดไปใช้ ทำให้การทำงานเกิดการผิดพลาด จึงมีการทำ Forwarding นั้นคือนำค่าที่ได้จากการทำงานในคำสั่งก่อนหน้านั้นไปใช้ทันทีก่อนที่จะมีการเขียนคำสั่งลงรีจิสเตอร์จริงๆ ข้อมูลจึงไม่ผิดพลาด

รายละเอียดของคำสั่งต่างๆ

คำสั่ง	Source 1	Source 2	Shift	Execute	Write-back
B	-	-	-	-	#Jump>Pc
BL	-	-	-	-	#Jump>Pc
AND	W	W	OP2	SHIFT,AND	W
EOR	W	W	OP2	SHIFT,XOR	W
SUB	W	W	OP2	SHIFT,SUB	W
RSB	W	W	OP2	SHIFT,RSB	W
ADD	W	W	OP2	SHIFT,ADD	W
ADC	W	W	OP2	SHIFT,ADC	W
SBC	W	W	OP2	SHIFT,SBC	W
RSC	W	W	OP2	SHIFT,RSC	W
TST	-	-	OP2	SHIFT,TST	-
TEQ	-	-	OP2	SHIFT,TEQ	-
CMP	-	-	OP2	SHIFT,CMP	-
CMN	-	-	OP2	SHIFT,CMN	-
ORR	W	W	OP2	SHIFT,ORR	W
MOV	-	W	OP2	SHIFT,MOV	W
BIC	W	W	OP2	SHIFT,OP2 AND NOT OP1	W
MVN	W	W	OP2	SHIFT,NOT OP2	W
MUL	W	W	-	MUL	W
MLA	W	W	-	MUL,ADD	W
LDR	W,B	W,B	OP2	SHIFT,LOAD	W,B
STR	W,B	W,B	OP2	SHIFT,STORE	W,B
SWP	W,B	W,B	OP2	LOAD,STORE	W,B

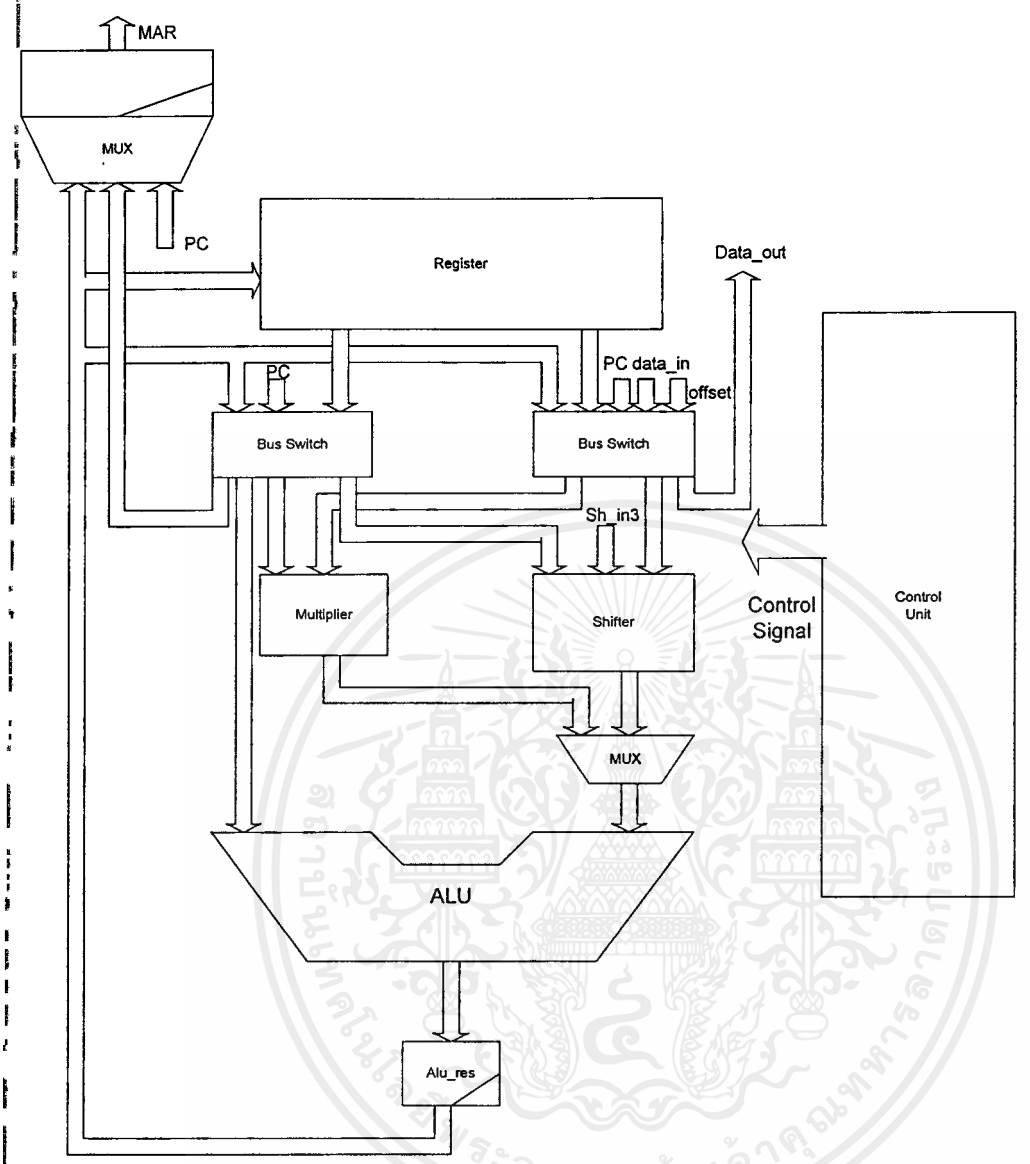
ตาราง 4-1 รายละเอียดของคำสั่งต่างๆ

คำสั่งสามารถทำการ shift ไปพร้อมกันกับทำคำสั่งปกติได้

หลังจากขั้นตอนนี้ทำให้สามารถมีข้อมูลเพียงพอที่จะใช้ในการออกแบบแล้ว ซึ่งจะทำการออกแบบแบบ top-down นั่นคือทำการแบ่งระบบออกเป็นระบบย่อยๆ (System Partition) ซึ่งต้องวิเคราะห์โครงสร้างโดยรวมของ ARM 7 ก่อนซึ่งแบ่งออกเป็นส่วนต่างๆได้ดังนี้

1. Arithmetic & Logic (ALU)
2. Register File Block
3. Multiplier
   - ทำการคูณ
4. Shifter Block
   - Shift ข้อมูลตามจำนวนที่ต้องการหรือผ่านค่าออกมาธรรมดา
5. Control Unit
   - Program Counter
   - Bus Switch and Multiplexer
   - Clock generator Unit
   - Instruction Decoder Unit

เมื่อรวมทั้ง 5 คอมโพเนนท์ เข้าด้วยกันแล้วจะได้เป็น ARM 7 ที่สร้างมี Data Pathดังรูป



รูปภาพ 4-4 Data Path ของARM 7

เพื่อให้ง่ายต่อการทำงานความเข้าใจ จึงแบ่งขั้นตอนการออกแบบออกเป็นดังนี้

4.2.1 สร้างโปรแกรมมิ่งโมเดล (Programming Model)

Programming Model ของแต่ละ block ได้จากการวิเคราะห์การทำงานของคำสั่งต่างๆที่มีผลต่อ block นั้นโดยมองในแง่ของการเขียนโปรแกรมเป็นหลัก โดยสิ่งที่เกี่ยวข้องกับการเขียนโปรแกรมจะเป็นส่วนประกอบในโปรแกรมมิ่งโมเดล

4.2.2 อธิบายการทำงานของบล็อค

4.2.2.1 Register File

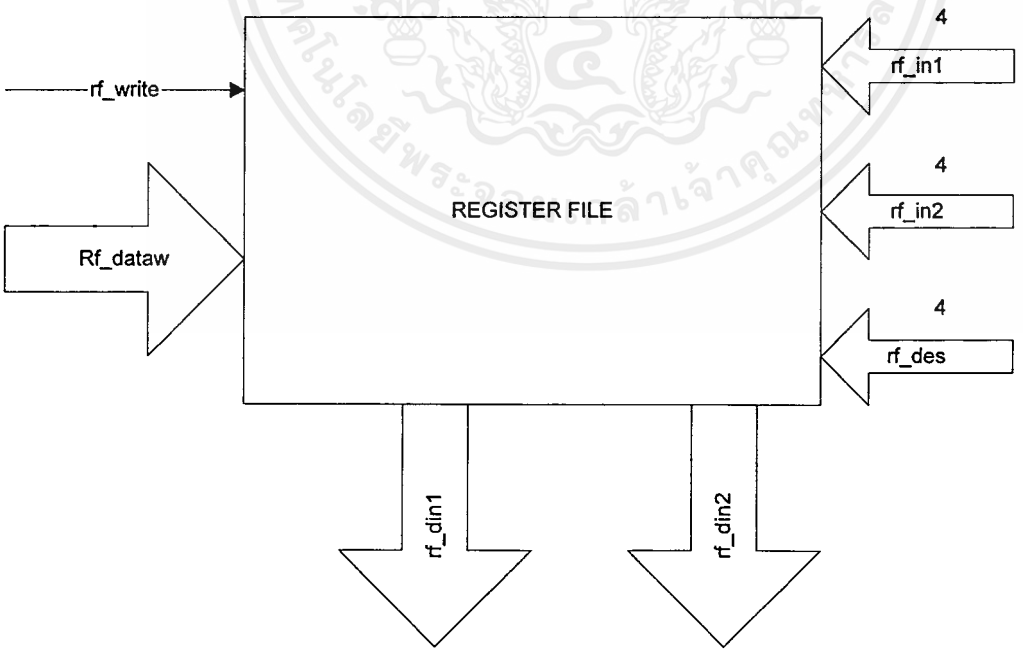
Register File ประกอบด้วย รีจิสเตอร์ทั่วไป (General Register) ขนาด 32 บิต จำนวน 16 ตัว โดยที่

R13 จะเป็นตัวที่เก็บค่าของสแตกพอยเตอร์ (Stack Pointer)

R14 จะเป็นตัวที่ใช้เก็บค่าของโปรแกรมเคาน์เตอร์(Program Counter: PC) ของคำสั่งที่จะกลับมาทำอีกครั้งเมื่อเสร็จสิ้นการทำงานทั้งหมดที่ได้ทำการกระโดดไปทำงานแล้ว

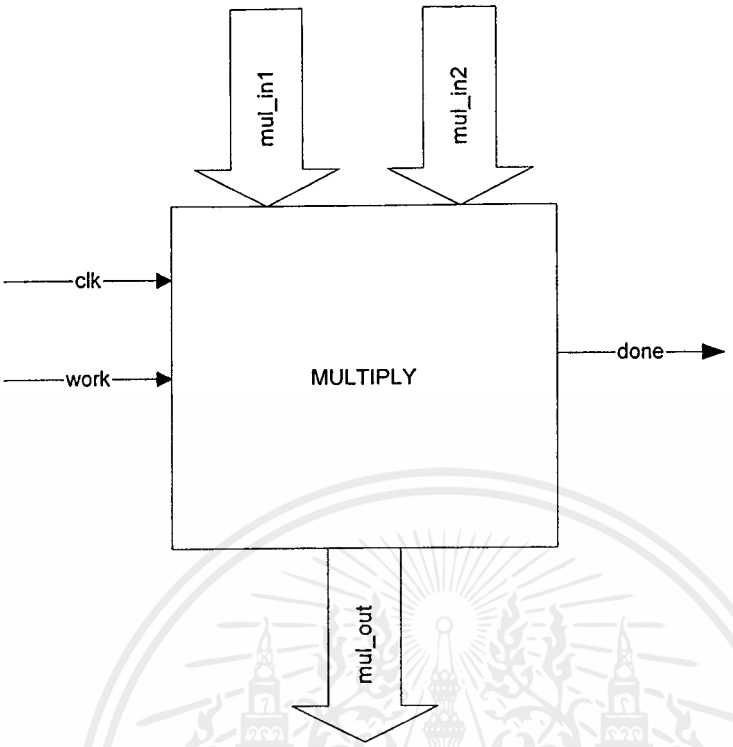
R15 จะเป็นตัวที่ใช้ในการเก็บค่า PC จึงไม่นิยมนำมาใช้เป็นโอเปอร์แลนด์ แต่ก็สามารถที่จะนำมาใช้งานได้ในบางคำสั่ง

ขาที่ใช้งานใน Register File จะมี port ของการ Read 2 port ส่วนการ write จะมี 1 port และมีสัญญาณควบคุมการทำงานโดยตรวจจากสัญญาณการ read, write เพื่อให้ทราบว่าเป็นการอ่านหรือเขียนค่าใน Register File



รูปภาพ 4-5 Register File

4.2.2.2 Multiplier



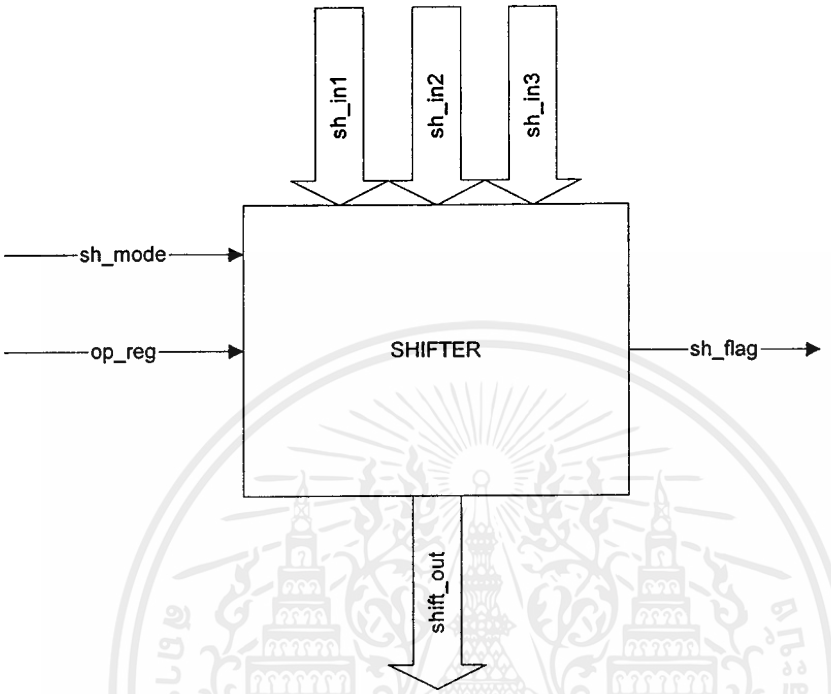
รูปภาพ 4-6 Multiplier

เป็นส่วนเพิ่มเติมของ ARM7 ที่พิเศษจากตัวอื่นๆทั่วไป เพราะจะเป็นตัวทำการคูณแยกออกจาก ALU ผลคูณที่ได้จะถูกส่งเข้า ALU เพื่อที่จะส่งผลคูณที่ได้ไปเก็บยังรีจิสเตอร์ที่ต้องการ (คำสั่ง MUL) หรือทำการบวกค่าผลคูณที่ได้กับค่าในรีจิสเตอร์อีกตัว (คำสั่ง MLA) Control Unit จะต้องส่งสัญญาณมาบอกให้ตัว Multiplier เริ่มทำงาน และจะต้องรอจนกระทั่งตัว Multiplier ส่งสัญญาณมาบอกว่าได้ทำงานเรียบร้อยแล้ว จึงจะทำการส่งสัญญาณไปบอกตัวอื่นๆ ที่เกี่ยวข้องให้เริ่มทำงาน โดยที่ ALU จะใช้โอเปอร์แลนด์ ตัวหนึ่งเป็นผลลัพธ์ของการคูณ และ โอเปอร์แลนด์อีกตัวอาจจะมีหรือไม่มีก็ได้ (ถ้าไม่มีก็เป็นการส่งค่าผลลัพธ์ที่ได้ไปเก็บยังรีจิสเตอร์เลย แต่ถ้ามีก็จะบวกค่าผลลัพธ์ที่ได้กับค่าในรีจิสเตอร์อีกตัว)

ทั้งนี้ผลลัพธ์ที่ได้จากการคูณนั้นจะมีขนาดเพียง 32 บิต ดังนั้นหากการคูณในครั้งนั้นขนาดของ โอเปอร์แลนด์ 2 ตัวที่นำมาคูณกันนั้นมีขนาดรวมกันเกิน 32 บิต ผลลัพธ์ที่ได้ก็จะผิดพลาด ซึ่งผู้เขียนโปรแกรมก็จะต้องยอมรับความผิดพลาดในครั้งนี้ แต่ทั้งนี้ข้อผิดพลาดนี้อาจจะไม่เกิดขึ้นหากตัวคอมไพล์เลอร์ของ ARM7 มีการตรวจสอบในส่วนนี้ให้ก็จะมีการแจ้งข้อผิดพลาด (Error) ให้ทราบทันทีว่าค่าของผลลัพธ์การคูณนั้นจะเกิน 32 บิต เพราะโอเปอร์แลนด์ 2 ตัวรวมกันมีขนาดเกิน 32 บิตนั่นเอง และโอเปอร์แลนด์ทั้ง 2 ตัวนั้นจะเป็นได้เฉพาะ ค่าจาก รีจิสเตอร์ เท่านั้น

4.2.2.3 Shifter

ส่วนพิเศษเพิ่มเติมของ ARM7 เช่นกัน เนื่องจากสามารถแยกการทำ Shift หรือ Rotate ออกมาจากALU ซึ่งก็มีผลทำให้สามารถสั่ง Shift หรือ Rotate และทำงานคำสั่งทาง Logical หรือ Arithmetic ได้ภายในคำสั่งเดียวเท่านั้น



รูปภาพ 4-7 Shifter

Shifter มีโหมดการทำงานให้เลือก 4 โหมดด้วยกันดังนี้

1. Logical Shift Left (LSL) เป็นการทำ Shift ไปทางซ้าย โดยหากการ Shift นั้นเกิน 32 ครั้ง ผลลัพธ์ที่ได้ก็จะเป็นศูนย์
2. Logical Shift Right (LSR) เป็นการทำ Shift ไปทางขวา โดยหากจำนวนครั้งเกิน 32 ครั้งก็จะได้ค่าเป็นศูนย์เช่นกัน
3. Arithmetic Shift Right (ASR) เป็นการทำ Shift ไปทางขวา โดยต่างจาก LSR อยู่ตรงที่บิตทางซ้ายแทนที่จะถูกเติมด้วยศูนย์ก็จะเติมด้วยค่าของบิตซ้ายสุดของโอเปอร์แลนด์นั้นก่อนที่จะถูก Shift
4. Rotate Right (ROR) เป็นการทำ Rotate ไปทางขวา โดยหากจำนวนครั้งเกิน 32 ก็จะทำเป็นจำนวนครั้งเท่ากับเศษที่เหลือจากการหารด้วย 32 แล้ว

โอเปอร์แลนด์ที่ส่งให้กับ Shifter นั้นสามารถที่จะเป็นไปได้ 3 กรณีดังนี้

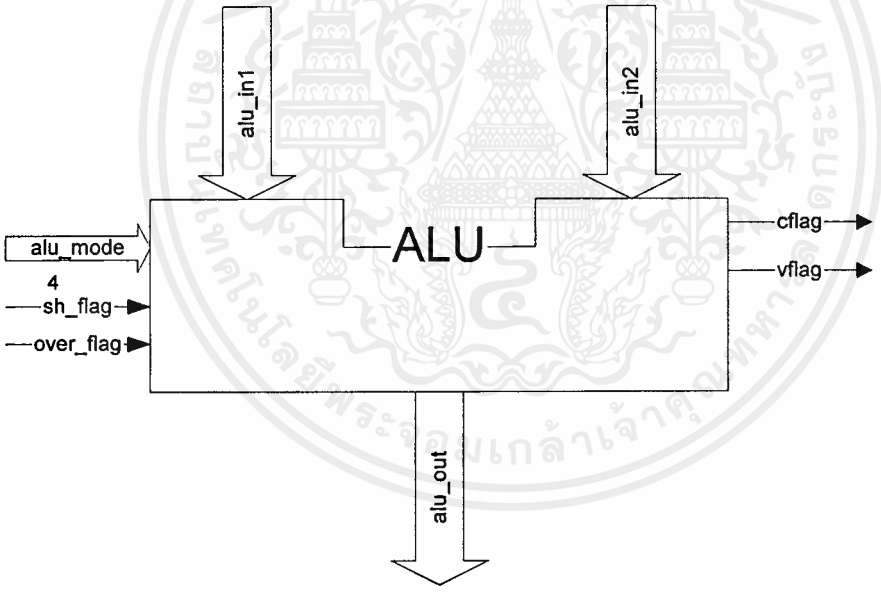
1. ตัวที่ถูกทำ และค่าที่บอกการจำนวนครั้งในการทำงานนั้น เป็นค่าที่ได้มาจากรีจิสเตอร์ทั้งคู่ โดยที่จำนวนครั้งจะใช้ค่าของ 8 บิตล่างเท่านั้น
2. ตัวที่ถูกทำเป็นค่าของรีจิสเตอร์ และจำนวนครั้งในการทำงานนั้นได้จากค่าคงที่ (Immediate Value) ซึ่งมีขนาด 5 บิต

3. ตัวที่ถูกทำ และจำนวนครั้งในการทำงานนั้นได้มาจากค่าคงที่ทั้งคู่ โดยที่จำนวนครั้งในการทำงานนั้นจะมีขนาด 5 บิต แต่เป็นค่าที่ได้มาจากการทำชิพไปทางซ้ายมา 1 ครั้งแล้ว โดยที่หากเป็นกรณีนี้จะเป็นการทำ Rotate Right เท่านั้น

ดังนั้นเราจึงจำเป็นต้องมี 2 บิตในการบอกโหมดการทำงาน และอีก 2 บิตในการบอกว่าจะใช้โอเปอร์แลนด์ชุดใด โดยที่ Shifter นั้นสามารถที่จะทำหน้าที่เป็นเพียงตัวผ่านค่าไปให้ ALU เท่านั้นก็ได้ ซึ่ง 2 บิตที่ใช้เลือกโอเปอร์แลนด์ ก็จะเป็นตัวบอกว่าเป็นเพียงการผ่านค่าด้วย

ARM7 นั้นสามารถที่จะทำการ Shift ค่าของโอเปอร์แลนด์ และทำ Data Processing ได้ภายในคำสั่งเดียว ดังนั้นสามารถที่จะเกิดการที่จะต้องใช้รีจิสเตอร์ ถึง 3 ตัวเป็นโอเปอร์แลนด์ แต่เนื่องจากว่า Register File นั้นมีเพียง 2 อินพุทเท่านั้น จึงจำเป็นที่จะต้องทำการยืดไซเคิล (Cycle) เพื่อเอ็กซิคิวส์คำสั่งออกไปอีก 1 ไซเคิล ซึ่งหากต้องการรายละเอียดของการทำงานในโหมดต่างๆสามารถอ่านได้จากภาคผนวก

4.2.2.4 ALU



รูปภาพ 4-8 ALU

จะมีการทำงานของส่วนของ Arithmetic และ Logical ที่สามารถ ถอดรหัส ได้จาก opcode

ที่อยู่ในคำสั่งที่นำเข้ามาทำงานในตอนนั้น โดยที่ ALU จะทำการส่งค่าของ Carry Flag และ Overflow Flag ที่ได้จากการทำงานออกมาให้ยัง Control Unit นำไปทำการกำหนดค่าของ Program Status Flag ด้วย ALU มีโหมดการทำงานทั้งสิ้น 16 โหมดดังนี้

AND => Rd:= Op1 and Op2

EOR => Rd:= Op1 EOR Op2 (xor)

SUB => Rd:= Op1 - Op2

RSB => Rd:= Op2 - Op1

ADD => Rd:= Op1 + Op2

ADC => Rd:= Op1 + Op2 + C

SBC => Rd:= Op1 – Op2 + C – 1

RSC => Rd:= Op2 – Op1 + C –1

TST => set condition on Op1 and Op2

TEQ => set condition on Op1 EOR Op2

CMP => set condition on Op1 – Op2

CMN => set condition on Op1 + Op2

ORR => Rd:= Op1 or Op2

MOV => Rd:=Op2

BIC => Rd:=Op1 and not Op2

MVN => Rd:=not Op2

4.2.2.5 Control Unit

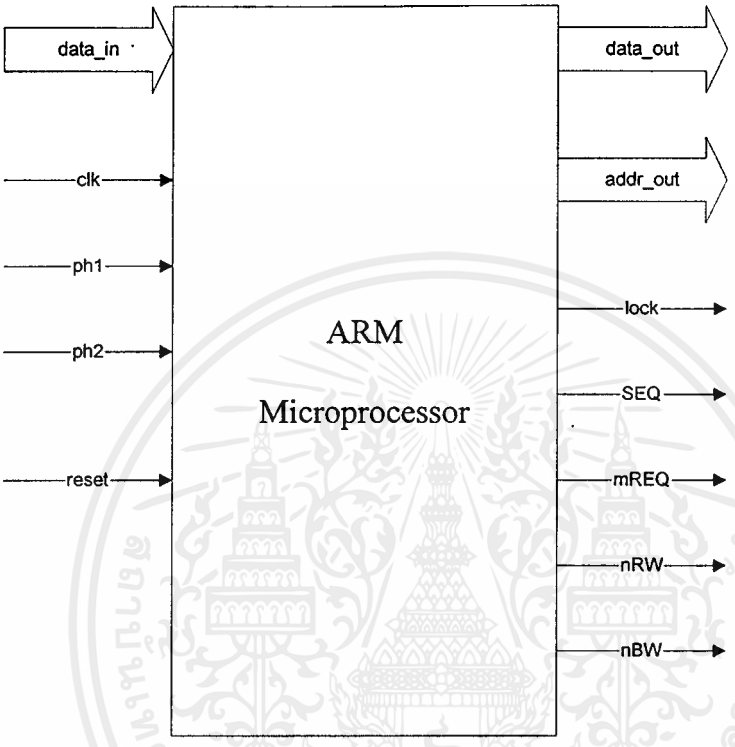
ส่วนนี้จะเป็นตัวที่ได้ทำการรวมแต่ละคอมโพเนนท์เข้ามาไว้ด้วยกันแล้ว โดยที่จะทำการควบคุมการทำงานของทั้งหมดให้สามารถได้ผลลัพธ์ตามที่ต้องการ ด้วยการทำการแปลคำสั่งที่ได้รับเข้ามาว่าเป็นคำสั่งอะไร และจะต้องมีการทำงานอย่างไร จากนั้นจึงส่งสัญญาณไปควบคุมการทำงานคอมโพเนนท์อื่นๆ ซึ่งเป็นส่วนที่มีความสำคัญเนื่องจากการทำงานทั้งหมดจะขึ้นอยู่กับส่วนนี้ ไม่ว่าจะเป็นการควบคุมไปป์ไลน์ หรือสเตตการทำงานทั้งหมดที่มีขึ้นในระบบ การทำงานภายในจะมีการส่งสัญญาณไปควบคุมการทำงานของ ALU, Register File, Shifter, Multiplier และMultiplexer ต่างๆ

ค่าที่เข้ามาเป็นโอเปอร์แลนด์ตัวหนึ่งนั้นสามารถนำมาจาก Multiplier หรือจากการอ่านค่าของรีจิสเตอร์ผ่านเข้าไปยัง Shifter แล้วส่งผ่านออกมายัง ALU จึงต้องมีการใช้ Multiplexerในการเลือกว่าจะให้ข้อมูลตัวใดเข้าสู่ ALU โดยที่ Multiplexer อีกตัวมีไว้สำหรับเลือกว่าจะใช้ค่าใดส่งไปยังตัวที่ใช้อ้างอิงตำแหน่งในหน่วยความจำ (Memory Address) เนื่องจากสามารถที่จะใช้ค่าผลลัพธ์ที่ได้จาก ALU ก็ได้ในกรณีที่เป็นการอ้างอิงหน่วยความจำแบบพรีอินเด็กซ์ (Pre-indexing) หรือจากรีจิสเตอร์ในกรณีที่เป็นการทำคำสั่ง Store หรือจากค่าของโปรแกรมเคาน์เตอร์ (PC)

Bus Switch ทั้ง 2 ข้างนั้นเป็นตัวเลือกว่าจะใช้ค่าทางด้านอินพุทตัวใด และจะส่งค่าที่เลือกนั้นออกไปให้คอมโพเนนท์ใด ทั้งนี้ทั้ง 2 ตัวจะทำงานตามสัญญาณคล็อกที่เข้ามาด้วย

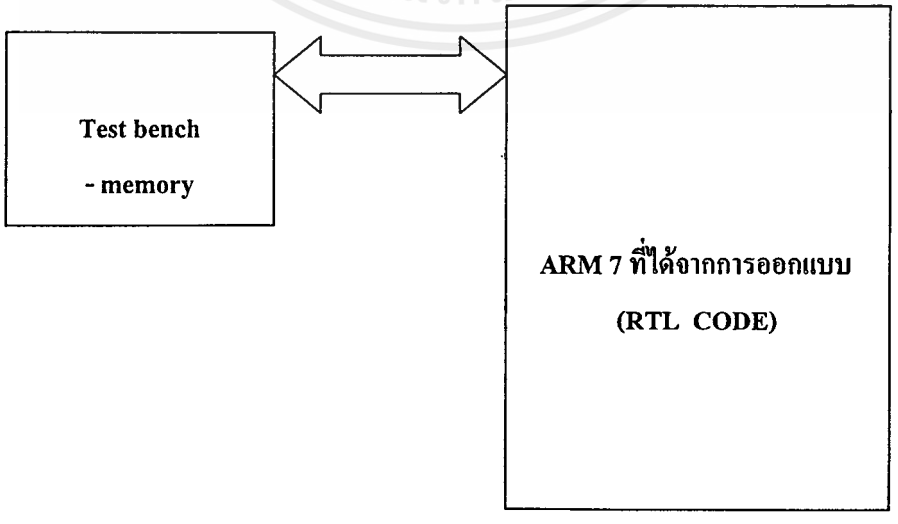
Temporary Register ที่อยู่ใต้ ALU นั้นไว้สำหรับเก็บค่าของผลลัพธ์ที่ได้จาก ALU เอาไว้เนื่องจากเป็นการทำงานแบบไปป์ไลน์จึงอาจทำให้ผลลัพธ์ที่ได้จาก ALU เปลี่ยนไปก่อนที่จะทำการนำค่าที่ได้ไปเก็บยังรีจิสเตอร์ได้ อีกทั้งมีไว้สำหรับการทำ Forwarding ด้วย

เมื่อทำการรวมทั้งหมดเข้าด้วยกัน จะได้ ARM7 ที่ออกแบบมีขาสัญญาณเป็นดังรูป 4- 7 โดยที่สัญญาณ LOCK นั้นไว้สำหรับเป็นการบอกไม่ให้อุปกรณ์ (Device) ตัวอื่นๆ เข้ามาใช้งานหน่วยความจำ ในขณะที่ทำคำสั่ง SWAP ขาสัญญาณ MREQ และ SEQ เป็นตัวบอกตัวควบคุมหน่วยความจำ (Memory Controller) ว่าต้องการที่จะทำการอ้างอิงหน่วยความจำ และตำแหน่งที่อ้างอิงนั้นเป็นตำแหน่งเดิมหรือเป็นตำแหน่งถัดมา 4 ตำแหน่งหรือไม่ ในไซเคิลถัดไป



รูปภาพ 4-9 ARM7 Microprocessor

4.3 แนวทางทดสอบ (Test Strategy)



รูปภาพ 4-10 รูปแบบของการทดสอบ Test Bench

การทดสอบนี้ทำขึ้นเพื่อจุดประสงค์ดังต่อไปนี้

- ตรวจสอบว่าการทำงานของ ARM 7 ที่ได้ทำขึ้นมานี้มีคุณสมบัติถูกต้อง
- สร้าง Test Bench ของแต่ละคำสั่งขึ้นมาเพื่อง่ายต่อการทดสอบ
- พยายามลดข้อผิดพลาดในการออกแบบที่เกิดขึ้น

โดยที่การตรวจสอบจะใช้คุณสมบัติ (Features) ดังต่อไปนี้เป็นพื้นฐาน

4.3.1 Instruction Set

ตรวจสอบว่า ไมโครโปรเซสเซอร์ที่ออกแบบนั้น สามารถที่จะทำงานในแค่ละคำสั่งของ ARM7 ได้ถูกต้องเหมือนกับการทำงานจริง หรือไม่

4.3.2 Instruction Pipelining

เป็นการตรวจสอบความถูกต้อง เมื่อนำแต่ละคำสั่งมาทำงานต่อเนื่องกัน เนื่องจากเป็นการทำงานแบบไปป์ไลน์ อาจจะเกิดการใช้คอมโพเนนท์ตัวเดียวกันได้ (คนละคำสั่ง คนละเฟสการทำงาน แต่อยู่ในไซเคิลเดียวกัน) ทำให้ทราบด้วยว่าคำสั่งใดห้ามทำงานต่อกัน

4.3.3 Pipeline Hazards

ตรวจสอบว่า ไมโครโปรเซสเซอร์ที่ได้ออกแบบนั้น มีการแก้ไขด้าน Pipeline Hazard หรือไม่ และถ้ามีสามารถแก้ไขปัญหาได้ถูกต้องหรือไม่

4.4 ขั้นตอนการทดสอบ (Test Plan)

ระบบที่ได้ทำการออกแบบไว้จำเป็นต้องมีการตรวจสอบการทำงาน เพื่อความถูกต้องและทำการแก้ไขข้อผิดพลาด ดังนั้นการทดสอบจึงจำเป็นต้องมีการวางแผนงานไว้เพื่อให้สามารถครอบคลุมความเป็นไปได้ในการทำงานทุกกรณี สิ่งสำคัญของระบบที่ได้ทำการออกแบบคือ การทำงานของไปป์ไลน์ เพราะจะมีการทำงานหลายๆอย่างพร้อมกัน ถ้าหากการจัดสรรการใช้งานไม่ลงตัวทำให้เกิดข้อผิดพลาดได้ง่าย

1. ทดสอบการทำงานของคำสั่งที่ใช้ในการกระโดดข้ามการทำงาน เนื่องจากต้องมีการกำหนดค่าให้กับ PC ใหม่ดังนั้นค่า PC ที่ได้ต้องถูกต้องและแม่นยำไม่สามารถมีข้อผิดพลาดได้เลย เพื่อการทำงานในส่วนของการ นำ คำสั่งถัดไปเข้ามารอการทำงานในขั้นต่อไป แต่ถ้าหากมีการกระโดดข้ามการทำงานติดต่อกันเรื่อยๆมากกว่า 1 ครั้งและมีการย้อมกลับไปยังตำแหน่งก่อนที่จะข้ามการทำงานไปอาจเกิดปัญหาได้ง่ายจึงต้องมีการทดสอบความถูกต้องตรงจุดนี้เพื่อความระมัดระวังในการข้ามการทำงานของคำสั่งดังเช่น ตัวอย่างต่อไปนี้

```
MOVS    R2,R15
BL      func1
MOVS    R3,R15
```

```
func1       BL      func2
            ADD     R3,R2,R1
func2       STRW    R2,[R15],R1
```

จากตัวอย่างข้างต้นจะเป็นการเล่นกับค่าใน PC เพื่อตรวจสอบความถูกต้องแม่นยำของค่า PC ไม่ว่าจะเป็นการนำค่า PC ไปใช้งานหรือเก็บค่าค่า PC ไว้ก็ตาม

2. ทดสอบกับการทำงานในส่วนของการเก็บและนำค่าจากหน่วยความจำ ในที่นี้คือใช้คำสั่งการ LDR และ STR เพราะการทำงานของระบบมีความจำเป็นต้องนำค่าเข้าออกจากหน่วยความจำ ดังนั้นการทำงานต้องมีความถูกต้อง ค่าที่นำไปใช้จะได้ไม่ผิดพลาด นอกจากนี้แล้วระบบที่ได้ทำการออกแบบยังมีการทำงานของการ LDR และ STR ไปพร้อมๆกันได้โดยการใช้คำสั่ง SWP ยิ่งทำให้ต้องเพิ่มความระมัดระวังมากยิ่งขึ้น ค่าที่ได้จึงผิดพลาดไม่ได้เลย

3. การทดสอบและดูเวลาที่ใช้ในการทำงานของการคูณ เนื่องจากระบบที่ได้ทำการออกแบบไม่สามารถทำการคูณได้อย่างระบบของ ARM 7 อย่างแท้จริง ดังนั้นต้องทำการวิเคราะห์ผลที่ได้อย่างรอบคอบ เนื่องจากการทำงานของไปป์ไลน์ จำเป็นต้องมีการหยุดรอการทำงาน เพื่อให้ระบบยังคงรักษาการทำงานของไปป์ไลน์ไว้ได้ต่อไป

4. ทดสอบการทำงานของ Flag เนื่องจาถว่าการทำงานของระบบ ที่ออกแบบนี้ให้ความสำคัญกับ Flag เป็นอย่างมากเพราะถ้าหากไม่มีความถูกต้องของ Flag แล้วจะไม่มีการทำงานใดๆเกิดขึ้นเลย จุดนี้ทำให้ต้องคำนึงถึงค่า Flag ทุกครั้งก่อนที่มีการทำงานของคำสั่งต่อไป

5. การทดสอบโดยการนำทุกคำสั่งที่ระบบสามารถทำงานได้มาทำงานเรียงกันไปให้ครบทุกกรณีว่าถ้าคำสั่งนั้นแล้วตามด้วยคำสั่งนี้การทำงานของไปป์ไลน์ จะยังคงทำงานได้อย่างปกติต่อไปหรือไม่ หากมีปัญหาที่เกิดขึ้นจำเป็นต้องมีการแก้ไขต่อไป

6. ทำสถิติของการใช้งานคำสั่งทุดคำสั่งในระบบว่ามีความถี่มากน้อยเพียงใด ในการทำงานเพื่อประโยชน์ของการคาดเดาว่าหากมีการทำงานของคำสั่งนี้แล้วคำสั่งที่น่าจะเป็นไปได้ในคำสั่งถัดไปคืออะไร

4.5 Test Bench ของการทดสอบระบบ ARM 7 ไมโครโปรเซสเซอร์

ในที่นี้ทางกลุ่มได้ทำการทดสอบไปแล้ว 2 โปรแกรมคือ ยังไม่ได้ทำการทดสอบการทำงานของโปรแกรมทดสอบการทำงาน(2) และยังจำเป็นต้องมีการทดสอบต่อไปอีกเป็นจำนวนมาก ซึ่งต้องใช้ระยะเวลาในการคิดวิธีและทำการทดสอบต่อไป โดยมีรายละเอียดของชุดคำสั่งที่ได้คิดขึ้นมาว่าจะทำการทดสอบดังต่อไปนี้

Address	Opcode	Byte Code
0	LDR R0,[R15,#44]	E59R002C h
4	ANDS R11,#0	E21BB000 h
8	SWPEQ R0,R0,[R2	01020090 h
12	ORR R11,[#A000]	E38BBCA0 h
16	ADDEQS R12,R9,R10	00B9C00A h
20	MLAS R6,R0,R1,R2	E0332091 h
24	SUBCS R7,R6,R5,LSR R4	20567435 h
28	SWPB R0,R1,[R2]	E1420091 h
32	STRB R7,[R3]!,R2,LSL #1	E6E37082 h
36	CMP R3,#3B	E353003B h
40	BLNE 35	1BFFFFFC h

ตาราง 4-2 ตารางโปรแกรมที่ใช้ในการทดสอบ (1)

Address	Opcode		Byte Code
0	MOV	R0,#d	E3A0000D h
4	MOV	R2,#5	E3A02005 h
8	MOV	r6,#5	E3A06000 h
12	SUB	R1,R6,#1	E2461001 h
16	MOV	R4,#0	E3A04000 h
20	RSBS	R3,R1,R2	E0713002 h
24	BMI	FUNC	4A000000 h
28	MLA	R4,R2,R0,R1	E0241092 h
32	SWP	R5,R3,[R4]	E1045093 h

ตาราง 4-3 ตารางโปรแกรมที่ใช้ในการทดสอบ (2)

โปรแกรม 2 โปรแกรมข้างต้นเป็นการทดสอบการทำงานของทุกคำสั่งที่ ARM7 ที่เราได้ออกแบบนั้นสามารถทำได้ว่า มีความถูกต้องหรือไม่ มีการทดสอบว่าสามารถทำ Forwarding และการทำงานที่มีการ Branch ที่ถูกต้องหรือไม่ รวมถึงการดูความถูกต้องเมื่อมีการนำค่า PC ไปใช้งานด้วย หลังการทดสอบพบว่าการทำงานเป็นไปโดยถูกต้อง

Address	Opecode		Byte Code
0	MOV	R5,R1	E1A05001 h
4	SUB	R1,R1,R2	E0411002 h
8	ADD	R3,R3,#1	E2833001 h
12	CMP	R1,#0	E3510000 h
16	BNE	LOOP	1AFFFFFB h
20	MOV	R4,R1,LSR #3	E1A041A5 h
24	CMP	R3,R4	E1530004 h
28	BEQ	KEEP	0A000000 h
32	MOV	R3,#XFF0	E3A03EFF h
36	STRB	R3,[R2],R1	E6C23001 h

ตาราง 4-4 ตารางโปรแกรมที่ใช้ในการทดสอบ (3)

เป็นโปรแกรมการทำการหารแบบง่ายๆ เพื่อที่จะสามารถพิสูจน์ได้ด้วยตนเองว่า การทำงานของระบบที่ได้ออกแบบมานั้นมีความถูกต้องหรือไม่ โดยที่ไม่ต้องอาศัยเครื่องคิดเลข หรืออุปกรณ์อื่นๆ มาทำการคำนวณ

Address	Instruction	Byte code	No.
0	LDR R0,[R15,#2C]	E59FCC2C	1
4	ANDS R11,#0	E21BB000	2
8	SWPEQ R0,R0,[R2]	01020090	3
12	ORR R11,#A00	E38BBCA0	4
16	ADDEQS R12,R9,R10	00B9C00A	5
20	MLAS R3,R0,R1,R2	E0332091	6
24	SUBCS R7,R6,R5,LSR #1	20567435	7
28	SWPB R0,R1,[R2]	E1420091	8
32	STRB R7,[R3]!,R2,LSR #1	E6E37082	9
36	CMP R3,#3B	E353003B	10
40	BLNE 35	1BFFFFFC	11
44		4123FC84	
48		00000000	
52		00000000	
56		00000000→⑨00000001	
60		01010101→③4123FC84→⑧ 4123FCFF	

ตาราง 4-5 ค่าในหน่วยความจำต่างๆ เมื่อทำตามโปรแกรมตารางที่ 4-2

0	FF00FF00 →①4123FC84→③01010101
1	000000FF →⑧00000001
2	0000003C
3	FF00FF00 →⑥0000003B →⑨000000B3
4	F72A1102
5	121EBC33
6	0487AF0D
7	00000022→⑦00000001
8	FF00FF00
9	456789AB
10	BA987655
11	FF00FF00 →② 00000000 →④0000A000
12	000000FF →⑤00000000
13	FFFFFFFF
14	00000022→(11)0000002C

ตาราง 4-6 ค่าในรีจิสเตอร์ไฟล์ในขณะทำคำสั่งตามตารางที่ 4-2

0	00000010
1	00000010 → ②00000008 →②00000000
2	00000008
3	00000000 →③00000001 →③00000002
4	F72A1102 →⑥00000002
5	121EBC33 →①00000010
6	0487AF0D
7	00000022
8	FF00FF00
9	456789AB
10	BA987655
11	FF00FF00
12	000000FF
13	FFFFFFFF
14	00000022

ตาราง 4-7 ค่าที่เกิดการเปลี่ยนแปลงไปตามการทำโปรแกรมตารางที่ 4-4

บทที่ 5

ผลการออกแบบ และพัฒนา

ในบทนี้อธิบายถึงโครงสร้างของแต่ละคอมโพเนนท์ ที่ได้ออกแบบ รวมทั้งผลการทดสอบการทำงานที่ได้ทำการกำหนดไว้ในบทที่แล้ว ว่าไมโครโปรเซสเซอร์ที่ออกแบบนั้น มีความสมบูรณ์ และใกล้เคียงกับ ARM7 มากน้อยเพียงไร

5.1 รายละเอียดของแต่ละคอมโพเนนท์ ที่ได้ออกแบบ และพัฒนา

5.1.1 Arithmetic & Logic (ALU)

```
Entity alu is
    Port    (alu_in1        : in std_logic_vector(31 downto 0);
             alu_in2        : in std_logic_vector(31 downto 0);
             sh_flag        : in std_logic;
             over_flag      : in std_logic;
             alu_mode       : in std_logic_vector(3 downto 0);
             alu_out        : out std_logic_vector(31 downto 0);
             cflag          : out std_logic;
             vflag          : out std_logic);
end alu;
```

ALU นั้นไม่ต้องทำการ Shift และการคูณ ในการออกแบบจึงไม่ยุ่งยาก แต่ ARM7 นั้นสามารถที่จะทำได้ทั้งการลบแบบไม่ใช้ Carry Flag หรือการลบที่ใช้ Carry Flag ด้วย และในการออกแบบนั้นก็ใช้วิธีการลบโดยการทำผ่านวิธีการบวก ซึ่งจะพบว่ามีปัญหาที่ Carry Flag ที่จะเป็น Carry Flag ที่ตรงข้ามกับ Carry Flag ที่จะต้องเกิดขึ้นจริง

ALU นั้นจะถูกใช้งานแทบทุกคำสั่ง โดยถ้าไม่ถูกใช้ในการทำ Arithemetic หรือLogic Operation ก็จะต้องเป็นตัวส่งค่าไปให้ Register File เพราะฉะนั้นในการออกแบบนั้น จะต้องมีการสั่งงานให้ ALU ทำงานให้ถูกต้อง และจำเป็นต้องมีตัวเก็บค่าผลลัพธ์ที่ได้ออกมาจาก ALU ไว้ เนื่องจากว่าการทำไปป์ไลน์นั้นอาจจะทำให้ผลลัพธ์ที่ได้ออกมาจาก ALU เปลี่ยนค่าไปก่อนที่จะนำค่านั้นไปเก็บยัง Register File ส่วนค่าของ Carry Flag และ Overflow Flag นั้น (ค่าของ Program Status Flag) จะใช้ค่าของสัญญาณที่ส่งออกมาจาก ALU เลย ซึ่งก็คือ ค่าของ Cflag และVflag นั่นเอง

5.1.2 Multiply

```
Entity mul is
    Port    (clk      : in std_logic;
             mul_in1   : in std_logic_vector(31 downto 0);
             mul_in2   : in std_logic_vector(31 downto 0);
             work      : in std_logic; -- mul
             done      : out std_logic;
             mul_out: out std_logic_vector(31 downto 0));
end mul;
```

Multiplier ที่ออกแบบนี้ยังไม่ได้ใช้วิธี Booth's Algorithm เนื่องจากยังไม่เข้าใจในการทำงานที่ดีพอ หากเสียเวลาทำความเข้าใจ จะทำให้การออกแบบล่าช้าได้ จึงได้ใช้วิธีการคูณแบบธรรมดาไปก่อน คือ การคูณแบบ Combination (การตั้งคูณที่เราใช้คิดกัน) และหากต้องการแก้เปลี่ยน ก็สามารถทำได้โดยไม่กระทบกับการทำงานของตัวอื่น โดยการแก้ไขที่ตัว Multiplier และ Control Unit เพียงเล็กน้อยเท่านั้น เนื่องจากในการออกแบบได้พยายามให้ใกล้เคียงกับการใช้วิธี Booth's Algorithm อยู่แล้ว

Multiplier ที่ออกแบบจำเป็นต้องใช้สัญญาณนาฬิกา (Clock) มาเป็นตัวหน่วงเวลาไว้ก่อนที่จะทำการส่งผลลัพธ์ออกไปให้ ALU เมื่อครบเวลา เนื่องจากวิธีการคูณที่ใช้นั้น สามารถที่จะทำงานเสร็จได้ภายในไซเคิลเดียว ซึ่งหากทำการส่งผลลัพธ์ออกไปเลย จะทำให้การทำงานผิดพลาด และไม่ใกล้เคียงกับ ARM7 ได้ และตัว Multiplier นั้นจะเริ่มทำงานเมื่อได้รับสัญญาณ work ที่ส่งมาจาก Control Unit และ Control Unit ทำการส่งสัญญาณ work ที่เป็นศูนย์ไปให้ Multiplier เพื่อบอกให้หยุดการทำงาน เมื่อได้รับสัญญาณ done

5.1.3 Register File

```
entity reg_file is
    port    (rf_in1   : in std_logic_vector(3 downto 0);
             rf_in2   : in std_logic_vector(3 downto 0);
             rf_des   : in std_logic_vector(3 downto 0);
             rf_dataw : in std_logic_vector(31 downto 0);
             rf_write : in std_logic;
             rf_din1  : out std_logic_vector(31 downto 0);
             rf_din2  : out std_logic_vector(31 downto 0));
end reg_file;
```

Register File นั้นจะมี 2 Input Port และ 1 Output Port แต่ว่า ARM7 นั้นสามารถที่จะทำคำสั่งที่ใช้ Register 3 ตัวเป็นโอเปอร์แลนด์ ได้ ดังนั้นจะเกิดปัญหาในกรณีเช่นนี้ได้ จึงจำเป็นต้องมีการยืดไซเคิล ที่ใช้ในการทำงานออกไปอีก 1 ไซเคิล ในการทดสอบนั้นจำเป็นต้องมีการกำหนดค่าเริ่มต้น (Initial) หรือจะเรียกได้ว่า ที่เก็บค่าของรีจิสเตอร์ต่างๆ ไว้ที่แพกเกจ (Package) ที่เขียนขึ้น คือทำการสร้างเป็นแอร์เรย์ (Array) เนื่องจากว่า Regiser File นั้นไม่มีสัญญาณรีเซต (Reset) ทำให้ไม่สามารถกำหนดไว้ใน Register File ที่เขียนขึ้น (ไม่สามารถกำหนดเป็นตัวแปรชนิด Variable ที่มีการกำหนดค่าเริ่มต้นได้) เพราะเมื่อไปทำการสังเคราะห์วงจร ตัวที่ทำการสังเคราะห์จะไม่สนใจการกำหนดค่าเริ่มต้น คือจะไม่ทำการกำหนดค่าเริ่มต้นตามที่ต้องการให้

5.1.4 Shifter

```
Entity shifter is
    Port    (reset          : in std_logic;
             sh_in1         : in std_logic_vector(31 downto 0);
             sh_in2         : in std_logic_vector(31 downto 0);
             sh_in3         : in std_logic_vector(12 downto 0);
             sh_mode        : in std_logic_vector(1 downto 0);
             op_regs        : in std_logic_vector(1 downto 0);
             sh_flag        : out std_logic;
             shift_out      : out std_logic_vector(31 downto 0));
end shifter;
```

Shifter นั้นก็ถูกใช้งานแทบทุกคำสั่งเช่นเดียวกับ ALU โดยหากไม่ทำการ Shift หรือ Rotate ก็จะต้องทำการผ่านค่าไปให้ยัง ALU จึงต้องมีการควบคุมการทำงานให้ดี ซึ่ง Shifter นั้นสามารถที่จะใช้ โอเปอร์แลนด์ ได้ 3 รูปแบบด้วยกันจึงต้องมีการระวังในส่วนนี้ให้ดี ในการออกแบบจึงใช้สัญญาณ Opregs เป็นตัวบอก Shifter ว่าจะใช้โอเปอร์แลนด์จาก sh_in1, sh_in2 หรือจาก sh_in1, sh_in3 หรือจาก sh_in3 และได้ใช้สัญญาณนี้เป็นตัวบอกด้วยว่าให้ Shifter ทำการผ่านค่าไปให้ ALU และ Shifter จะทำการส่งสัญญาณ sh_flag ไปให้ ALU ซึ่งจะใช้สัญญาณนี้เป็น Carry In

ARM7 นั้นสามารถที่จะทำการอ้างอิงค่า Immediate Value ได้เป็นค่ามากๆ โดยที่ไม่จำเป็นต้องใช้บิตในการอ้างอิงเป็นจำนวนมากๆ ด้วยการใช้การ Rotate Right และจำนวนครั้งในการ Rotate นั้นก็จะ 2 เท่าของจำนวนที่ได้ระบุไว้ทำให้ประหยัดบิตในการอ้างอิงได้

5.1.5 Control Unit

```
Entity ctl is
    Port    (reset          : in std_logic;
             clk            : in std_logic;
             ph1            : in std_logic;
             ph2            : in std_logic;
             data_in        : in std_logic_vector(31 downto 0);
             lock           : out std_logic;
             data_out       : out std_logic_vector(31 downto 0);
             mem_addr       : out std_logic_vector(31 downto 0);
             nRW            : out std_logic;
             nBW            : out std_logic;
             mreq           : out std_logic;
             seq            : out std_logic);
end ctl;
```

เป็นส่วนที่ออกแบบยากที่สุด เนื่องจากเป็นตัวควบคุมการทำงานของทุกๆ คอมโพเนนท์ โดยจะทำการถอดรหัสคำสั่งที่อ่านมาจากหน่วยความจำ เพื่อจะได้ทราบว่าต้องใช้คอมโพเนนท์ใด แล้วจึงส่งสัญญาณไปควบคุมคอมโพเนนท์ต่างๆ โดยที่แต่ละคำสั่งนั้นจะใช้ไซเคิลในการทำงานไม่เท่ากันจึงจำเป็นต้องมีการจดจำว่าเป็นไซเคิลที่เท่าไรของคำสั่งนั้นแล้ว (แต่ละไซเคิลก็มีการทำงานแตกต่างกันไป) ซึ่งจะพบว่าแตกต่างจากไปป์ไลน์ที่พบๆ มา ที่จะมักจะเป็น 1 คำสั่งใช้ 1 ไซเคิล ในการเอ็กซิคิวส์ และจะต้องส่งสัญญาณไปควบคุมการทำงานของแต่ละ Block ให้ดี

ในตอนแรกนั้นจะใช้วิธีส่งแอสเดรส (Address) ออกไปยังหน่วยความจำในเฟสที่ 2 ของไซเคิล แต่พบว่ามีปัญหาในการเขียนค่าลงหน่วยความจำสามารถจะเกิดการเขียนลงไป 2 ตำแหน่งได้ เนื่องมาจากการที่สัญญาณที่บอกให้หน่วยความจำรู้ว่าจะต้องทำการอ่านหรือเขียนนั้น เกิดการเปลี่ยนแปลงก่อนที่ Address จะเปลี่ยน จึงได้แก้ไขโดยให้ส่ง Address ออกไปในเฟสที่ 1 ของไซเคิล แทน

จะเห็นว่า Control Unit นี้ก็คือ ไมโครโปรเซสเซอร์นั่นเอง ซึ่งสามารถที่จะทำการแยก Control Unit ออกมาเป็นอีกคอมโพเนนท์หนึ่งก็ได้ ก่อนที่จะทำการรวมทุกๆ ตัวเป็นไมโครโปรเซสเซอร์อีกที

5.1.6 Memory

```
Entity memory is
    Port    (rw    : in std_logic; -- nRW
             bw    : in std_logic; -- nBW
             din   : in std_logic_vector(31 downto 0);
             addr  : in std_logic_vector(31 downto 0);
             dout  : out std_logic_vector(31 downto 0));
end memory;
```

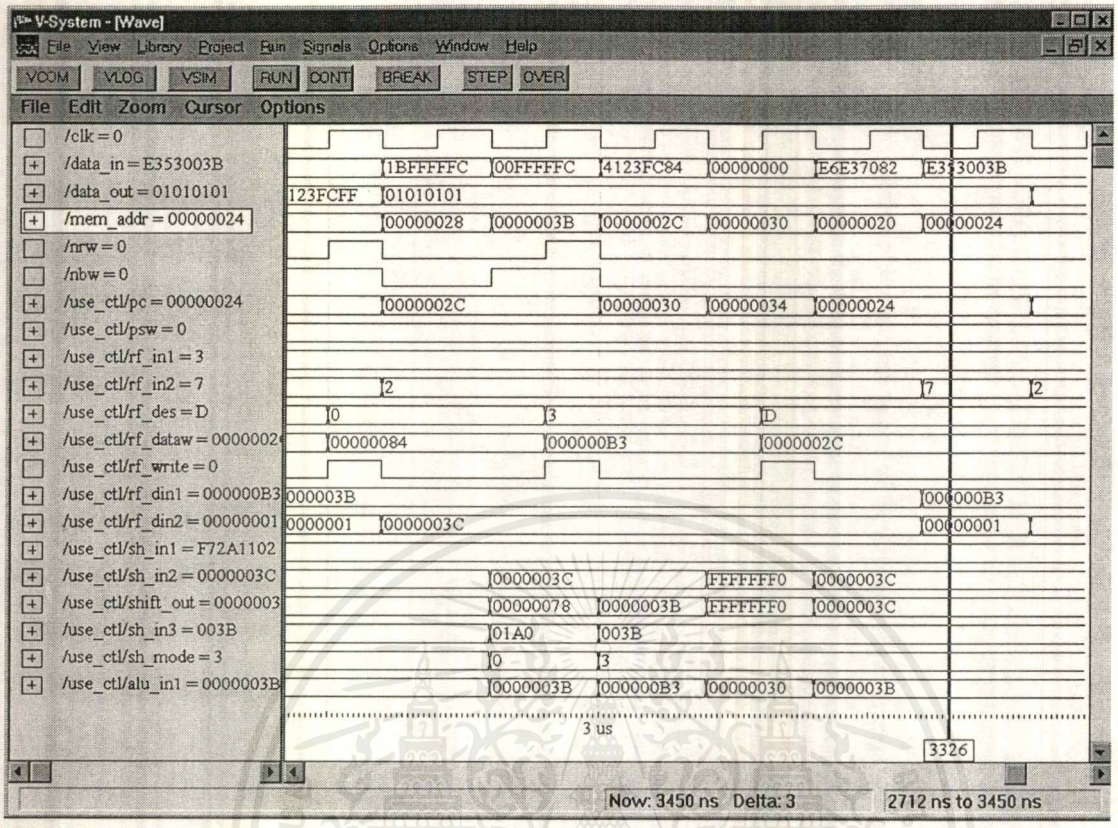
มีการสร้างเป็น Array ขนาด 1024 * 8 เพื่อใช้เป็นหน่วยความจำ โดยจะเก็บเป็นไบต์ เนื่องจากคำสั่งใน ARM 7 สามารถทำการอ้างอิงเป็นไบต์ได้ ในโครงงานนี้เราจำเป็นต้องทำการจำลองสร้างหน่วยความจำขึ้นมาด้วย เพื่อที่จะสามารถทำการทดสอบการทำงานได้เสมือนว่าเป็นการทำงานจริง นั่นคือ จะต้องมีการอ่านคำสั่งมาจากหน่วยความจำ รวมทั้งสามารถที่จะทำการอ่านค่าจากหน่วยความจำในการทำคำสั่ง LOAD หรือการเก็บค่าที่ต้องการลงหน่วยความจำในการทำคำสั่ง STORE

5.2 ผลการจำลองการทำงาน

หลังจากที่ได้เขียน Code และทำการจำลองการทำงานแล้ว ผลการทดลองของคอมโพเนนท์ต่างๆ เป็นดังนี้

เพื่อตรวจสอบความถูกต้องว่า ARM7 ที่ได้ออกแบบนั้นมีการทำงานที่ถูกต้องหรือไม่ จึงได้ใช้ Test Bench 2 แบบด้วยกันได้ผลดังนี้

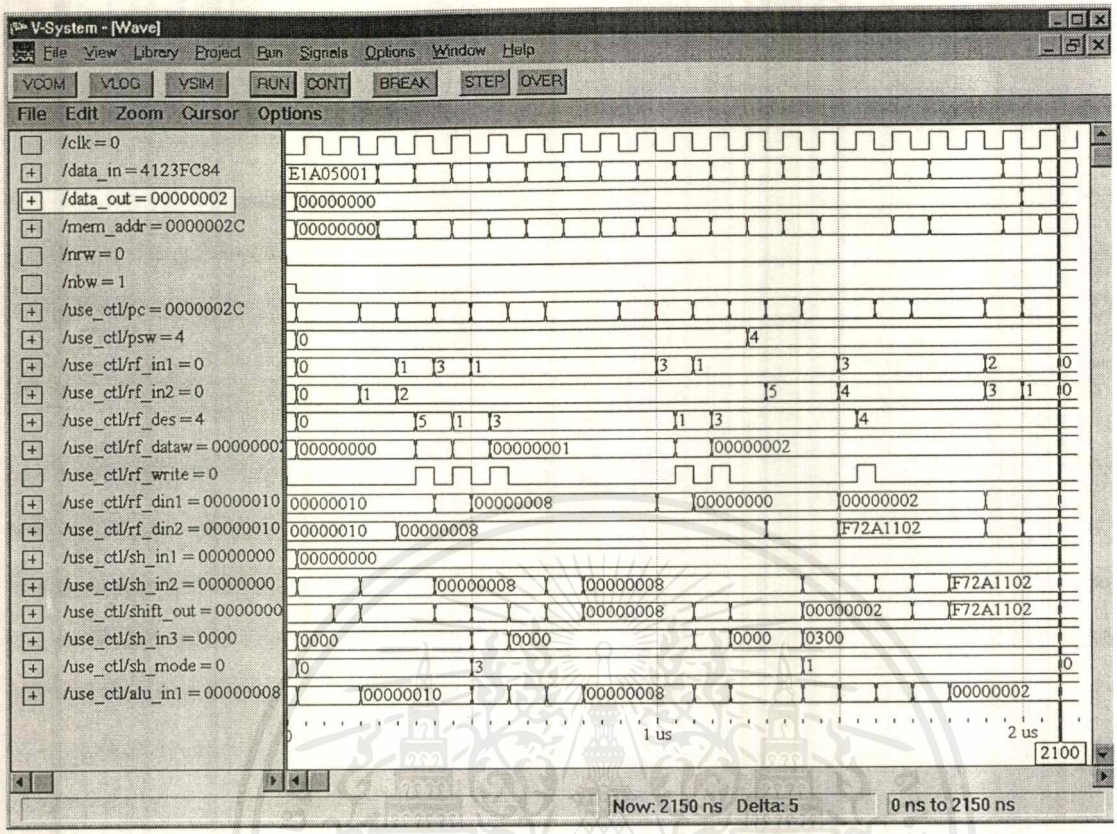
5.2.1 ทดสอบความถูกต้องของการทำงานของทุกคำสั่งที่สามารถทำได้



รูปภาพ 5-1 Waveform ของการทำคำสั่ง SUBCS , SWPB , STRB , CMP , BLNE

เป็นการทดสอบ การทำงานในแต่ละคำสั่งว่า สามารถทำงานได้ถูกต้องตามการทำงานของคำสั่ง ARM7 หรือไม่ โดยที่การทดสอบนี้ยังไม่ได้เป็นการทดสอบครบทุกรูปแบบของการใช้คำสั่งนั้นๆ จากกราฟ เมื่อสิ้นสุดการทำงาน (เส้นสีดำเข้ม) ค่าของ Mem_addr (ตำแหน่งในหน่วยความจำที่ต้องการอ้างอิงถัดมา) เป็น 24H ซึ่งเป็นค่าตำแหน่งที่เก็บคำสั่ง CMP ไว้ (ค่าของ Data_in เป็น Code ของคำสั่ง CMP ในรูปฐาน 16) และก็ถูกต้องตามการทำงานของคำสั่ง Branch ที่ต้องทำการนำคำสั่ง ณ ตำแหน่งใหม่ และคำสั่งถัดมาเข้ามาก่อน จึงจะถือว่าสิ้นสุดการทำงานคำสั่ง Branch โปรแกรม และผลที่ได้จากการทำงานในแต่ละคำสั่งในโปรแกรม แสดงไว้ในตารางที่ 4-2, ตารางที่ 4-5 และตารางที่ 4-6 (บทที่ 4) ตามลำดับ

5.2.2 โปรแกรมทำการหาร ทดสอบความถูกต้องโดยรวม



รูปภาพ 5-2 Waveform ของการทำโปรแกรมการหาร

เป็นการทดสอบว่า สามารถที่จะทำงานตามโปรแกรมที่เขียนขึ้นด้วยชุดคำสั่งของ ARM7 ถูกต้องหรือไม่ โดยโปรแกรมนี้เป็นการทำการหารแบบง่ายๆ คือ การหาร 8 ด้วย 4 ซึ่งคำตอบที่ได้จะถูกนำไปเก็บลงหน่วยความจำตำแหน่งที่ 8 จะสามารถเห็นได้จากกราฟ ว่าการทำงานได้ผลลัพธ์ที่ถูกต้อง คือ จากเส้นสีดำเข้ม ที่เป็นตัวบอกสิ้นสุดการทำงาน จะเห็นค่าของ Mem_addr (ตัวชี้ตำแหน่งในหน่วยความจำ) มีค่าเป็น 8 และค่าของ Data_out (ค่าที่ต้องการจะเขียน) มีค่าเป็น 2 และสัญญาณ nRW มีค่าเป็น 1 ซึ่งเป็นการบอกให้เขียนข้อมูลใน Data_out ลงหน่วยความจำ โปรแกรม และผลจากการทำงาน ในแต่ละคำสั่งในโปรแกรม แสดงอยู่ในตารางที่ 4-4 และตารางที่ 4-7 (บทที่ 4) ตามลำดับ

5.3 ผลการทดสอบ

5.3.1 Instruction Set

จากการทดสอบด้วยการจำลองการทำงาน พบว่าไมโครโปรเซสเซอร์ที่ออกแบบสามารถทำคำสั่งของ ARM7 ได้ทั้งสิ้น 22 คำสั่ง โดยคำสั่งที่ยังไม่สามารถทำงานได้ คือ Block Data Transfer และคำสั่งการคูณ (Multiplier) ซึ่งในการออกแบบนี้ ยังใช้วิธีการคูณแบบ Combination ไปก่อน เนื่องจากยังไม่เข้าใจในการทำงานเพียงพอ

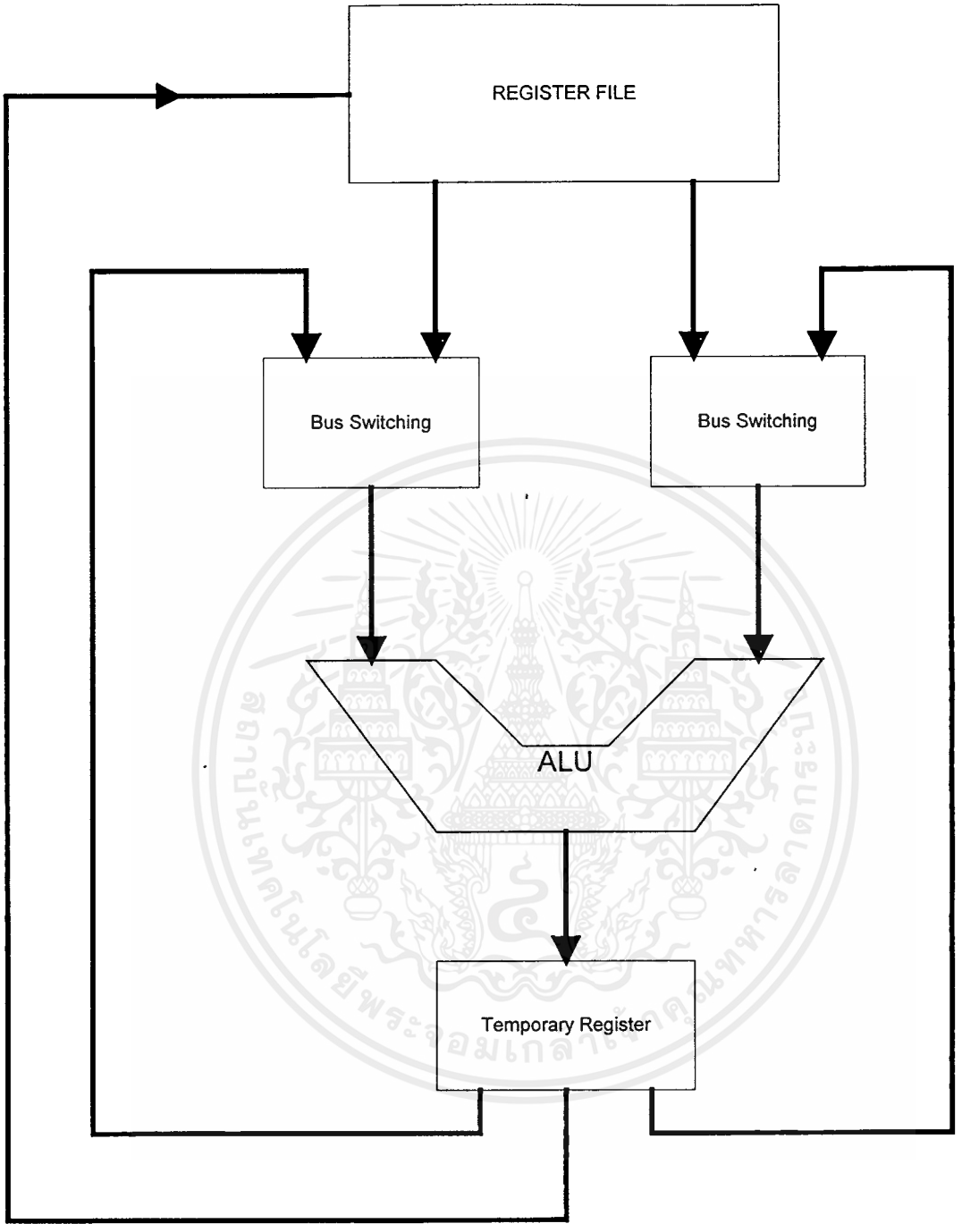
5.3.2 Pipeline Hazards

5.3.2.1 Data Hazard

ปัญหาที่เกิดขึ้นในการออกแบบ ARM 7 นี้คือ การอ่านค่าในรีจิสเตอร์ไปใช้ก่อนที่จะเขียนค่าใหม่ที่ได้จากการเอ็กซิคิวส์ในคำสั่งที่แล้ว ทำให้นำค่าที่ยังไม่ได้แก้ไขไปใช้ จึงทำให้ต้องมีการป้อนค่ากลับเพื่อนำค่านั้นไปใช้ทันทีก่อนที่มีการเขียนค่าลงในรีจิสเตอร์เรียกว่า การทำฟอร์เวิร์ดดิ่ง (Forwarding) โดยหากตรวจสอบได้ว่า คำสั่งนั้นต้องการใช้ค่าจากรีจิสเตอร์ตัวเดียวกับรีจิสเตอร์ที่เป็นตัวเก็บผลลัพธ์ (Destination Register) ในคำสั่งที่แล้ว ก็จะทำการอ่านค่าจากรีจิสเตอร์ที่นำมาเก็บค่าของผลลัพธ์ที่ออกมาจาก ALU ตัวนี้แทน ดังแสดงในรูป 5-3

5.3.2.2 Control Hazard

ARM7 ก็ใช้วิธี Delay Branch ที่กล่าวไว้ในบทที่ 2 ในการแก้ปัญหา เมื่อเกิดการกระโดดไปยังตำแหน่ง (Branch) อื่นขึ้น โดยที่ ARM7 นั้นจะถือว่าสิ้นสุดการทำการเอ็กซิคิวส์คำสั่ง Branch ก็ต่อเมื่อได้ทำการเติมให้ไปป์ไลน์สมบูรณ์ดังเดิมแล้ว หรือก็คือ จะสิ้นสุดการทำงานคำสั่ง Branch ก็ต่อเมื่อได้ทำการนำคำสั่งที่กระโดดไป กับคำสั่งถัดจากนั้นมาแล้ว และได้ทำการถอดรหัสคำสั่งที่กระโดดไปทำงานเป็นที่เรียบร้อยแล้วนั่นเอง



รูปภาพ 5-3 Forwarding Schema

บทที่ 6
สรุปและวิจารณ์

6.1 สรุปผลการออกแบบ และพัฒนา

จากการออกแบบ และทดสอบการทำงานในแต่ละคอมโพเนนท์นั้นมีความถูกต้องสมบูรณ์ โดยการทดสอบในกรณีต่างๆ ที่คาดว่าจะต้องผิดพลาดหากการออกแบบนั้นไม่ถูกต้อง และในส่วนของตัว Multiplier นั้นยังไม่ได้ออกแบบให้มีการทำงานเป็นแบบ Booth's Algorithm แต่ใช้วิธีออกแบบให้ใช้วิธีคูณแบบปกติไปก่อน

เมื่อนำทุกคอมโพเนนท์มารวมกัน และทำการทดสอบความถูกต้องโดยในขณะนี้ยังไม่สามารถที่จะยืนยันได้ว่า ARM7 ที่ออกแบบมีความสมบูรณ์ 100 เปอร์เซ็นต์ เนื่องจากยังไม่สามารถทำการทดสอบได้ครบทุกกรณีของการนำคำสั่งมาใช้ต่อกัน (การทดสอบคุณสมบัติ Instruction Pipelining) โดยที่การทดสอบให้ครบทุกกรณีนั้นจำเป็นต้องใช้เวลาในการทดสอบนานพอสมควร และชุดคำสั่งที่ยังไม่สามารถทำได้ก็คือ การทำ Block Data Transfer ซึ่งเป็นคำสั่งพิเศษที่ ARM7 มี

อย่างไรก็ตามโครงงานนี้ก็ถือว่าประสบความสำเร็จตามที่ได้กำหนดไว้ คือสามารถทำการจำลองการทำงานได้ในระดับหนึ่ง เนื่องจากงานนี้เป็นการออกแบบ Microprocessor ขนาด 32 บิต ซึ่งมีความยุ่งยาก และมีรายละเอียดมากพอควร

6.2 สรุปภาพรวมทั้งหมดของที่ได้ออกแบบ

ไมโครโปรเซสเซอร์ที่ได้ทำการออกแบบนี้สามารถทำงานได้ตามคำสั่งของ ARM 7 22 คำสั่งคิดเป็น 73 เปอร์เซ็นต์ จากคำสั่งทั้งหมดของ ARM 30 คำสั่ง

Code ที่ได้จากการเขียนอธิบายการทำงานสามารถสรุปได้ดังนี้

- จำนวนของ package ที่ใช้ 1 package
- จำนวนของ process ที่ใช้มีทั้งหมด 18 process
- จำนวนของ function ที่ใช้มี 10 function
- มีจำนวนบรรทัดทั้งหมด 2144 บรรทัด

ส่วนของการทดสอบด้วย Test Bench ซึ่งมีด้วยกัน 3 โปรแกรมดังนี้

- บรรทัดคำสั่งที่ใช้ 30 คำสั่ง
- จำนวนบรรทัดที่ใช้ 99 บรรทัด

6.3 ข้อเสนอแนะ

หากต้องการจะทำโครงงานนี้ต่อไป ก็สามารถที่จะทำได้ โดยจะเป็นการตรวจสอบการทำงานทั้งหมดเพื่อดูความสมบูรณ์ ด้วยการทำการสร้าง Test Bench มาทดสอบให้ครบทุกกรณีของการนำคำสั่งต่างๆ มาทำงานต่อกันว่าไปป์ไลน์ที่ได้นั้นถูกต้องหรือไม่ รวมถึงการออกแบบการทำงานของคำสั่ง Block

Data Transfer รวมทั้งการแก้ไขให้ตัว Multiplier มีการทำงานเป็นแบบ Booth's Algorithm ซึ่งการแก้ไขก็เพียงแต่แก้ไขที่ตัว Multiplier กับ Control Unit เพียงเล็กน้อยเท่านั้น

นอกจากนี้ยังสามารถที่จะทำการแก้ไข และออกแบบการทำงานเพิ่มเติมจนสามารถที่จะทำการสังเคราะห์เป็นวงจรได้ โดยจะต้องมีการคำนึงถึง Delay ที่จะเกิดขึ้นด้วย และนำไปทำ FPGA ลงเป็นชิพได้ในที่สุด ซึ่งจะพบว่ายังมีรายละเอียดอีกมากมายในการทำงาน ถึงจะสามารถได้ชิพออกมา

แต่หากสามารถทำได้สำเร็จ เราก็จะสามารถผลิตชิพได้เอง โดยที่ไม่ต้องสั่งจากต่างประเทศ ซึ่งจะต้องเสียค่าใช้จ่ายสูง เนื่องมาจากการที่จะต้องเสียค่าลิขสิทธิ์ของการออกแบบชิพนั้นๆ ขึ้นมา

ภาคผนวก

ARM Instruction Set Quick Reference Card

Key to Tables	
{cond}	Refer to Table **Condition Field {cond}**
<Oprnd2>	Refer to Table **Oprnd2**
{field}	Refer to Table **Field**
S	Sets condition codes (optional)
B	Byte operation (optional)
H	Halfword operation (optional)
T	Forces address translation. Cannot be used with pre-indexed addresses
<a_mode1>	Refer to Table **Addressing Mode 1**
<a_mode2>	Refer to Table **Addressing Mode 2**
<a_mode3>	Refer to Table **Addressing Mode 3**
<a_mode4>	Refer to Table **Addressing Mode 4**
<a_mode5>	Refer to Table **Addressing Mode 5**
<a_mode6>	Refer to Table **Addressing Mode 6**
#32_Bit_Immed	A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits

Operation		Assembler	S updates	Action	Notes
Move	Move	`MOV{cond}{S} Rd, <Oprnd2>`	N Z C	Rd:= <Oprnd2>	
	NOT	`MVN{cond}{S} Rd, <Oprnd2>`	N Z C	Rd:= 0xFFFFFFFF EOR <Oprnd2>	
	SPSR to register	`MRS{cond} Rd, SPSR`		Rd:= SPSR	*Architecture 3, 3M and 4 only*
	CPSR to register	`MRS{cond} Rd, CPSR`		Rd:= CPSR	*Architecture 3, 3M and 4 only*
	register to SPSR	`MSR{cond} SPSR{field}, Rm`		SPSR:= Rm	*Architecture 3, 3M and 4 only*
	register to CPSR	`MSR{cond} CPSR{field}, Rm`		CPSR:= Rm	*Architecture 3, 3M and 4 only*
	immediate to SPSR flags	`MSR{cond} SPSR_f, #32_Bit_Immed`		SPSR:= #32_Bit_Immed	*Architecture 3, 3M and 4 only*
	immediate to CPSR flags	`MSR{cond} CPSR_f, #32_Bit_Immed`		CPSR:= #32_Bit_Immed	*Architecture 3, 3M and 4 only*
ALU	Arithmetic				
	Add	`ADD{cond}{S} Rd, Rn, <Oprnd2>`	N Z C V	Rd:= Rn + <Oprnd2>	
	with carry	`ADC{cond}{S} Rd, Rn, <Oprnd2>`	N Z C V	Rd:= Rn + <Oprnd2> + Carry	
	Subtract	`SUB{cond}{S} Rd, Rn, <Oprnd2>`	N Z C V	Rd:= Rn - <Oprnd2>	
	with carry	`SBC{cond}{S} Rd, Rn, <Oprnd2>`	N Z C V	Rd:= Rn - <Oprnd2> - NOT(Carry)	
	reverse subtract	`RSB{cond}{S} Rd, Rn, <Oprnd2>`	N Z C V	Rd:= <Oprnd2> - Rn	
	reverse subtract with carry	`RSC{cond}{S} Rd, Rn, <Oprnd2>`	N Z C V	Rd:= <Oprnd2> - Rn - NOT(Carry)	
	Negate				
	Multiply	`MUL{cond}{S} Rd, Rm, Rs`	N Z	Rd:= Rm * Rs	*Not in Architecture 1*
	accumulate	`MLA{cond}{S} Rd, Rm, Rs, Rn`	N Z	Rd:= (Rm * Rs) + Rn	*Not in Architecture 1*
	unsigned long	`UMULL{cond}{S} RdHi, RdLo, Rm, Rs`	N Z	RdHi:= (Rm*Rs)[63:32] RdLo:= (Rm*Rs)[31:0]	*Architecture 3M and 4 only*
	unsigned accumulate long	`UMLAL{cond}{S} RdHi, RdLo, Rm, Rs`	N Z	RdLo:=(Rm*Rs)+RdLo RdHi:=(Rm*Rs)+RdHi+ CarryFrom((Rm*Rs)[31:0]+RdLo))	*Architecture 3M and 4 only*
	signed long	`SMULL{cond}{S} RdHi, RdLo, Rm, Rs`	N Z	RdHi:= signed(Rm*Rs)[63:32] RdLo:= signed(Rm*Rs)[31:0]	*Architecture 3M and 4 only*
	signed accumulate long	`SMLAL{cond}{S} RdHi, RdLo, Rm, Rs`	N Z	RdHi:=signed(Rm*Rs)+RdHi+ CarryFrom((Rm*Rs)[31:0]+RdLo))	*Architecture 3M and 4 only*
	Compare	`CMP{cond} Rd, <Oprnd2>`	N Z C V	CPSR flags:= Rn - <Oprnd2>	
	negative	`CMN{cond} Rd, <Oprnd2>`	N Z C V	CPSR flags:= Rn + <Oprnd2>	
	Logical				
	Test	`TST{cond} Rn, <Oprnd2>`	N Z C	CPSR flags:= Rn AND <Oprnd2>	
	Test equivalence	`TEQ{cond} Rn, <Oprnd2>`	N Z C	CPSR flags:= Rn EOR <Oprnd2>	Does not update the V flag
	AND	`AND{cond}{S} Rd, Rn, <Oprnd2>`	N Z C	Rd:= Rn AND <Oprnd2>	
	EOR	`EOR{cond}{S} Rd, Rn, <Oprnd2>`	N Z C	Rd:= Rn EOR <Oprnd2>	
	ORR	`ORR{cond}{S} Rd, Rn, <Oprnd2>`	N Z C	Rd:= Rn OR <Oprnd2>	
	Bit Clear	`BIC{cond}{S} Rd, Rn, <Oprnd2>`	N Z C	Rd:= Rn AND NOT <Oprnd2>	
	Shift/Rotate				See Table **Oprnd2**

ARM Instruction Set Quick Reference Card

Operation		Assembler	Action	Notes
Branch	Branch	`B{cond} label`	R15:= address	
	with link	`BL{cond} label`	R14:=R15, R15:= address	
	and exchange instruction set	`BX{cond} Rn`	R15:=Rn, T bit:= Rn[0]	*Architecture 4 with Thumb only* Thumb state; Rn[0] = 0 ARM state; Rn[0] =1
Load	Word	`LDR{cond} Rd, <a_mode1>`	Rd:= [address]	
	with user-mode privilege	`LDR{cond}T Rd, <a_mode2>`		
	Byte	`LDR{cond}B Rd, <a_mode1>`	Rd:= [byte value from address] Loads bits 0 to 7 and sets bits 8-31 to 0	
	with user-mode privilege	`LDR{cond}BT Rd, <a_mode2>`		
	signed	`LDR{cond}SB Rd, <a_mode3>`	Rd:= [signed byte value from address] Loads bits 0 to 7 and sets bits 8-31 to bit 7	*Architecture 4 only*
	Halfword	`LDR{cond}H Rd, <a_mode3>`	Rd:= [halfword value from address] Loads bits 0 to 15 and sets bits 16-31 to 0	*Architecture 4 only*
	signed	`LDR{cond}SH Rd, <a_mode3>`	Rd:= [signed halfword value from address] Loads bits 0 to 15 and sets bits 16-31 to bit 15	*Architecture 4 only*
	Multiple			
	Block data operations			
	Increment Before	`LDM{cond}IB Rd{!}, <regs>{^}`	Stack manipulation (pop)	! sets the W bit (updates the base register after the transfer ^ sets the S bit
	Increment After	`LDM{cond}IA Rd{!}, <regs>{^}`		
	Decrement Before	`LDM{cond}DB Rd{!}, <regs>{^}`		
	Decrement After	`LDM{cond}DA Rd{!}, <regs>{^}`		
	Stack operations	`LDM{cond}<a_mode4> Rd{!},<registers>`		! sets the W bit (updates the base register after the transfer
	and restore CPSR	`LDM{cond}<a_mode4> Rd{!}, <registers+pc>`		
	User registers	`LDM{cond}<a_mode4> Rd, <registers>^`		
Store	Word	`STR{cond} Rd, <a_mode1>`	[address]:= Rd	
	with user-mode privilege	`STRT{cond} Rd, <a_mode2>`		
	Byte	`STRB{cond} Rd, <a_mode1>`	[address]:= byte value from Rd	
	with user-mode privilege	`STRBT{cond} Rd, <a_mode2>`		
	Halfword	`STR{cond}H Rd, <a_mode3>`	[address]:= halfword value from Rd	*Architecture 4 only*
	Multiple			
	Block data operations			
	Increment Before	`STM{cond}IB Rd{!}, <registers>{^}`	Stack manipulation (push)	! sets the W bit (updates the base register after the transfer ^ sets the S bit
	Increment After	`STM{cond}IA Rd{!}, <registers>{^}`		
	Decrement Before	`STM{cond}DB Rd{!}, <registers>{^}`		
	Decrement After	`STM{cond}DA Rd{!}, <registers>{^}`		
	Stack operations	`STM{cond}<a_mode5> Rd{!}, <regs>`		
	User registers	`STM{cond}<a_mode5> Rd{!}, <regs>^`		
Swap	Word	`SWP{cond} Rd, Rm, [Rn]`		*Not in Architecture 1 or 2*
	Byte	`SWP{cond}B Rd, Rm, [Rn]`		*Not in Architecture 1 or 2*
Coprocessors	Data operations	`CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2>`		*Not in Architecture 1*
	Move to ARM reg from coproc	`MRC{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`		
	Move to coproc from ARM reg	`MCR{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`		
	Load	`LDC{cond} p<cpnum>, CRd, <a_mode6>`		
	Store	`STC{cond} p<cpnum>, CRd, <a_mode6>`		
Software Interrupt		`SWI #24_Bit_Value`		24-bit immediate value

ARM Addressing Modes
Quick Reference Card

Addressing Mode 1	
Immediate offset	`[Rn, #+/-12_Bit_Offset]`
Register offset	`[Rn, +/-Rm]`
Scaled register offset	`[Rn, +/-Rm, LSL #shift_imm]`
	`[Rn, +/-Rm, LSR #shift_imm]`
	`[Rn, +/-Rm, ASR #shift_imm]`
	`[Rn, +/-Rm, ROR #shift_imm]`
	`[Rn, +/-Rm, RRX]`
Pre-indexed offset	
Immediate	`[Rn, #+/-12_Bit_Offset]!`
Register	`[Rn, +/-Rm]!`
Scaled register	`[Rn, +/-Rm, LSL #shift_imm]!`
	`[Rn, +/-Rm, LSR #shift_imm]!`
	`[Rn, +/-Rm, ASR #shift_imm]!`
	`[Rn, +/-Rm, ROR #shift_imm]!`
	`[Rn, +/-Rm, RRX]!`
Post-indexed offset	
Immediate	`[Rn], #+/-12_Bit_Offset`
Register	`[Rn], +/-Rm`
Scaled register	`[Rn], +/-Rm, LSL #shift_imm`
	`[Rn], +/-Rm, LSR #shift_imm`
	`[Rn], +/-Rm, ASR #shift_imm`
	`[Rn], +/-Rm, ROR #shift_imm`
	`[Rn, +/-Rm, RRX]`

Addressing Mode 2	
Immediate offset	`[Rn, #+/-12_Bit_Offset]`
Register offset	`[Rn, +/-Rm]`
Scaled register offset	`[Rn, +/-Rm, LSL #shift_imm]`
	`[Rn, +/-Rm, LSR #shift_imm]`
	`[Rn, +/-Rm, ASR #shift_imm]`
	`[Rn, +/-Rm, ROR #shift_imm]`
	`[Rn, +/-Rm, RRX]`
Post-indexed offset	
Immediate	`[Rn], #+/-12_Bit_Offset`
Register	`[Rn], +/-Rm`
Scaled register	`[Rn], +/-Rm, LSL #shift_imm`
	`[Rn], +/-Rm, LSR #shift_imm`
	`[Rn], +/-Rm, ASR #shift_imm`
	`[Rn], +/-Rm, ROR #shift_imm`
	`[Rn, +/-Rm, RRX]`

Addressing Mode 3 - Signed Byte and Halfword Data Transfer	
Immediate offset	`[Rn, #+/-8_Bit_Offset]`
Pre-indexed	`[Rn, #+/-8_Bit_Offset]!`
Post-indexed	`[Rn], #+/-8_Bit_Offset`
Register	`[Rn, +/-Rm]`
Pre-indexed	`[Rn, +/-Rm]!`
Post-indexed	`[Rn], +/-Rm`

Addressing Mode 6 - Coprocessor Data Transfer	
Immediate offset	`[Rn, #+/-(8_Bit_Offset*4)]`
Pre-indexed	`[Rn, #+/-(8_Bit_Offset*4)]!`
Post-indexed	`[Rn], #+/-(8_Bit_Offset*4)`

Oprnd2	
Immediate value	`#32_Bit_Immed`
Logical shift left	`Rm LSL #5_Bit_Immed`
Logical shift right	`Rm LSR #5_Bit_Immed`
Arithmetic shift right	`Rm ASR #5_Bit_Immed`
Rotate right	`Rm ROR #5_Bit_Immed`
Register	`Rm`
Logical shift left	`Rm LSL Rs`
Logical shift right	`Rm LSR Rs`
Arithmetic shift right	`Rm ASR Rs`
Rotate right	`Rm ROR Rs`
Rotate right extended	`Rm RRX`

Field		
Suffix	**Sets**	
`_c`	Control field mask bit	(bit 3)
`_f`	Flags field mask bit	(bit 0)
`_s`	Status field mask bit	(bit 1)
`_x`	Extension field mask bit	(bit 2)

Condition Field {cond}	
Suffix	**Description**
`EQ`	Equal
`NE`	Not equal
`CS`	Unsigned higher or same
`CC`	Unsigned lower
`MI`	Negative
`PL`	Positive or zero
`VS`	Overflow
`VC`	No overflow
`HI`	Unsigned higher
`LS`	Unsigned lower or same
`GE`	Greater or equal
`LT`	Less than
`GT`	Greater than
`LE`	Less than or equal
`AL`	Always

Addressing Mode 4			
Addressing Mode		**Stack Type**	
`IA`	Increment After	`FD`	Full Descending
`IB`	Increment Before	`ED`	Empty Descending
`DA`	Decrement After	`FA`	Full Ascending
`DB`	Decrement Before	`EA`	Empty Ascending

Addressing Mode 5			
Addressing Mode		**Stack Type**	
`IA`	Increment After	`EA`	Empty Ascending
`IB`	Increment Before	`FA`	Full Ascending
`DA`	Decrement After	`ED`	Empty Descending
`DB`	Decrement Before	`FD`	Full Descending

4.0 Instruction Set

4.1 Instruction Set Summary

A summary of the ARM7 instruction set is shown in *Figure 7: Instruction Set Summary.*

Note: some instruction codes are not defined but do not cause the Undefined instruction trap to be taken, for instance a Multiply instruction with bit 6 changed to a 1. These instructions shall not be used, as their action may change in future ARM implementations.

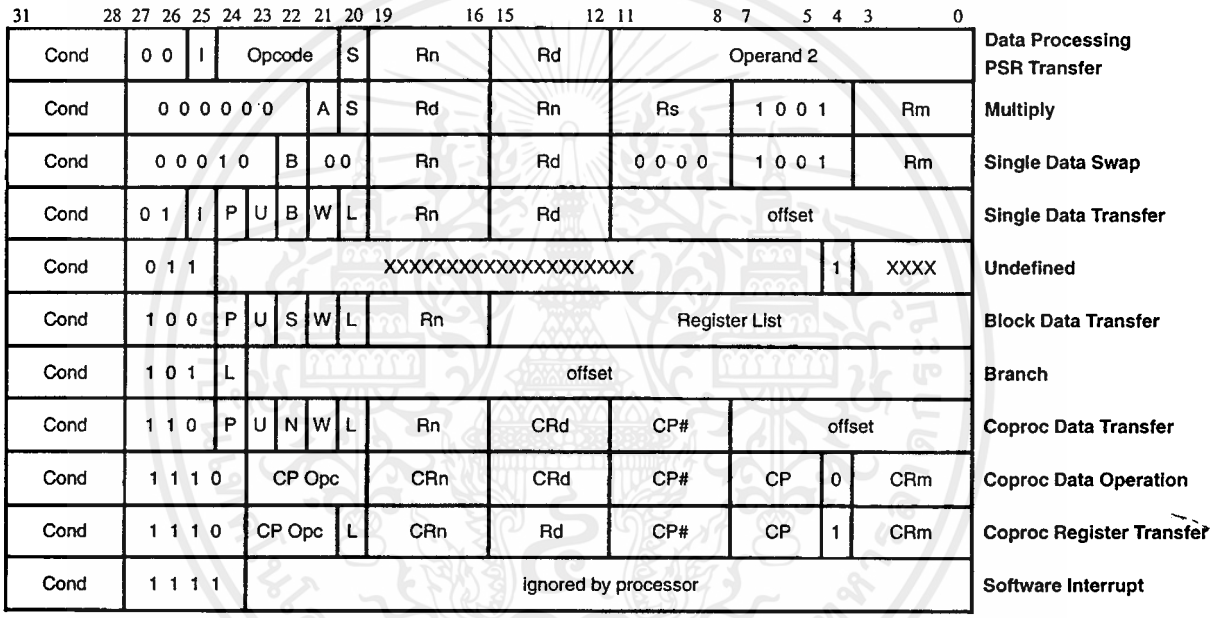


Figure 7: Instruction Set Summary

4.2 The Condition Field

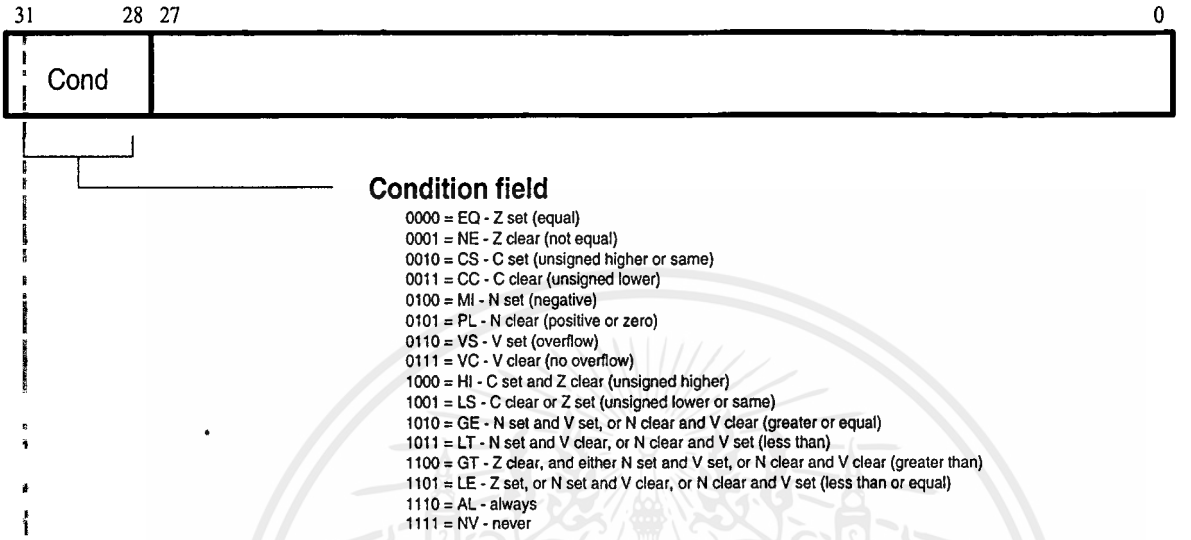


Figure 8: Condition Codes

All ARM7 instructions are conditionally executed, which means that their execution may or may not take place depending on the values of the N, Z, C and V flags in the CPSR. The condition encoding is shown in *Figure 8: Condition Codes*.

If the *always* (AL) condition is specified, the instruction will be executed irrespective of the flags. The *never* (NV) class of condition codes shall not be used as they will be redefined in future variants of the ARM architecture. If a NOP is required it is suggested that MOV R0,R0 be used. The assembler treats the absence of a condition code as though *always* had been specified.

The other condition codes have meanings as detailed in *Figure 8: Condition Codes*, for instance code 0000 (EQual) causes the instruction to be executed only if the Z flag is set. This would correspond to the case where a compare (CMP) instruction had found the two operands to be equal. If the two operands were different, the compare instruction would have cleared the Z flag and the instruction will not be executed.

4.3 Branch and Branch with link (B, BL)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 9: Branch Instructions.*

Branch instructions contain a signed 2's complement 24 bit offset. This is shifted left two bits, sign extended to 32 bits, and added to the PC. The instruction can therefore specify a branch of +/- 32Mbytes. The branch offset must take account of the prefetch operation, which causes the PC to be 2 words (8 bytes) ahead of the current instruction.

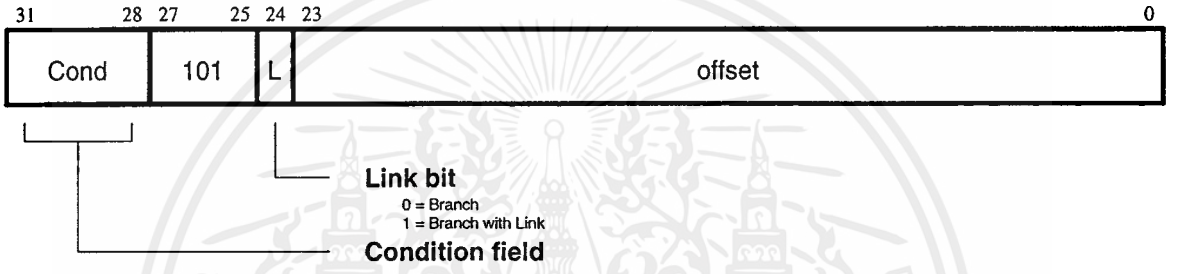


Figure 9: Branch Instructions

Branches beyond +/- 32Mbytes must use an offset or absolute destination which has been previously loaded into a register. In this case the PC should be manually saved in R14 if a Branch with Link type operation is required.

4.3.1 The link bit

Branch with Link (BL) writes the old PC into the link register (R14) of the current bank. The PC value written into R14 is adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction. Note that the CPSR is not saved with the PC.

To return from a routine called by Branch with Link use MOV PC,R14 if the link register is still valid or LDM Rn!,{..PC} if the link register has been saved onto a stack pointed to by Rn.

4.3.2 Instruction Cycle Times

Branch and Branch with Link instructions take 2S + 1N incremental cycles, where S and N are as defined in section 5.1 Cycle types on page 65.

4.3.3 Assembler syntax

B{L}{cond} <expression>

{L} is used to request the Branch with Link form of the instruction. If absent, R14 will not be affected by the instruction.

{cond} is a two-char mnemonic as shown in *Figure 8: Condition Codes* (EQ, NE, VS etc). If absent then AL (ALways) will be used.

<expression> is the destination. The assembler calculates the offset.

Items in {} are optional. Items in <> must be present.

4.3.4 Examples

```
here  BAL      here      ; assembles to 0xEAFFFFFE (note effect of PC offset)
      B        there     ; ALways condition used as default

      CMP      R1,#0     ; compare R1 with zero and branch to fred if R1
      BEQ      fred      ; was zero otherwise continue to next instruction

      BL       sub+ROM   ; call subroutine at computed address

      ADDS     R1,#1     ; add 1 to register 1, setting CPSR flags on the
      BLCC     sub       ; result then call subroutine if the C flag is clear,
                         ; which will be the case unless R1 held 0xFFFFFFFF
```

4.4 Data processing

The instruction is only executed if the condition is true, defined at the beginning of this chapter. The instruction encoding is shown in *Figure 10: Data Processing Instructions.*

The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands. The first operand is always a register (Rn). The second operand may be a shifted register (Rm) or a rotated 8 bit immediate value (Imm) according to the value of the I bit in the instruction. The condition codes in the CPSR may be preserved or updated as a result of this instruction, according to the value of the S bit in the instruction. Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd. They are used only to perform tests and to set the condition codes on the result and always have the S bit set. The instructions and their effects are listed in *Table 4: ARM Data Processing Instructions.*

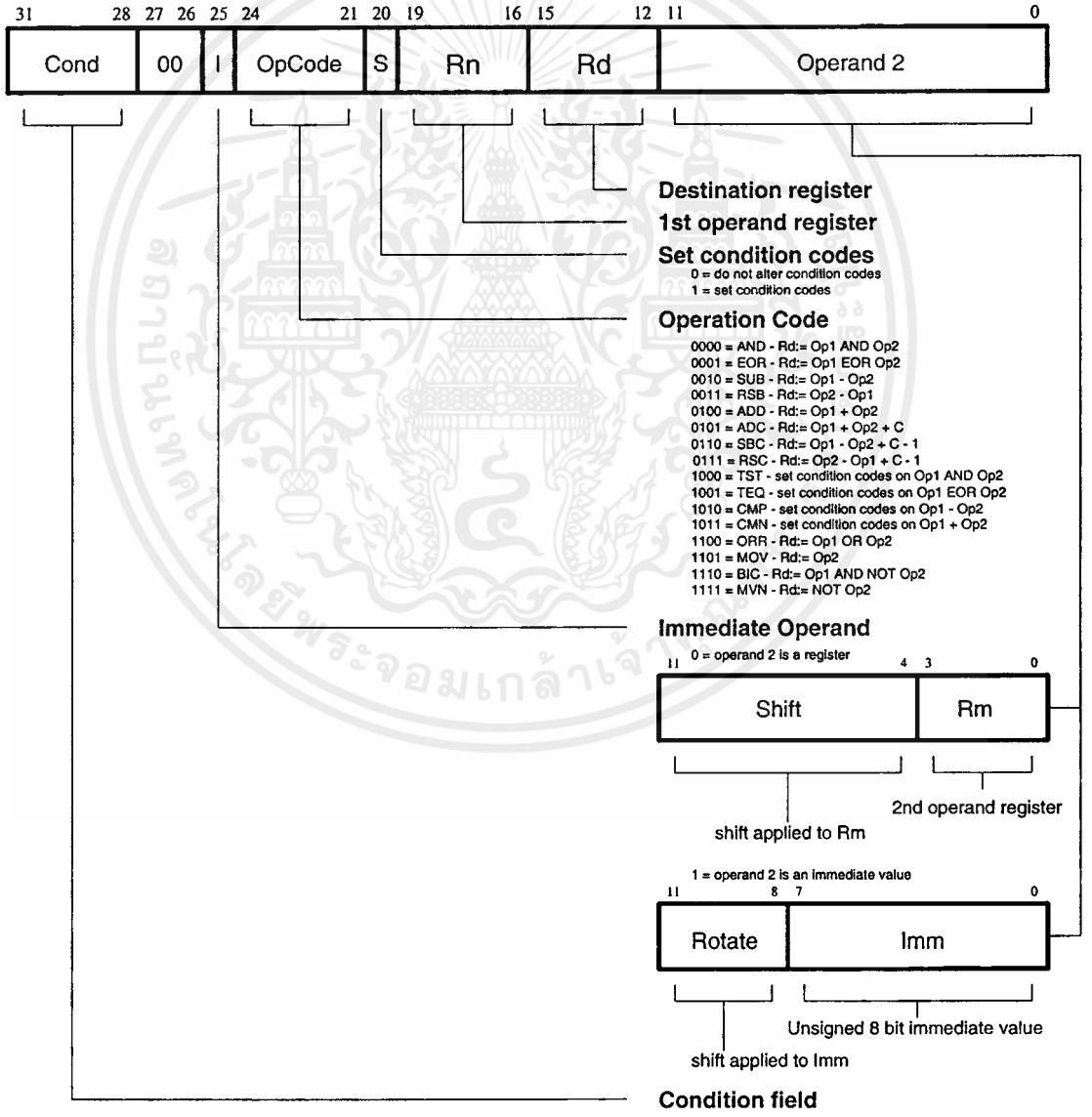


Figure 10: Data Processing Instructions

4.4.1 CPSR flags

The data processing operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15, see below) the V flag in the CPSR will be unaffected, the C flag will be set to the carry out from the barrel shifter (or preserved when the shift operation is LSL #0), the Z flag will be set if and only if the result is all zeros, and the N flag will be set to the logical value of bit 31 of the result.

AND	0000	operand1 AND operand2
EOR	0001	operand1 EOR operand2
SUB	0010	operand1 - operand2
RSB	0011	operand2 - operand1
ADD	0100	operand1 + operand2
ADC	0101	operand1 + operand2 + carry
SBC	0110	operand1 - operand2 + carry - 1
RSC	0111	operand2 - operand1 + carry - 1
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	operand1 OR operand2
MOV	1101	operand2 (operand1 is ignored)
BIC	1110	operand1 AND NOT operand2 (Bit clear)
MVN	1111	NOT operand2 (operand1 is ignored)

Table 4: ARM Data Processing Instructions

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32 bit integer (either unsigned or 2's complement signed, the two are equivalent). If the S bit is set (and Rd is not R15) the V flag in the CPSR will be set if an overflow occurs into bit 31 of the result; this may be ignored if the operands were considered unsigned, but warns of a possible error if the operands were 2's complement signed. The C flag will be set to the carry out of bit 31 of the ALU, the Z flag will be set if and only if the result was zero, and the N flag will be set to the value of bit 31 of the result (indicating a negative result if the operands are considered to be 2's complement signed).

4.4.2 Shifts

When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the Shift field in the instruction. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register (other than R15). The encoding for the different shift types is shown in *Figure 11: ARM Shift Operations*.

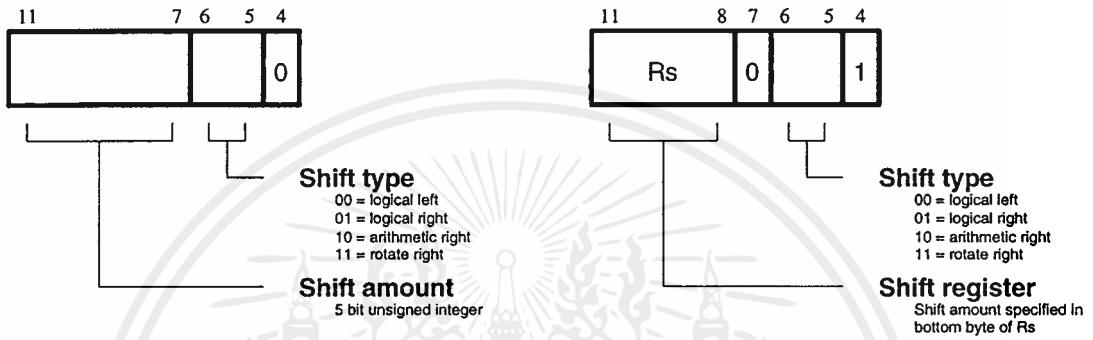


Figure 11: ARM Shift Operations

Instruction specified shift amount

When the shift amount is specified in the instruction, it is contained in a 5 bit field which may take any value from 0 to 31. A logical shift left (LSL) takes the contents of Rm and moves each bit by the specified amount to a more significant position. The least significant bits of the result are filled with zeros, and the high bits of Rm which do not map into the result are discarded, except that the least significant discarded bit becomes the shifter carry output which may be latched into the C bit of the CPSR when the ALU operation is in the logical class (see above). For example, the effect of LSL #5 is shown in *Figure 12: Logical Shift Left*.

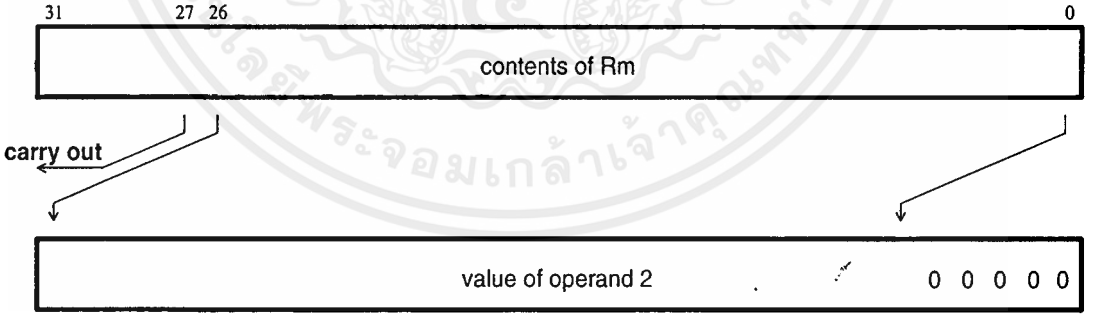


Figure 12: Logical Shift Left

Note that LSL #0 is a special case, where the shifter carry out is the old value of the CPSR C flag. The contents of Rm are used directly as the second operand.

A logical shift right (LSR) is similar, but the contents of Rm are moved to less significant positions in the result. LSR #5 has the effect shown in *Figure 13: Logical Shift Right*.

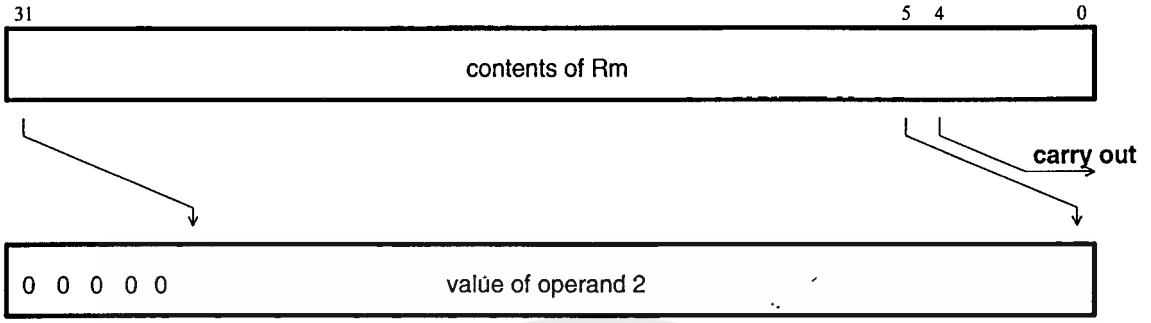


Figure 13: Logical Shift Right

The form of the shift field which might be expected to correspond to LSR #0 is used to encode LSR #32, which has a zero result with bit 31 of Rm as the carry output. Logical shift right zero is redundant as it is the same as logical shift left zero, so the assembler will convert LSR #0 (and ASR #0 and ROR #0) into LSL #0, and allow LSR #32 to be specified.

An arithmetic shift right (ASR) is similar to logical shift right, except that the high bits are filled with bit 31 of Rm instead of zeros. This preserves the sign in 2's complement notation. For example, ASR #5 is shown in *Figure 14: Arithmetic Shift Right*.

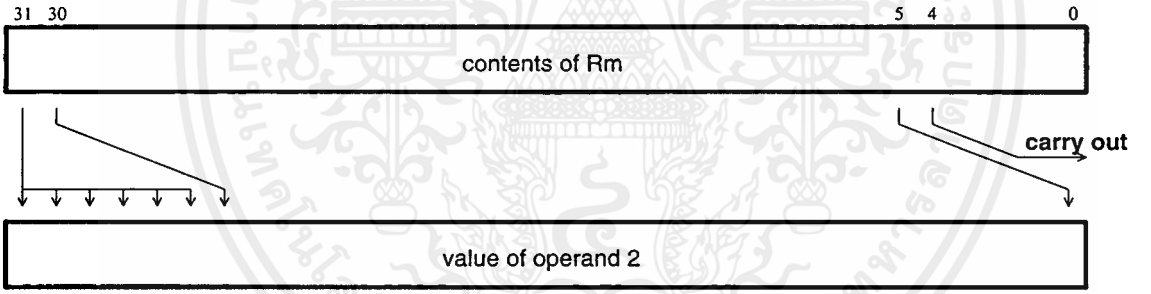


Figure 14: Arithmetic Shift Right

The form of the shift field which might be expected to give ASR #0 is used to encode ASR #32. Bit 31 of Rm is again used as the carry output, and each bit of operand 2 is also equal to bit 31 of Rm. The result is therefore all ones or all zeros, according to the value of bit 31 of Rm.

Rotate right (ROR) operations reuse the bits which 'overshoot' in a logical shift right operation by reintroducing them at the high end of the result, in place of the zeros used to fill the high end in logical right operations. For example, ROR #5 is shown in *Figure 15: Rotate Right*.

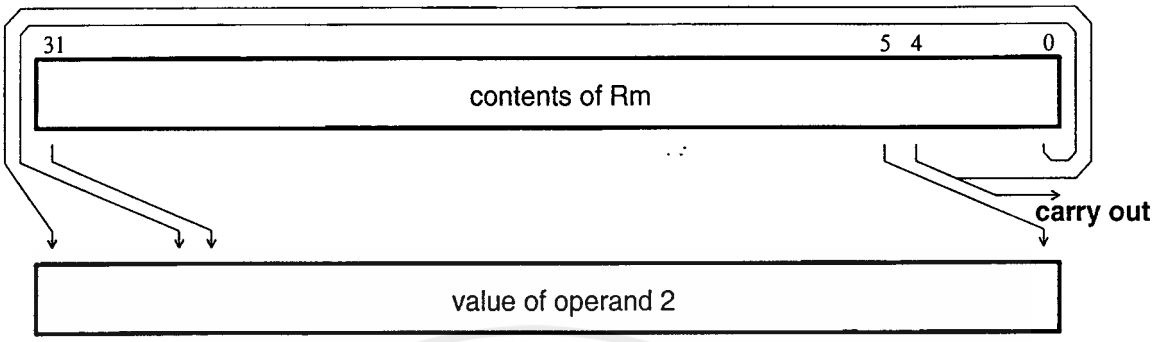


Figure 15: Rotate Right

The form of the shift field which might be expected to give ROR #0 is used to encode a special function of the barrel shifter, rotate right extended (RRX). This is a rotate right by one bit position of the 33 bit quantity formed by appending the CPSR C flag to the most significant end of the contents of Rm as shown in *Figure 16: Rotate Right Extended.*

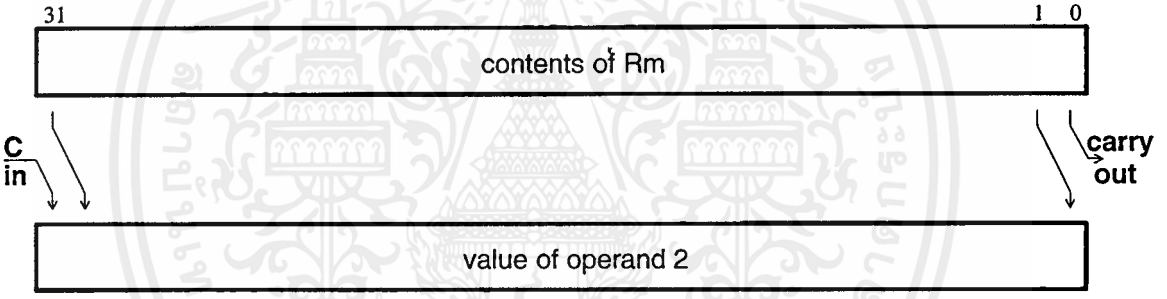


Figure 16: Rotate Right Extended

Register specified shift amount

Only the least significant byte of the contents of Rs is used to determine the shift amount. Rs can be any general register other than R15.

If this byte is zero, the unchanged contents of Rm will be used as the second operand, and the old value of the CPSR C flag will be passed on as the shifter carry output.

If the byte has a value between 1 and 31, the shifted result will exactly match that of an instruction specified shift with the same value and shift operation.

If the value in the byte is 32 or more, the result will be a logical extension of the shift described above:

(1) LSL by 32 has result zero, carry out equal to bit 0 of Rm.

(2) LSL by more than 32 has result zero, carry out zero.

(3) LSR by 32 has result zero, carry out equal to bit 31 of Rm.

(4) LSR by more than 32 has result zero, carry out zero.

(5) ASR by 32 or more has result filled with and carry out equal to bit 31 of Rm.

(6) ROR by 32 has result equal to Rm, carry out equal to bit 31 of Rm.

(7) ROR by n where n is greater than 32 will give the same result and carry out as ROR by n-32; therefore repeatedly subtract 32 from n until the amount is in the range 1 to 32 and see above.

Note that the zero in bit 7 of an instruction with a register controlled shift is compulsory; a one in this bit will cause the instruction to be a multiply or undefined instruction.

4.4.3 Immediate operand rotates

The immediate operand rotate field is a 4 bit unsigned integer which specifies a shift operation on the 8 bit immediate value. This value is zero extended to 32 bits, and then subject to a rotate right by twice the value in the rotate field. This enables many common constants to be generated, for example all powers of 2.

4.4.4 Writing to R15

When Rd is a register other than R15, the condition code flags in the CPSR may be updated from the ALU flags as described above.

When Rd is R15 and the S flag in the instruction is not set the result of the operation is placed in R15 and the CPSR is unaffected.

When Rd is R15 and the S flag is set the result of the operation is placed in R15 and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of instruction shall not be used in User mode.

4.4.5 Using R15 as an operand

If R15 (the PC) is used as an operand in a data processing instruction the register is used directly.

The PC value will be the address of the instruction, plus 8 or 12 bytes due to instruction prefetching. If the shift amount is specified in the instruction, the PC will be 8 bytes ahead. If a register is used to specify the shift amount the PC will be 12 bytes ahead.

4.4.6 TEQ, TST, CMP & CMN opcodes

These instructions do not write the result of their operation but do set flags in the CPSR. An assembler shall always set the S flag for these instructions even if it is not specified in the mnemonic.

The TEQP form of the instruction used in earlier processors shall not be used in the 32 bit modes, the PSR transfer operations should be used instead. If used in these modes, its effect is to move SPSR_<mode> to CPSR if the processor is in a privileged mode and to do nothing if in User mode.

4.4.7 Instruction Cycle Times

Data Processing instructions vary in the number of incremental cycles taken as follows:

Normal Data Processing	1S
Data Processing with register specified shift	1S + 1I
Data Processing with PC written	2S + 1N
Data Processing with register secified shift and PC written	2S + 1N + 1I

S, N and I are as defined in section 5.1 Cycle types on page 65.

4.4.8 Assembler syntax

(1) MOV,MVN - single operand instructions

<opcode>{cond}{S} Rd,<Op2>

(2) CMP,CMN,TEQ,TST - instructions which do not produce a result.

<opcode>{cond} Rn,<Op2>

(3) AND,EOR,SUB,RSB,ADD,ADC,SBC,RSC,ORR,BIC

<opcode>{cond}{S} Rd,Rn,<Op2>

where <Op2> is **Rm{,<shift>}** or,**<#expression>**

{cond} - two-character condition mnemonic, see *Figure 8: Condition Codes*

{S} - set condition codes if S present (implied for CMP, CMN, TEQ, TST).

Rd, Rn and Rm are expressions evaluating to a register number.

If <#expression> is used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.

<shift> is <shiftname> <register> or <shiftname> #expression, or RRX (rotate right one bit with extend).

<shiftname>s are: ASL, LSL, LSR, ASR, ROR. (ASL is a synonym for LSL, they assemble to the same code.)

4.4.9 Examples

```
ADDEQ    R2,R4,R5            ; if the Z flag is set make R2:=R4+R5

TEQS     R4,#3               ; test R4 for equality with 3
                             ; (the S is in fact redundant as the
                             ; assembler inserts it automatically)

SUB      R4,R5,R7,LSR R2     ; logical right shift R7 by the number in
                             ; the bottom byte of R2, subtract result
                             ; from R5, and put the answer into R4

MOV      PC,R14              ; return from subroutine

MOVS     PC,R14              ; return from exception and restore CPSR
                               from SPSR_mode
```

4.5 PSR Transfer (MRS, MSR)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter.

The MRS and MSR instructions are formed from a subset of the Data Processing operations and are implemented using the TEQ, TST, CMN and CMP instructions without the S flag set. The encoding is shown in *Figure 17: PSR Transfer*.

These instructions allow access to the CPSR and SPSR registers. The MRS instruction allows the contents of the CPSR or SPSR_<mode> to be moved to a general register. The MSR instruction allows the contents of a general register to be moved to the CPSR or SPSR_<mode> register.

The MSR instruction also allows an immediate value or register contents to be transferred to the condition code flags (N,Z,C and V) of CPSR or SPSR_<mode> without affecting the control bits. In this case, the top four bits of the specified register contents or 32 bit immediate value are written to the top four bits of the relevant PSR.

4.5.1 Operand restrictions

In User mode, the control bits of the CPSR are protected from change, so only the condition code flags of the CPSR can be changed. In other (privileged) modes the entire CPSR can be changed.

The SPSR register which is accessed depends on the mode at the time of execution. For example, only SPSR_fiq is accessible when the processor is in FIQ mode.

R15 shall not be specified as the source or destination register.

A further restriction is that no attempt shall be made to access an SPSR in User mode, since no such register exists.

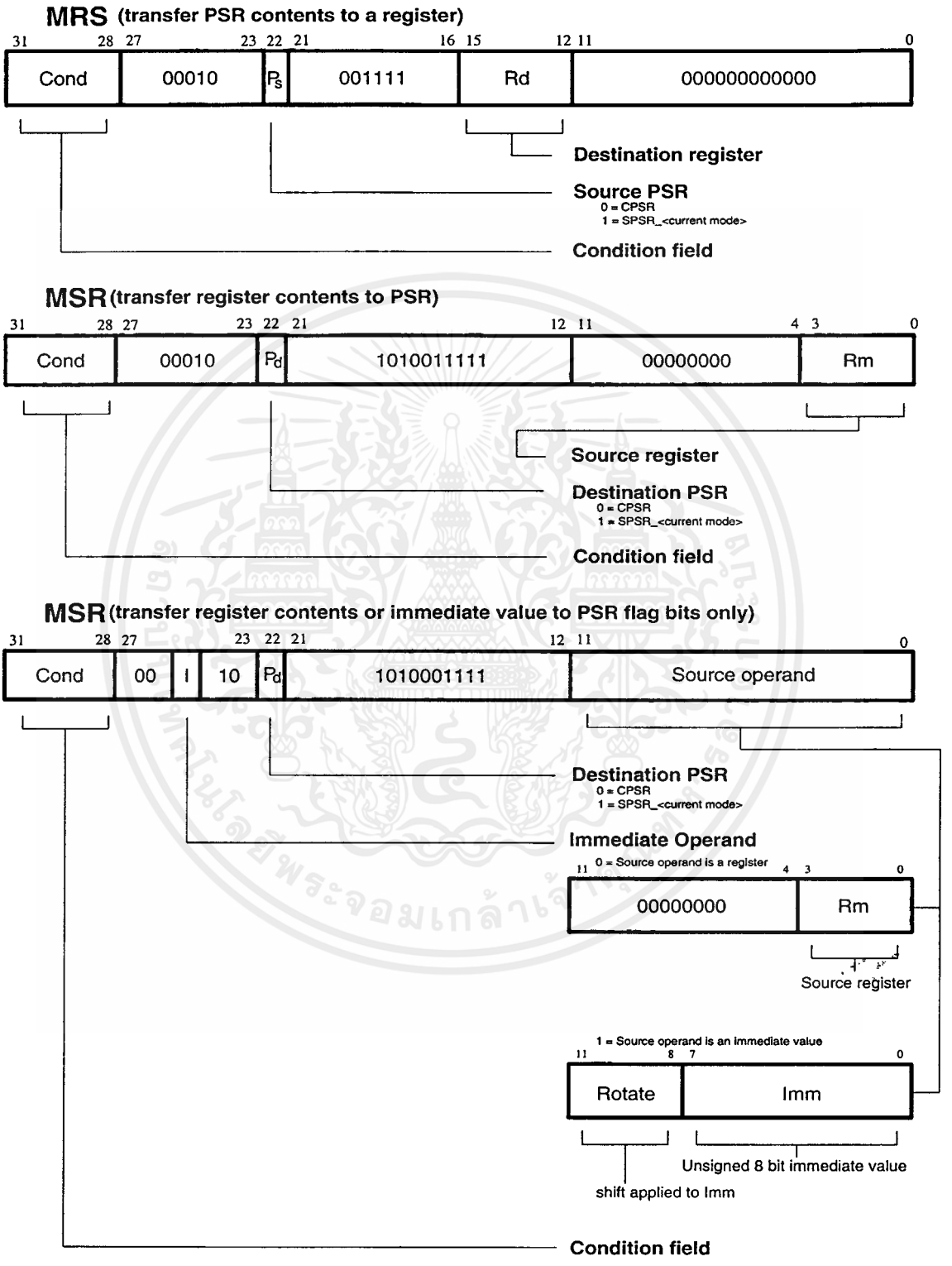


Figure 17: PSR Transfer

4.5.2 Reserved bits

Only eleven bits of the PSR are defined in ARM7 (N,Z,C,V,I,F & M[4:0]); the remaining bits (= PSR[27:8,5]) are reserved for use in future versions of the processor. To ensure the maximum compatibility between ARM7 programs and future processors, the following rules should be observed:

(1) The reserved bits shall be preserved when changing the value in a PSR.

(2) Programs shall not rely on specific values from the reserved bits when checking the PSR status, since they may read as one or zero in future processors.

A read-modify-write strategy should therefore be used when altering the control bits of any PSR register; this involves transferring the appropriate PSR register to a general register using the MRS instruction, changing only the relevant bits and then transferring the modified value back to the PSR register using the MSR instruction.

e.g. The following sequence performs a mode change:

```
MRS     R0,CPSR                 ; take a copy of the CPSR
BIC     R0,R0,#0x1F             ; clear the mode bits
ORR     R0,R0,#new_mode         ; select new mode
MSR     CPSR,R0                 ; write back the modified CPSR
```

When the aim is simply to change the condition code flags in a PSR, a value can be written directly to the flag bits without disturbing the control bits. e.g. The following instruction sets the N,Z,C & V flags:

```
MSR   CPSR_flg,#0xF0000000      ; set all the flags regardless of
                                ; their previous state (does not
                                ; affect any control bits)
```

No attempt shall be made to write an 8 bit immediate value into the whole PSR since such an operation cannot preserve the reserved bits.

4.5.3 Instruction Cycle Times

PSR Transfers take 1S incremental cycles, where S is as defined in section 5.1 Cycle types on page 65.

4.5.4 Assembler syntax

(1) MRS - transfer PSR contents to a register

MRS{cond} Rd,<psr>

(2) MSR - transfer register contents to PSR

MSR{cond} <psr>,Rm

(3) MSR - transfer register contents to PSR flag bits only

MSR{cond} <psrf>,Rm

The most significant four bits of the register contents are written to the N,Z,C & V flags respectively.

(4) MSR - transfer immediate value to PSR flag bits only

MSR{cond} <psrf>,<#expression>

The expression should symbolise a 32 bit value of which the most significant four bits are written to the N,Z,C & V flags respectively.

{cond} - two-character condition mnemonic, see *Figure 8: Condition Codes*

Rd and Rm are expressions evaluating to a register number other than R15

<psr> is CPSR, CPSR_all, SPSR or SPSR_all. (CPSR and CPSR_all are synonyms as are SPSR and SPSR_all)

<psrf> is CPSR_flg or SPSR_flg

Where <#expression> is used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.

4.5.5 Examples

In User mode the instructions behave as follows:

```
MSR     CPSR_all,Rm           ; CPSR[31:28] <- Rm[31:28]
MSR     CPSR_flg,Rm           ; CPSR[31:28] <- Rm[31:28]

MSR     CPSR_flg,#0xA0000000  ; CPSR[31:28] <- 0xA
                              ;   (i.e. set N,C; clear Z,V)

MRS     Rd,CPSR               ; Rd[31:0] <- CPSR[31:0]
```

In privileged modes the instructions behave as follows:

```
MSR     CPSR_all,Rm           ; CPSR[31:0]  <- Rm[31:0]
MSR     CPSR_flg,Rm           ; CPSR[31:28] <- Rm[31:28]

MSR     CPSR_flg,#0x50000000  ; CPSR[31:28] <- 0x5
                              ;   (i.e. set Z,V; clear N,C)

MRS     Rd,CPSR               ; Rd[31:0] <- CPSR[31:0]

MSR     SPSR_all,Rm           ; SPSR_<mode>[31:0]  <- Rm[31:0]
MSR     SPSR_flg,Rm           ; SPSR_<mode>[31:28] <- Rm[31:28]

MSR     SPSR_flg,#0xC0000000  ; SPSR_<mode>[31:28] <- 0xC
                              ;   (i.e. set N,Z; clear C,V)

MRS     Rd,SPSR               ; Rd[31:0] <- SPSR_<mode>[31:0]
```

4.6 Multiply and Multiply-Accumulate (MUL, MLA)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 18: Multiply Instructions*.

The multiply and multiply-accumulate instructions use a 2 bit Booth's algorithm to perform integer multiplication. They give the least significant 32 bits of the product of two 32 bit operands, and may be used to synthesize higher precision multiplications.

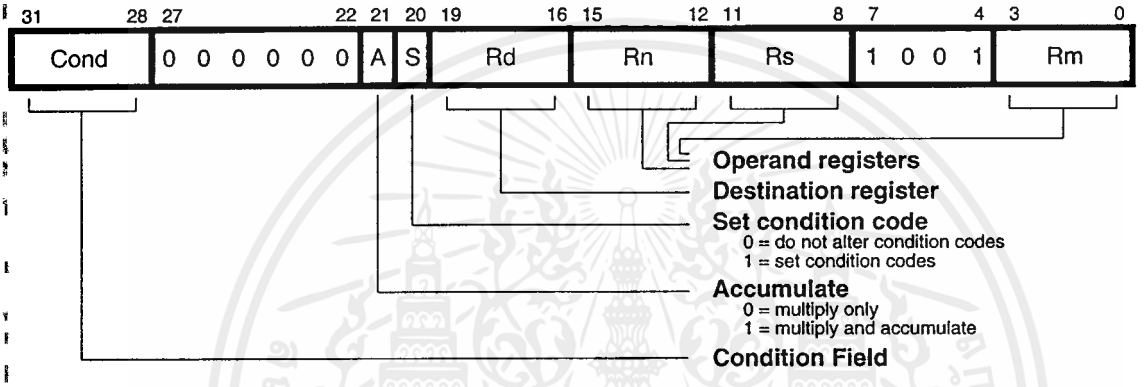


Figure 18: Multiply Instructions

The multiply form of the instruction gives Rd:=Rm*Rs. Rn is ignored, and should be set to zero for compatibility with possible future upgrades to the instruction set.

The multiply-accumulate form gives Rd:=Rm*Rs+Rn, which can save an explicit ADD instruction in some circumstances.

The results of a signed multiply and of an unsigned multiply of 32 bit operands differ only in the upper 32 bits - the low 32 bits of the signed and unsigned results are identical. As these instructions only produce the low 32 bits of a multiply, they can be used for both signed and unsigned multiplies.

For example consider the multiplication of the operands:

Operand A	Operand B	Result
0xFFFFFFF6	0x00000014	0xFFFFFF38

If the operands are interpreted as signed, operand A has the value -10, operand B has the value 20, and the result is -200 which is correctly represented as 0xFFFFFF38

If the operands are interpreted as unsigned, operand A has the value 4294967286, operand B has the value 20 and the result is 85899345720, which is represented as 0x13FFFFFF38, so the least significant 32 bits are 0xFFFFFF38.

4.6.1 Operand restrictions

Due to the way multiplication was implemented in other ARM processors, certain combinations of operand registers should be avoided. The ARM7's advanced multiplier can handle all operand combinations but by observing these restrictions code written for the ARM7 will be more compatible with other ARM processors. (The assembler will issue a warning if these restrictions are overlooked.)

The destination register Rd shall not be the same as the operand register Rm. R15 shall not be used as an operand or as the destination register.

All other register combinations will give correct results, and Rd, Rn and Rs may use the same register when required.

4.6.2 CPSR flags

Setting the CPSR flags is optional, and is controlled by the S bit in the instruction. The N (Negative) and Z (Zero) flags are set correctly on the result (N is made equal to bit 31 of the result, and Z is set if and only if the result is zero). The C (Carry) flag is set to a meaningless value and the V (oVerflow) flag is unaffected.

4.6.3 Instruction Cycle Times

The Multiply instructions take 1S + mI cycles to execute, where S and I are as defined in section 5.1 Cycle types on page 65.

m is the number of cycles required by the multiply algorithm, which is determined by the contents of Rs. Multiplication by any number between *2^(2m-3)* and *2^(2m-1)-1* takes 1S+mI *m* cycles for *1<m>16*. Multiplication by 0 or 1 takes 1S+1I cycles, and multiplication by any number greater than or equal to 2^(29) takes 1S+16I cycles. The maximum time for any multiply is thus 1S+16I cycles.

4.6.4 Assembler syntax

MUL{cond}{S} Rd,Rm,Rs

MLA{cond}{S} Rd,Rm,Rs,Rn

{cond} - two-character condition mnemonic, see *Figure 8: Condition Codes*

{S} - set condition codes if S present

Rd, Rm, Rs and Rn are expressions evaluating to a register number other than R15.

4.6.5 Examples

```
MUL       R1,R2,R3              ; R1:=R2*R3
MLAEQS    R1,R2,R3,R4           ; conditionally R1:=R2*R3+R4,
                                ; setting condition codes
```

4.7 Single data transfer (LDR, STR)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 19: Single Data Transfer Instructions.*

The single data transfer instructions are used to load or store single bytes or words of data. The memory address used in the transfer is calculated by adding an offset to or subtracting an offset from a base register. The result of this calculation may be written back into the base register if `auto-indexing' is required.

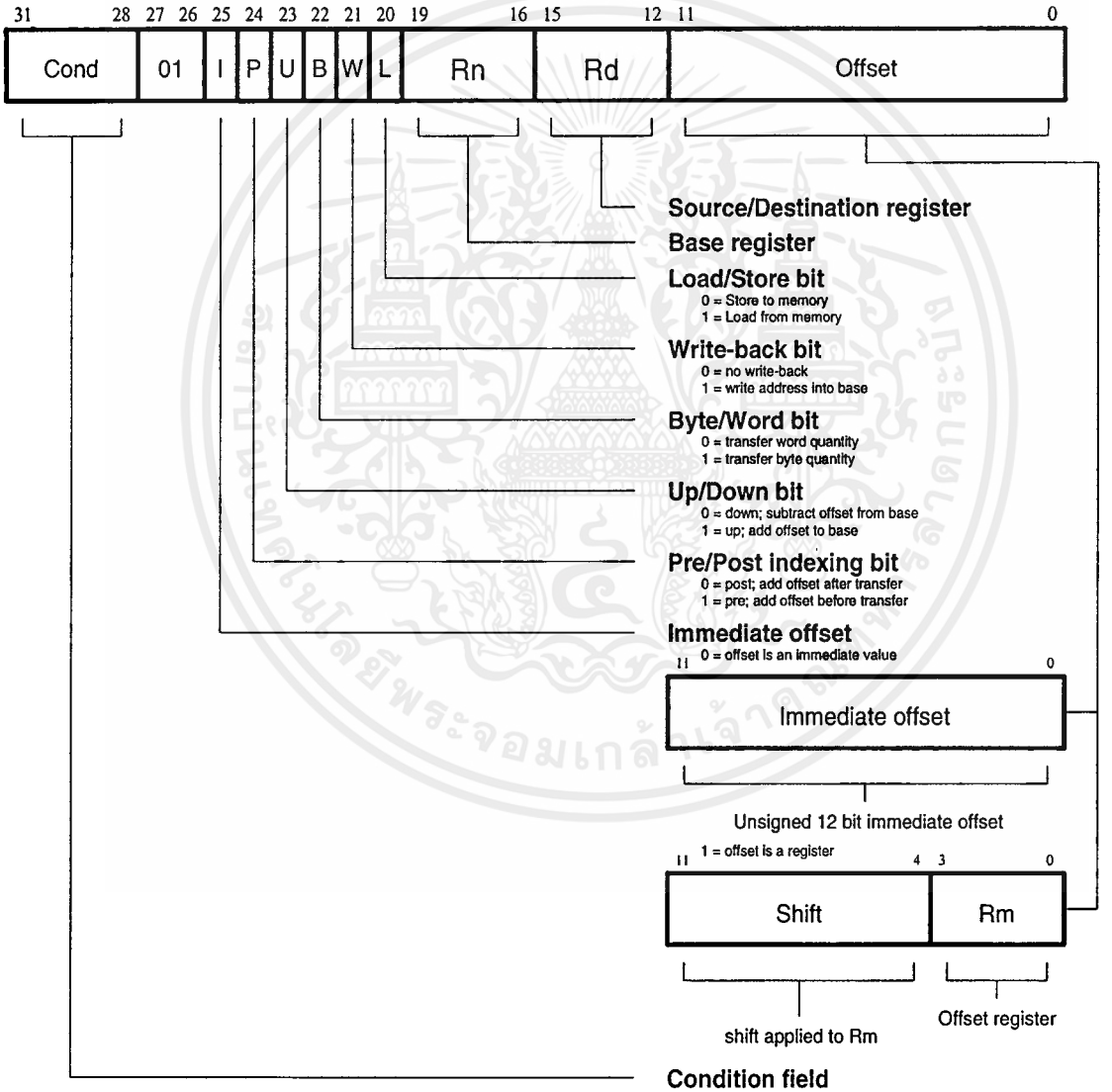


Figure 19: Single Data Transfer Instructions

4.7.1 Offsets and auto-indexing

The offset from the base may be either a 12 bit unsigned binary immediate value in the instruction, or a second register (possibly shifted in some way). The offset may be added to (U=1) or subtracted from (U=0) the base register Rn. The offset modification may be performed either before (pre-indexed, P=1) or after (post-indexed, P=0) the base is used as the transfer address.

The W bit gives optional auto increment and decrement addressing modes. The modified base value may be written back into the base (W=1), or the old base value may be kept (W=0). In the case of post-indexed addressing, the write back bit is redundant and is always set to zero, since the old base value can be retained by setting the offset to zero. Therefore post-indexed data transfers always write back the modified base. The only use of the W bit in a post-indexed data transfer is in privileged mode code, where setting the W bit forces non-privileged mode for the transfer, allowing the operating system to generate a user address in a system where the memory management hardware makes suitable use of this hardware.

4.7.2 Shifted register offset

The 8 shift control bits are described in the data processing instructions section. However, the register specified shift amounts are not available in this instruction class. See *4.4.2 Shifts*.

4.7.3 Bytes and words

This instruction class may be used to transfer a byte (B=1) or a word (B=0) between an ARM7 register and memory.

The action of LDR(B) and STR(B) instructions is influenced by the **BIGEND** control signal. The two possible configurations are described below.

Little Endian Configuration

A byte load (LDRB) expects the data on data bus inputs 7 through 0 if the supplied address is on a word boundary, on data bus inputs 15 through 8 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register, and the remaining bits of the register are filled with zeros. Please see *Figure 3: Little Endian addresses of bytes within words.*

A byte store (STRB) repeats the bottom 8 bits of the source register four times across data bus outputs 31 through 0. The external memory system should activate the appropriate byte subsystem to store the data.

A word load (LDR) will normally use a word aligned address. However, an address offset from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 0 to 7. This means that half-words accessed at offsets 0 and 2 from the word boundary will be correctly loaded into bits 0 through 15 of the register. Two shift operations are then required to clear or to sign extend the upper 16 bits. This is illustrated in *Figure 20: Little Endian Offset Addressing.*

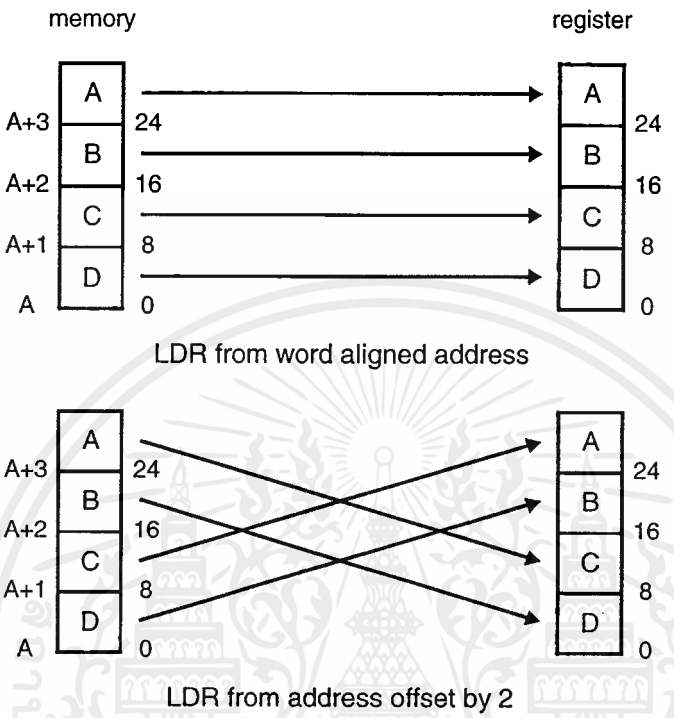


Figure 20: Little Endian Offset Addressing

A word store (STR) should generate a word aligned address. The word presented to the data bus is not affected if the address is not word aligned. That is, bit 31 of the register being stored always appears on data bus output 31.

Big Endian Configuration

A byte load (LDRB) expects the data on data bus inputs 31 through 24 if the supplied address is on a word boundary, on data bus inputs 23 through 16 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register and the remaining bits of the register are filled with zeros. Please see *Figure 4: Big Endian addresses of bytes within words.*

A byte store (STRB) repeats the bottom 8 bits of the source register four times across data bus outputs 31 through 0. The external memory system should activate the appropriate byte subsystem to store the data.

A word load (LDR) should generate a word aligned address. An address offset of 0 or 2 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 31 through 24. This means that half-words accessed at these offsets will be correctly loaded into bits 16 through 31 of the register. A shift operation is then required to move (and optionally sign extend) the data into the bottom 16 bits. An address offset of 1 or 3 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 15 through 8.

A word store (STR) should generate a word aligned address. The word presented to the data bus is not affected if the address is not word aligned. That is, bit 31 of the register being stored always appears on data bus output 31.

4.7.4 Use of R15

Write-back shall not be specified if R15 is specified as the base register (Rn). When using R15 as the base register you must remember it contains an address 8 bytes on from the address of the current instruction.

R15 shall not be specified as the register offset (Rm).

When R15 is the source register (Rd) of a register store (STR) instruction, the stored value will be address of the instruction plus 12.

4.7.5 Restriction on the use of base register

When configured for late aborts, the following example code is difficult to unwind as the base register, Rn, gets updated before the abort handler starts. Sometimes it may be impossible to calculate the initial value.

For example:

```
LDR    R0,[R1],R1
```

<LDR|STR> Rd, [Rn],{+/-}Rn{,<shift>}

Therefore a post-indexed LDR|STR where Rm is the same register as Rn shall not be used.

4.7.6 Data Aborts

A transfer to or from a legal address may cause problems for a memory management system. For instance, in a system which uses virtual memory the required data may be absent from main memory. The memory manager can signal a problem by taking the processor ABORT input HIGH whereupon the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

4.7.7 Instruction Cycle Times

Normal LDR instructions take 1S + 1N + 1I and LDR PC take 2S + 2N +1I incremental cycles, where S,N and I are as defined in section 5.1 Cycle types on page 65.

STR instructions take 2N incremental cycles to execute.

4.7.8 Assembler syntax

<LDR|STR>{cond}{B}{T} Rd,<Address>

LDR - load from memory into a register

STR - store from a register into memory

{cond} - two-character condition mnemonic, see *Figure 8: Condition Codes*

{B} - if B is present then byte transfer, otherwise word transfer

{T} - if T is present the W bit will be set in a post-indexed instruction, forcing non-privileged mode for the transfer cycle. T is not allowed when a pre-indexed addressing mode is specified or implied.

Rd is an expression evaluating to a valid register number.

<Address> can be:

(i) An expression which generates an address:

<expression>

The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.

(ii) A pre-indexed addressing specification:

[Rn] offset of zero

[Rn,<#expression>]{!} offset of <expression> bytes

[Rn,{+/-}Rm{,<shift>}]{!} offset of +/- contents of index register, shifted by <shift>

(iii) A post-indexed addressing specification:

[Rn],<#expression> offset of <expression> bytes

[Rn],{+/-}Rm{,<shift>} offset of +/- contents of index register, shifted as by <shift>.

Rn and Rm are expressions evaluating to a register number. If Rn is R15 then the assembler will subtract 8 from the offset value to allow for ARM7 pipelining. In this case base write-back shall not be specified.

<shift> is a general shift operation (see section on data processing instructions) but note that the shift amount may not be specified by a register.

{!} writes back the base register (set the W bit) if ! is present.

4.7.9 Examples

```
STR     R1,[R2,R4]!          ; store R1 at R2+R4 (both of which are
                             ; registers) and write back address to R2

STR     R1,[R2],R4           ; store R1 at R2 and write back
                             ; R2+R4 to R2

LDR     R1,[R2,#16]          ; load R1 from contents of R2+16
                             ;  Don't write back

LDR     R1,[R2,R3,LSL#2]     ; load R1 from contents of R2+R3*4
```

```
LDREQB    R1,[R6,#5]          ; conditionally load byte at R6+5 into
                              ;  R1 bits 0 to 7, filling bits 8 to 31
                              ;  with zeros

STR       R1,PLACE            ; generate PC relative offset to address
          •                   ;  PLACE
          •
PLACE
```

4.8 Block data transfer (LDM, STM)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 21: Block Data Transfer Instructions.*

Block data transfer instructions are used to load (LDM) or store (STM) any subset of the currently visible registers. They support all possible stacking modes, maintaining full or empty stacks which can grow up or down memory, and are very efficient instructions for saving or restoring context, or for moving large blocks of data around main memory.

4.8.1 The register list

The instruction can cause the transfer of any registers in the current bank (and non-user mode programs can also transfer to and from the user bank, see below). The register list is a 16 bit field in the instruction, with each bit corresponding to a register. A 1 in bit 0 of the register field will cause R0 to be transferred, a 0 will cause it not to be transferred; similarly bit 1 controls the transfer of R1, and so on.

Any subset of the registers, or all the registers, may be specified. The only restriction is that the register list should not be empty.

Whenever R15 is stored to memory the stored value is the address of the STM instruction plus 12.

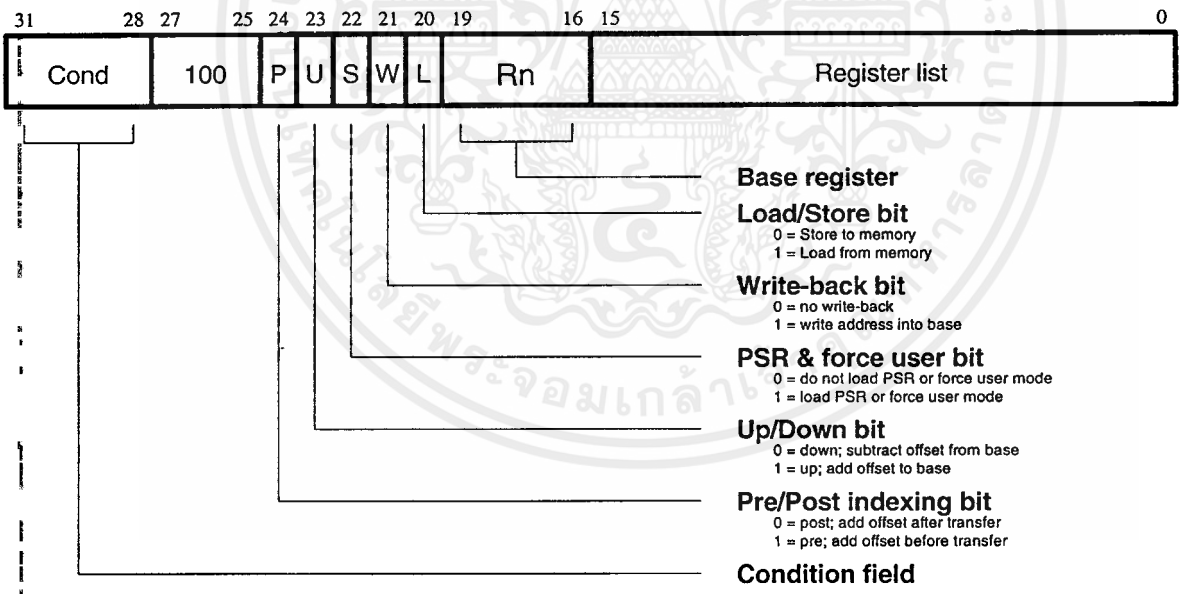


Figure 21: Block Data Transfer Instructions

4.8.2 Addressing modes

The transfer addresses are determined by the contents of the base register (Rn), the pre/post bit (P) and the up/down bit (U). The registers are transferred in the order lowest to highest, so R15 (if in the list) will always be transferred last. The lowest register also gets transferred to/from the lowest memory address. By way of illustration, consider the transfer of R1, R5 and R7 in the case where Rn=0x1000 and write back of

the modified base is required (W=1). *Figure 22: Post-increment addressing*, *Figure 23: Pre-increment addressing*, *Figure 24: Post-decrement addressing* and *Figure 25: Pre-decrement addressing* show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

In all cases, had write back of the modified base not been required (W=0), Rn would have retained its initial value of 0x1000 unless it was also in the transfer list of a load multiple register instruction, when it would have been overwritten with the loaded value.

4.8.3 Address Alignment

The address should normally be a word aligned quantity and non-word aligned addresses do not affect the instruction. However, the bottom 2 bits of the address will appear on A[1:0] and might be interpreted by the memory system.

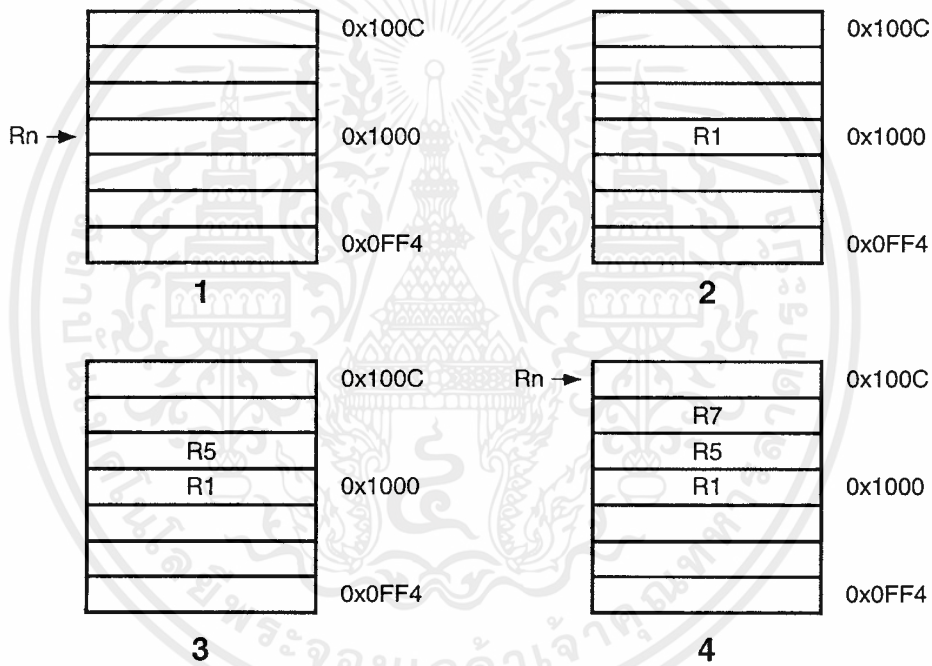


Figure 22: Post-increment addressing

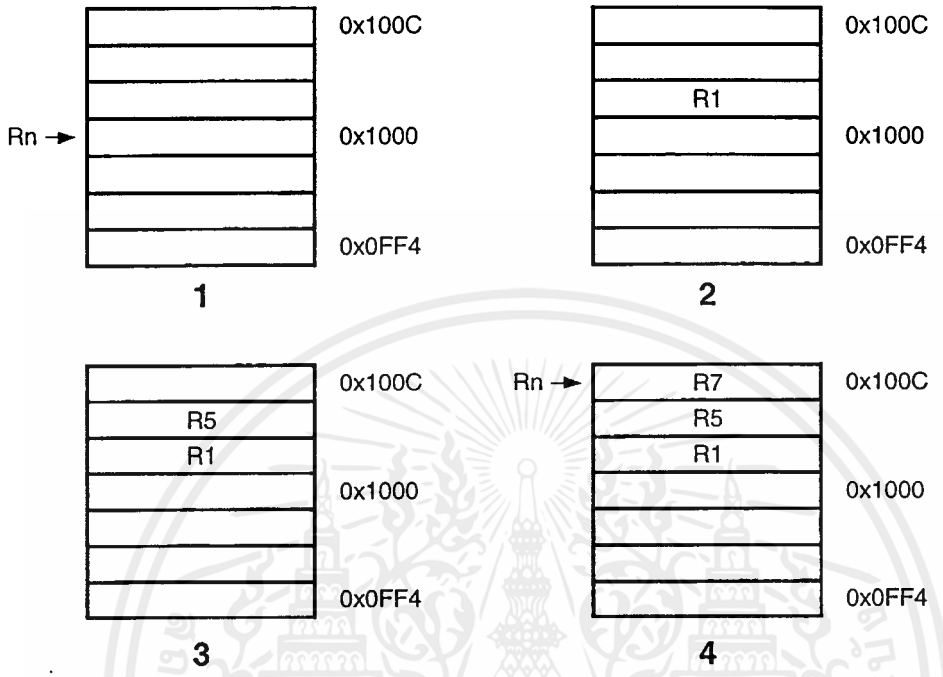


Figure 23: Pre-increment addressing

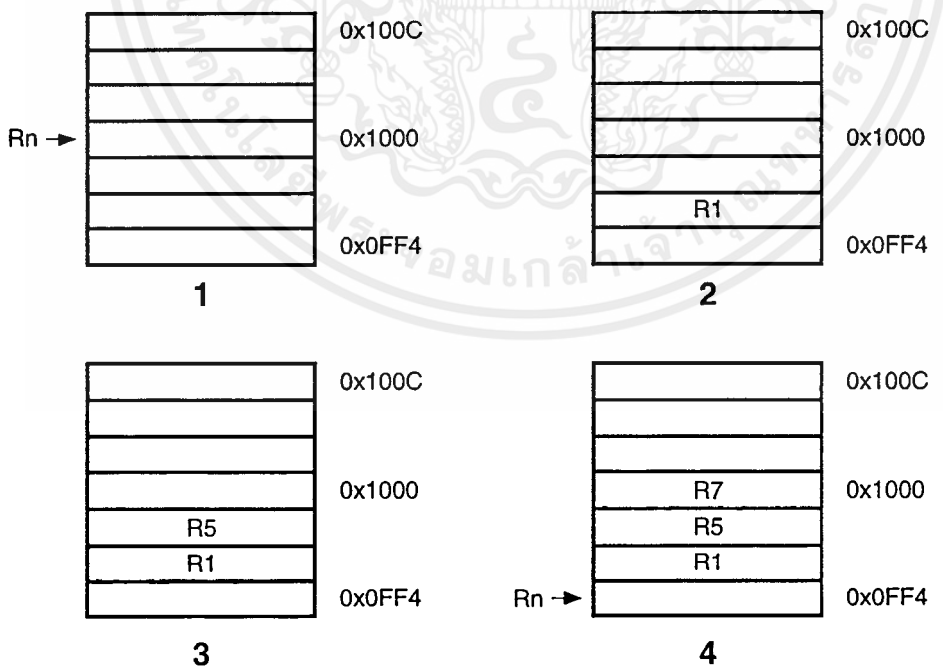


Figure 24: Post-decrement addressing

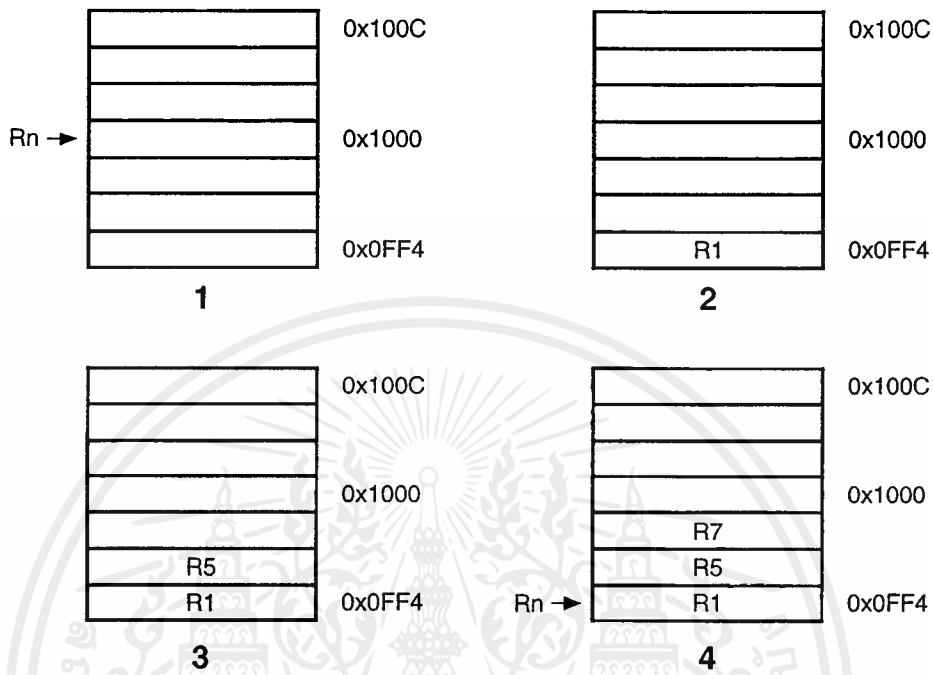


Figure 25: Pre-decrement addressing

4.8.4 Use of the S bit

When the S bit is set in a LDM/STM instruction its meaning depends on whether or not R15 is in the transfer list and on the type of instruction. The S bit should only be set if the instruction is to execute in a privileged mode.

LDM with R15 in transfer list and S bit set (Mode changes)

If the instruction is a LDM then SPSR_<mode> is transferred to CPSR at the same time as R15 is loaded.

STM with R15 in transfer list and S bit set (User bank transfer)

The registers transferred are taken from the User bank rather than the bank corresponding to the current mode. This is useful for saving the user state on process switches. Base write-back shall not be used when this mechanism is employed.

R15 not in list and S bit set (User bank transfer)

For both LDM and STM instructions, the User bank registers are transferred rather than the register bank corresponding to the current mode. This is useful for saving the user state on process switches. Base write-back shall not be used when this mechanism is employed.

When the instruction is LDM, care must be taken not to read from a banked register during the following cycle (inserting a NOP after the LDM will ensure safety).

4.8.5 Use of R15 as the base

R15 shall not be used as the base register in any LDM or STM instruction.

4.8.6 Inclusion of the base in the register list

When write-back is specified, the base is written back at the end of the second cycle of the instruction. During a STM, the first register is written out at the start of the second cycle. A STM which includes storing the base, with the base as the first register to be stored, will therefore store the unchanged value, whereas with the base second or later in the transfer order, will store the modified value. A LDM will always overwrite the updated base if the base is in the list.

4.8.7 Data Aborts

Some legal addresses may be unacceptable to a memory management system, and the memory manager can indicate a problem with an address by taking the ABORT signal HIGH. This can happen on any transfer during a multiple register load or store, and must be recoverable if ARM7 is to be used in a virtual memory system.

Aborts during STM instructions

If the abort occurs during a store multiple instruction, ARM7 takes little action until the instruction completes, whereupon it enters the data abort trap. The memory manager is responsible for preventing erroneous writes to the memory. The only change to the internal state of the processor will be the modification of the base register if write-back was specified, and this must be reversed by software (and the cause of the abort resolved) before the instruction may be retried.

Aborts during LDM instructions

When ARM7 detects a data abort during a load multiple instruction, it modifies the operation of the instruction to ensure that recovery is possible.

(i) Overwriting of registers stops when the abort happens. The aborting load will not take place but earlier ones may have overwritten registers. The PC is always the last register to be written and so will always be preserved.

(ii) The base register is restored, to its modified value if write-back was requested. This ensures recoverability in the case where the base register is also in the transfer list, and may have been overwritten before the abort occurred.

The data abort trap is taken when the load multiple has completed, and the system software must undo any base modification (and resolve the cause of the abort) before restarting the instruction.

4.8.8 Instruction Cycle Times

Normal LDM instructions take nS + 1N + 1I and LDM PC takes (n+1)S + 2N + 1I incremental cycles, where S,N and I are as defined in section 5.1 Cycle types on page 65.

STM instructions take (n-1)S + 2N incremental cycles to execute.

n is the number of words transferred.

4.8.9 Assembler syntax

<LDM | STM>{cond}<FD | ED | FA | EA | IA | IB | DA | DB> Rn{!},<Rlist>{^}

{cond} - two character condition mnemonic, see *Figure 8: Condition Codes*

Rn is an expression evaluating to a valid register number

<Rlist> is a list of registers and register ranges enclosed in {} (eg {R0,R2-R7,R10}).

{!} if present requests write-back (W=1), otherwise W=0

{^} if present set S bit to load the CPSR along with the PC, or force transfer of user bank when in privileged mode

Addressing mode names

There are different assembler mnemonics for each of the addressing modes, depending on whether the instruction is being used to support stacks or for other purposes. The equivalences between the names and the values of the bits in the instruction are shown in the following table:

pre-increment load	LDMED	LDMIB	1	1	1
post-increment load	LDMFD	LDMIA	1	0	1
pre-decrement load	LDMEA	LDMDB	1	1	0
post-decrement load	LDMFA	LDMDA	1	0	0
pre-increment store	STMFA	STMIB	0	1	1
post-increment store	STMEA	STMIA	0	0	1
pre-decrement store	STMFD	STMDB	0	1	0
post-decrement store	STMED	STMDA	0	0	0

Table 5: Addressing Mode Names

FD, ED, FA, EA define pre/post indexing and the up/down bit by reference to the form of stack required. The F and E refer to a "full" or "empty" stack, i.e. whether a pre-index has to be done (full) before storing to the stack. The A and D refer to whether the stack is ascending or descending. If ascending, a STM will go up and LDM down, if descending, vice-versa.

IA, IB, DA, DB allow control when LDM/STM are not being used for stacks and simply mean Increment After, Increment Before, Decrement After, Decrement Before.

4.8.10 Examples

```
LDMFD    SP!,{R0,R1,R2}        ; unstack 3 registers

STMIA    R0,{R0-R15}           ; save all registers

LDMFD    SP!,{R15}             ; R15 <- (SP),CPSR unchanged
LDMFD    SP!,{R15}^            ; R15 <- (SP), CPSR <- SPSR_mode (allowed
                               ;  only in privileged modes)
STMFD    R13,{R0-R14}^         ; Save user mode regs on stack (allowed
                               ;  only in privileged modes)
```

These instructions may be used to save state on subroutine entry, and restore it efficiently on return to the calling routine:

```
STMED    SP!,{R0-R3,R14}       ; save R0 to R3 to use as workspace
                               ;  and R14 for returning

BL       somewhere             ; this nested call will overwrite R14

LDMED    SP!,{R0-R3,R15}       ; restore workspace and return
```

4.9 Single data swap (SWP)

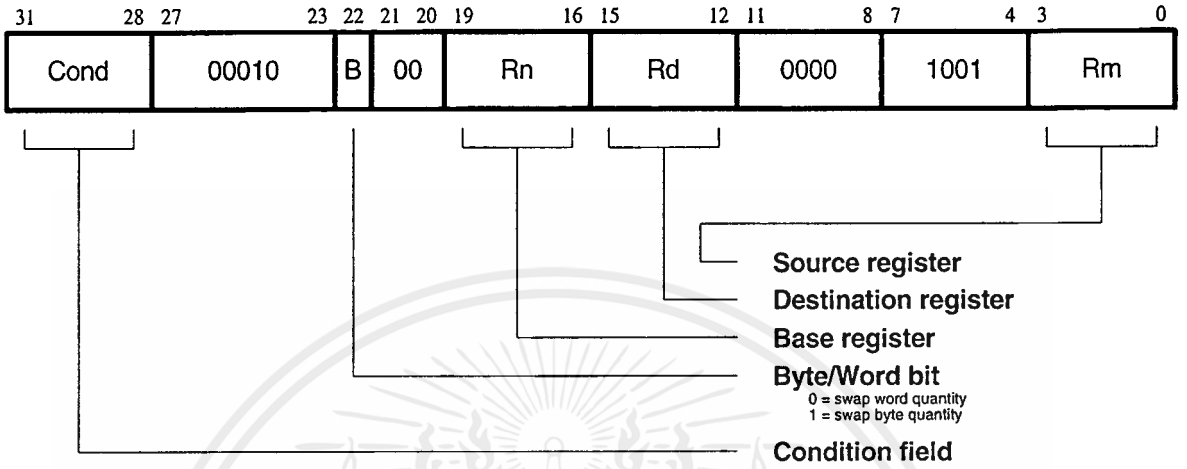


Figure 26: Swap Instruction

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 26: Swap Instruction.*

The data swap instruction is used to swap a byte or word quantity between a register and external memory. This instruction is implemented as a memory read followed by a memory write which are "locked" together (the processor cannot be interrupted until both operations have completed, and the memory manager is warned to treat them as inseparable). This class of instruction is particularly useful for implementing software semaphores.

The swap address is determined by the contents of the base register (Rn). The processor first reads the contents of the swap address. Then it writes the contents of the source register (Rm) to the swap address, and stores the old memory contents in the destination register (Rd). The same register may be specified as both the source and destination.

The LOCK output goes HIGH for the duration of the read and write operations to signal to the external memory manager that they are locked together, and should be allowed to complete without interruption. This is important in multi-processor systems where the swap instruction is the only indivisible instruction which may be used to implement semaphores; control of the memory must not be removed from a processor while it is performing a locked operation.

4.9.1 Bytes and words

This instruction class may be used to swap a byte (B=1) or a word (B=0) between an ARM7 register and memory. The SWP instruction is implemented as a LDR followed by a STR and the action of these is as described in the section on single data transfers. In particular, the description of Big and Little Endian configuration applies to the SWP instruction.

4.9.2 Use of R15

R15 shall not be used as an operand (Rd, Rn or Rs) in a SWP instruction.

4.9.3 Data Aborts

If the address used for the swap is unacceptable to a memory management system, the internal MMU or external memory manager can flag the problem by driving ABORT HIGH. This can happen on either the read or the write cycle (or both), and in either case, the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

4.9.4 Instruction Cycle Times

Swap instructions take 1S + 2N +1I incremental cycles to execute, where S,N and I are as defined in section 5.1 Cycle types on page 65.

4.9.5 Assembler syntax

<SWP>{cond}{B} Rd,Rm,[Rn]

{cond} - two-character condition mnemonic, see *Figure 8: Condition Codes*

{B} - if B is present then byte transfer, otherwise word transfer

Rd,Rm,Rn are expressions evaluating to valid register numbers

4.9.6 Examples

```
SWP     R0,R1,[R2]          ; load R0 with the word addressed by R2, and
                            ; store R1 at R2

SWPB    R2,R3,[R4]          ; load R2 with the byte addressed by R4, and
                            ; store bits 0 to 7 of R3 at R4

SWPEQ   R0,R0,[R1]          ; conditionally swap the contents of R1
                            ; with R0
```

4.10 Software interrupt (SWI)

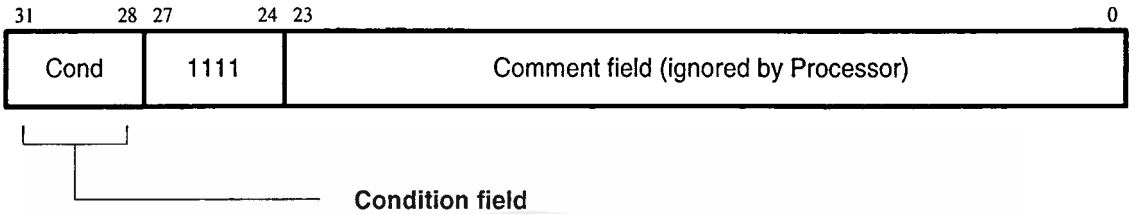


Figure 27: Software Interrupt Instruction

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 27: Software Interrupt Instruction.*

The software interrupt instruction is used to enter Supervisor mode in a controlled manner. The instruction causes the software interrupt trap to be taken, which effects the mode change. The PC is then forced to a fixed value (0x08) and the CPSR is saved in SPSR_svc. If the SWI vector address is suitably protected (by external memory management hardware) from modification by the user, a fully protected operating system may be constructed.

4.10.1 Return from the supervisor

The PC is saved in R14_svc upon entering the software interrupt trap, with the PC adjusted to point to the word after the SWI instruction. MOVS PC,R14_svc will return to the calling program and restore the CPSR.

Note that the link mechanism is not re-entrant, so if the supervisor code wishes to use software interrupts within itself it must first save a copy of the return address and SPSR.

4.10.2 Comment field

The bottom 24 bits of the instruction are ignored by the processor, and may be used to communicate information to the supervisor code. For instance, the supervisor may look at this field and use it to index into an array of entry points for routines which perform the various supervisor functions.

4.10.3 Instruction Cycle Times

Software interrupt instructions take 2S + 1N incremental cycles to execute, where S and N are as defined in section 5.1 Cycle types on page 65.

4.10.4 Assembler syntax

SWI{cond} <expression>

{cond} - two character condition mnemonic, see *Figure 8: Condition Codes*

<expression> is evaluated and placed in the comment field (which is ignored by ARM7).

4.10.5 Examples

```
SWI     ReadC                   ; get next character from read stream
SWI     WriteI+"k"              ; output a "k" to the write stream
SWINE   0                       ; conditionally call supervisor
                                ; with 0 in comment field
```

The above examples assume that suitable supervisor code exists, for instance:

```
   0x08 B Supervisor            ; SWI entry point

   EntryTable                   ; addresses of supervisor routines
            DCD ZeroRtn
            DCD ReadCRtn
            DCD WriteIRtn
             . . .

   Zero     EQU  0
   ReadC    EQU  256
   WriteI   EQU  512

Supervisor

; SWI has routine required in bits 8-23 and data (if any) in bits 0-7.
; Assumes R13_svc points to a suitable stack

   STMFD    R13,{R0-R2,R14}     ; save work registers and return address
   LDR      R0,[R14,#-4]        ; get SWI instruction
   BIC      R0,R0,#0xFF000000   ; clear top 8 bits
   MOV      R1,R0,LSR#8         ; get routine offset
   ADR      R2,EntryTable       ; get start address of entry table
   LDR      R15,[R2,R1,LSL#2]   ; branch to appropriate routine

   WriteIRtn                    ; enter with character in R0 bits 0-7
       .  .  .  .  .  .
   LDMFD    R13,{R0-R2,R15}^    ; restore workspace and return
                                ; restoring processor mode and flags
```

4.11 Coprocessor data operations (CDP)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 28: Coprocessor Data Operation Instruction.*
This class of instruction is used to tell a coprocessor to perform some internal operation. No result is communicated back to ARM7, and it will not wait for the operation to complete. The coprocessor could contain a queue of such instructions awaiting execution, and their execution can overlap other ARM72S + 1N incremental cycles, where S and N are as defined in section 5.1 Cycle types on page 65. activity allowing the coprocessor and ARM7 to perform independent tasks in parallel.

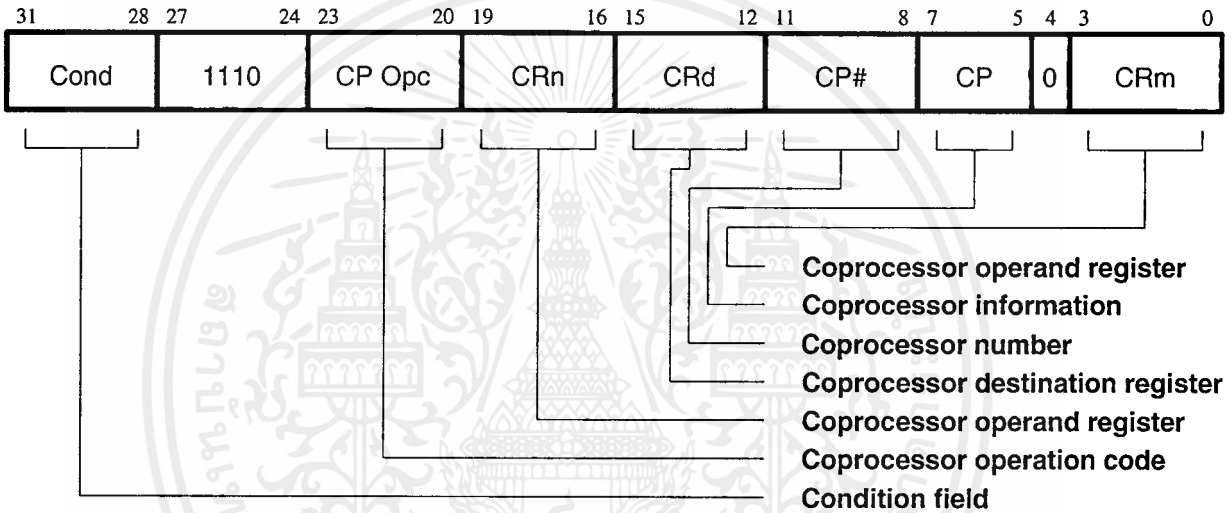


Figure 28: Coprocessor Data Operation Instruction

4.11.1 The Coprocessor fields

Only bit 4 and bits 24 to 31 are significant to ARM7; the remaining bits are used by coprocessors. The above field names are used by convention, and particular coprocessors may redefine the use of all fields except CP# as appropriate. The CP# field is used to contain an identifying number (in the range 0 to 15) for each coprocessor, and a coprocessor will ignore any instruction which does not contain its number in the CP# field.

The conventional interpretation of the instruction is that the coprocessor should perform an operation specified in the CP Opc field (and possibly in the CP field) on the contents of CRn and CRm, and place the result in CRd.

4.11.2 Instruction Cycle Times

Coprocessor data operations take 1S + bI incremental cycles to execute, where S and I are as defined in section 5.1 Cycle types on page 65.

b is the number of cycles spent in the coprocessor busy-wait loop.

4.11.3 Assembler syntax

CDP{cond} p#,<expression1>,cd,cn,cm{,<expression2>}

{cond} - two character condition mnemonic, see *Figure 8: Condition Codes*

p# - the unique number of the required coprocessor

<expression1> - evaluated to a constant and placed in the CP Opc field

cd, cn and cm evaluate to the valid coprocessor register numbers CRd, CRn and CRm respectively

<expression2> - where present is evaluated to a constant and placed in the CP field

4.11.4 Examples

```
CDP     p1,10,c1,c2,c3      ; request coproc 1 to do operation 10
                            ; on CR2 and CR3, and put the result in CR1

CDPEQ   p2,5,c1,c2,c3,2     ; if Z flag is set request coproc 2 to do
                            ; operation 5 (type 2) on CR2 and CR3,
                            ; and put the result in CR1
```

4.12 Coprocessor data transfers (LDC, STC)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 29: Coprocessor Data Transfer Instructions.*
This class of instruction is used to load (LDC) or store (STC) a subset of a coprocessors's registers directly to memory. ARM7 is responsible for supplying the memory address, and the coprocessor supplies or accepts the data and controls the number of words transferred.

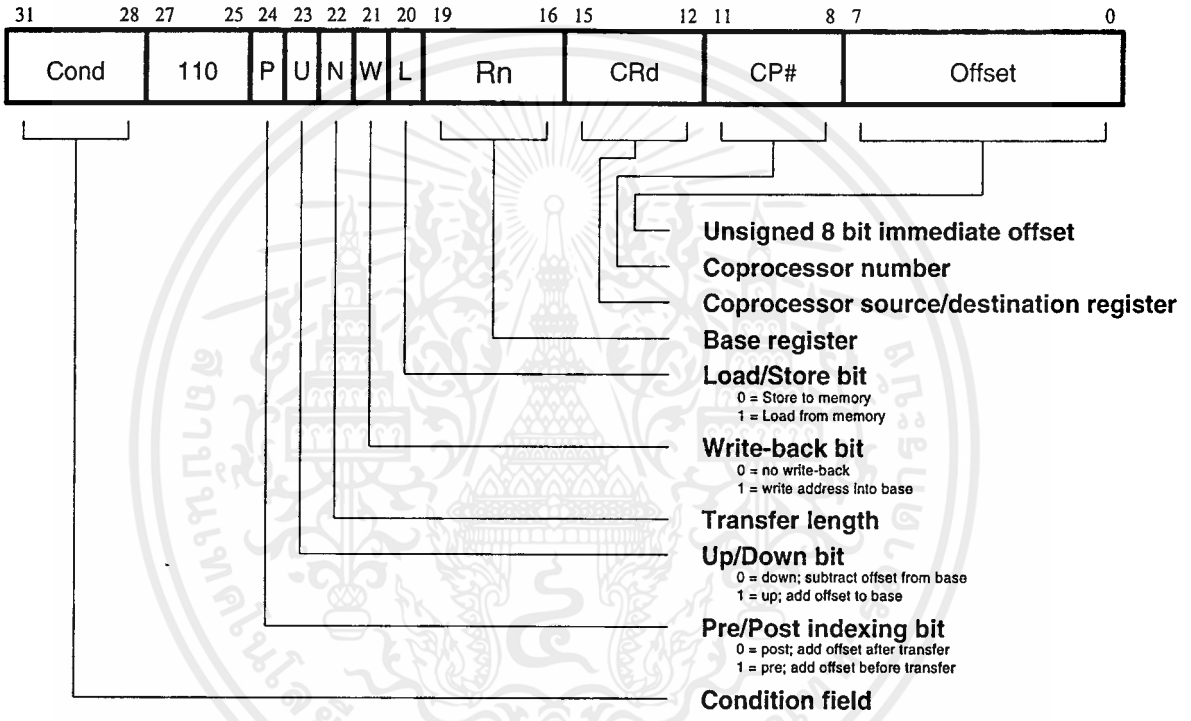


Figure 29: Coprocessor Data Transfer Instructions

4.12.1 The Coprocessor fields

The CP# field is used to identify the coprocessor which is required to supply or accept the data, and a coprocessor will only respond if its number matches the contents of this field.

The CRd field and the N bit contain information for the coprocessor which may be interpreted in different ways by different coprocessors, but by convention CRd is the register to be transferred (or the first register where more than one is to be transferred), and the N bit is used to choose one of two transfer length options. For instance N=0 could select the transfer of a single register, and N=1 could select the transfer of all the registers for context switching.

4.12.2 Addressing modes

ARM7 is responsible for providing the address used by the memory system for the transfer, and the addressing modes available are a subset of those used in single data transfer instructions. Note, however, that the immediate offsets are 8 bits wide and specify word offsets for coprocessor data transfers, whereas they are 12 bits wide and specify byte offsets for single data transfers.

The 8 bit unsigned immediate offset is shifted left 2 bits and either added to (U=1) or subtracted from (U=0) the base register (Rn); this calculation may be performed either before (P=1) or after (P=0) the base is used as the transfer address. The modified base value may be overwritten back into the base register (if W=1), or the old value of the base may be preserved (W=0). Note that post-indexed addressing modes require explicit setting of the W bit, unlike LDR and STR which always write-back when post-indexed.

The value of the base register, modified by the offset in a pre-indexed instruction, is used as the address for the transfer of the first word. The second word (if more than one is transferred) will go to or come from an address one word (4 bytes) higher than the first transfer, and the address will be incremented by one word for each subsequent transfer.

4.12.3 Address Alignment

The base address should normally be a word aligned quantity. The bottom 2 bits of the address will appear on A[1:0] and might be interpreted by the memory system.

4.12.4 Use of R15

If Rn is R15, the value used will be the address of the instruction plus 8 bytes. Base write-back to R15 shall not be specified.

4.12.5 Data aborts

If the address is legal but the memory manager generates an abort, the data trap will be taken. The write-back of the modified base will take place, but all other processor state will be preserved. The coprocessor is partly responsible for ensuring that the data transfer can be restarted after the cause of the abort has been resolved, and must ensure that any subsequent actions it undertakes can be repeated when the instruction is retried.

4.12.6 Instruction Cycle Times

All LDC instructions are emulated in software: the number of cycles taken will depend on the coprocessor support software.
Coprocessor data transfer instructions take (n-1)S + 2N + bI incremental cycles to execute, where S, N and I are as defined in section 5.1 Cycle types on page 65.

n is the number of words transferred.

b is the number of cycles spent in the coprocessor busy-wait loop.

4.12.7 Assembler syntax

<LDC|STC>{cond}{L} p#,cd,<Address>

LDC - load from memory to coprocessor

STC - store from coprocessor to memory

{L} - when present perform long transfer (N=1), otherwise perform short transfer (N=0)

{cond} - two character condition mnemonic, see *Figure 8: Condition Codes*

p# - the unique number of the required coprocessor

cd is an expression evaluating to a valid coprocessor register number that is placed in the CRd field

<Address> can be:

(i) An expression which generates an address:

<expression>

The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.

(ii) A pre-indexed addressing specification:

[Rn] offset of zero

[Rn,<#expression>]{!} offset of <expression> bytes

(iii) A post-indexed addressing specification:

[Rn],<#expression> offset of <expression> bytes

Rn is an expression evaluating to a valid ARM7 register number. Note, if Rn is R15 then the assembler will subtract 8 from the offset value to allow for ARM7 pipelining.

{!} write back the base register (set the W bit) if ! is present

4.12.8 Examples

```
LDC     p1,c2,table          ; load c2 of coproc 1 from address table,
                             ; using a PC relative address.
STCEQL  p2,c3,[R5,#24]!      ; conditionally store c3 of coproc 2 into
                             ; an address 24 bytes up from R5, write this
                             ; address back to R5, and use long transfer
                             ; option (probably to store multiple words)
```

Note that though the address offset is expressed in bytes, the instruction offset field is in words. The assembler will adjust the offset appropriately.

4.13 Coprocessor register transfers (MRC, MCR)

The is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 30: Coprocessor Register Transfer Instructions.*

This class of instruction is used to communicate information directly between ARM7 and a coprocessor. An example of a coprocessor to ARM7 register transfer (MRC) instruction would be a FIX of a floating point value held in a coprocessor, where the floating point number is converted into a 32 bit integer within the coprocessor, and the result is then transferred to ARM7 register. A FLOAT of a 32 bit value in ARM7 register into a floating point value within the coprocessor illustrates the use of ARM7 register to coprocessor transfer (MCR).

An important use of this instruction is to communicate control information directly from the coprocessor into the ARM7 CPSR flags. As an example, the result of a comparison of two floating point values within a coprocessor can be moved to the CPSR to control the subsequent flow of execution.

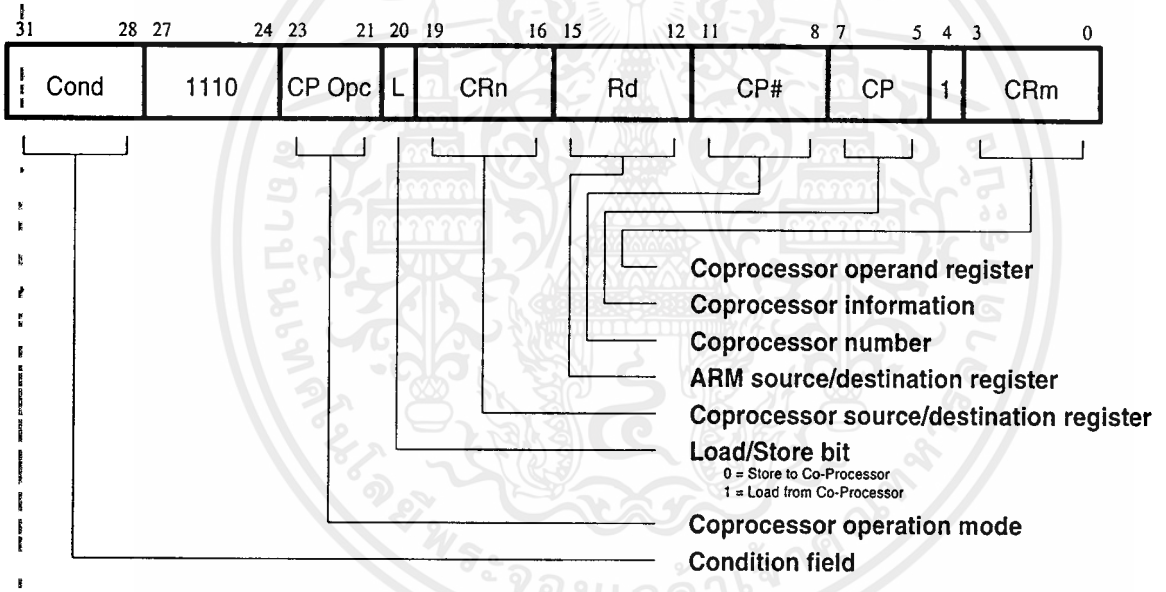


Figure 30: Coprocessor Register Transfer Instructions

4.13.1 The Coprocessor fields

The CP# field is used, as for all coprocessor instructions, to specify which coprocessor is being called upon.

The CP Opc, CRn, CP and CRm fields are used only by the coprocessor, and the interpretation presented here is derived from convention only. Other interpretations are allowed where the coprocessor functionality is incompatible with this one. The conventional interpretation is that the CP Opc and CP fields specify the operation the coprocessor is required to perform, CRn is the coprocessor register which is the source or destination of the transferred information, and CRm is a second coprocessor register which may be involved in some way which depends on the particular operation specified.

4.13.2 Transfers to R15

When a coprocessor register transfer to ARM7 has R15 as the destination, bits 31, 30, 29 and 28 of the transferred word are copied into the N, Z, C and V flags respectively. The other bits of the transferred word are ignored, and the PC and other CPSR bits are unaffected by the transfer.

4.13.3 Transfers from R15

A coprocessor register transfer from ARM7 with R15 as the source register will store the PC+12.

4.13.4 Instruction Cycle Times

MRC instructions take 1S + (b+1)I +1C incremental cycles to execute, where S, I and C are as defined in section 5.1 Cycle types on page 65.

MCR instructions take 1S + bI +1C incremental cycles to execute.

b is the number of cycles spent in the coprocessor busy-wait loop.

4.13.5 Assembler syntax

<MCR | MRC>{cond} p#,<expression1>,Rd,cn,cm{,<expression2>}

MRC - move from coprocessor to ARM7 register (L=1)

MCR - move from ARM7 register to coprocessor (L=0)

{cond} - two character condition mnemonic, see *Figure 8: Condition Codes*

p# - the unique number of the required coprocessor

<expression1> - evaluated to a constant and placed in the CP Opc field

Rd is an expression evaluating to a valid ARM7 register number

cn and cm are expressions evaluating to the valid coprocessor register numbers CRn and CRm respectively

<expression2> - where present is evaluated to a constant and placed in the CP field

4.13.6 Examples

```
MRC     2,5,R3,c5,c6        ; request coproc 2 to perform operation 5
                            ;  on c5 and c6, and transfer the (single
                            ;  32 bit word) result back to R3

MCR     6,0,R4,c6           ; request coproc 6 to perform operation 0
                            ;  on R4 and place the result in c6

MRCEQ   3,9,R3,c5,c6,2      ; conditionally request coproc 2 to perform
                            ;  operation 9 (type 2) on c5 and c6, and
                            ;  transfer the result back to R3
```

4.14 Undefined instruction

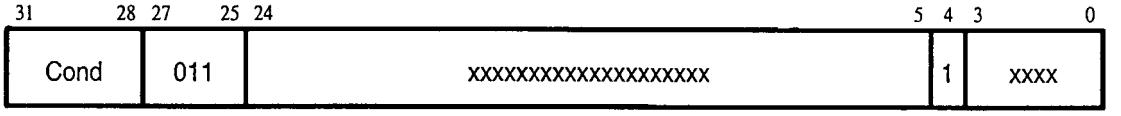


Figure 31: Undefined Instruction

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction format is shown in *Figure 31: Undefined Instruction.*

If the condition is true, the undefined instruction trap will be taken.

Note that the undefined instruction mechanism involves offering this instruction to any coprocessors which may be present, and all coprocessors must refuse to accept it by driving CPA and CPB HIGH.

4.14.1 Assembler syntax

At present the assembler has no mnemonics for generating this instruction. If it is adopted in the future for some specified use, suitable mnemonics will be added to the assembler. Until such time, this instruction shall not be used.

4.15 Instruction Set Examples

The following examples show ways in which the basic ARM7 instructions can combine to give efficient code. None of these methods saves a great deal of execution time (although they may save some), mostly they just save code.

4.15.1 Using the conditional instructions

(1) using conditionals for logical OR

```
CMP     Rn,#p                   ; if Rn=p OR Rm=q THEN GOTO Label
BEQ     Label
CMP     Rm,#q
BEQ     Label
```

can be replaced by

```
CMP     Rn,#p
CMPNE   Rm,#q                   ; if condition not satisfied try other test
BEQ     Label
```

(2) absolute value

```
TEQ     Rn,#0                   ; test sign
RSBMI   Rn,Rn,#0                ; and 2's complement if necessary
```

(3) multiplication by 4, 5 or 6 (run time)

```
MOV     Rc,Ra,LSL#2             ; multiply by 4
CMP     Rb,#5                   ; test value
ADDCS   Rc,Rc,Ra                ; complete multiply by 5
ADDHI   Rc,Rc,Ra                ; complete multiply by 6
```

(4) combining discrete and range tests

```
TEQ     Rc,#127                 ; discrete test
CMPNE   Rc,#" "-1               ; range test
MOVLS   Rc,#"."                 ; IF   Rc<=" " OR Rc=ASCII(127)
                                ; THEN Rc:="."
```

(5) division and remainder

A number of divide routines for specific applications are provided in source form as part of the ANSI C library provided with the ARM Cross Development Toolkit, available from your supplier. A short general pupose divide routine follows.

```
                                    ; enter with numbers in Ra and Rb
                                    ;
      MOV       Rcnt,#1             ; bit to control the division
Div1  CMP       Rb,#0x80000000      ; move Rb until greater than Ra
      CMPCC     Rb,Ra
      MOVCC     Rb,Rb,ASL#1
      MOVCC     Rcnt,Rcnt,ASL#1
      BCC       Div1
      MOV       Rc,#0
Div2  CMP       Ra,Rb               ; test for possible subtraction
      SUBCS     Ra,Ra,Rb            ; subtract if ok
      ADDCS     Rc,Rc,Rcnt          ; put relevant bit into result
      MOVS      Rcnt,Rcnt,LSR#1     ; shift control bit
      MOVNE     Rb,Rb,LSR#1         ; halve unless finished
      BNE       Div2
                                    ;
                                    ; divide result in Rc
                                    ; remainder in Ra
```

4.15.2 Pseudo random binary sequence generator

It is often necessary to generate (pseudo-) random numbers and the most efficient algorithms are based on shift generators with exclusive-OR feedback rather like a cyclic redundancy check generator. Unfortunately the sequence of a 32 bit generator needs more than one feedback tap to be maximal length (i.e. 2^32-1 cycles before repetition), so this example uses a 33 bit register with taps at bits 33 and 20. The basic algorithm is newbit:=bit 33 eor bit 20, shift left the 33 bit number and put in newbit at the bottom; this operation is performed for all the newbits needed (i.e. 32 bits). The entire operation can be done in 5 S cycles:

```
                                    ; enter with seed in Ra (32 bits),
                                      Rb (1 bit in Rb lsb), uses Rc
                                    ;
      TST       Rb,Rb,LSR#1         ; top bit into carry
      MOVS      Rc,Ra,RRX           ; 33 bit rotate right
      ADC       Rb,Rb,Rb            ; carry into lsb of Rb
      EOR       Rc,Rc,Ra,LSL#12     ; (involved!)
      EOR       Ra,Rc,Rc,LSR#20     ; (similarly involved!)
                                    ;
                                    ; new seed in Ra, Rb as before
```

4.15.3 Multiplication by constant using the barrel shifter

(1) Multiplication by 2^n (1,2,4,8,16,32..)

```
      MOV       Ra, Rb, LSL #n
```

(2) Multiplication by 2^n+1 (3,5,9,17..)

```
      ADD       Ra,Ra,Ra,LSL #n
```

(3) Multiplication by 2^n-1 (3,7,15..)

```
RSB     Ra,Ra,Ra,LSL #n
```

(4) Multiplication by 6

```
ADD     Ra,Ra,Ra,LSL #1       ; multiply by 3
MOV     Ra,Ra,LSL#1           ; and then by 2
```

(5) Multiply by 10 and add in extra number

```
ADD     Ra,Ra,Ra,LSL#2        ; multiply by 5
ADD     Ra,Rc,Ra,LSL#1        ; multiply by 2 and add in next digit
```

(6) General recursive method for Rb := Ra*C, C a constant:

(a) If C even, say C = 2^n*D, D odd:

```
D=1:    MOV   Rb,Ra,LSL #n
D<>1:   {Rb := Ra*D}
        MOV     Rb,Rb,LSL #n
```

(b) If C MOD 4 = 1, say C = 2^n*D+1, D odd, n>1:

```
D=1:    ADD   Rb,Ra,Ra,LSL #n
D<>1:   {Rb := Ra*D}
        ADD     Rb,Ra,Rb,LSL #n
```

(c) If C MOD 4 = 3, say C = 2^n*D-1, D odd, n>1:

```
D=1:    RSB   Rb,Ra,Ra,LSL #n
D<>1:   {Rb := Ra*D}
        RSB     Rb,Ra,Rb,LSL #n
```

This is not quite optimal, but close. An example of its non-optimality is multiply by 45 which is done by:

```
RSB     Rb,Ra,Ra,LSL#2        ; multiply by 3
RSB     Rb,Ra,Rb,LSL#2        ; multiply by 4*3-1 = 11
ADD     Rb,Ra,Rb,LSL# 2       ; multiply by 4*11+1 = 45
```

rather than by:

```
ADD     Rb,Ra,Ra,LSL#3        ; multiply by 9
ADD     Rb,Rb,Rb,LSL#2        ; multiply by 5*9 = 45
```

4.15.4 Loading a word from an unknown alignment

```
                                ; enter with address in Ra (32 bits)
                                ; uses Rb, Rc; result in Rd.
                                ; Note d must be less than c e.g. 0,1
                                ;
BIC       Rb,Ra,#3              ; get word aligned address
LDMIA     Rb,{Rd,Rc}            ; get 64 bits containing answer
AND       Rb,Ra,#3              ; correction factor in bytes
MOVS      Rb,Rb,LSL#3           ; ...now in bits and test if aligned
MOVNE     Rd,Rd,LSR Rb          ; produce bottom of result word
                                ; (if not aligned)
RSBNE     Rb,Rb,#32             ; get other shift amount
ORRNE     Rd,Rd,Rc,LSL Rb       ; combine two halves to get result
```

4.15.5 Loading a halfword (Little Endian)

```
LDR       Ra, [Rb,#2]           ; Get halfword to bits 15:0
MOV       Ra,Ra,LSL #16         ; move to top
MOV       Ra,Ra,LSR #16         ; and back to bottom
                                ; use ASR to get sign extended version
```

4.15.6 Loading a halfword (Big Endian)

```
LDR       Ra, [Rb,#2]           ; Get halfword to bits 31:16
MOV       Ra,Ra,LSR #16         ;  and back to bottom
                                ;  use ASR to get sign extended version
```

กิตติกรรมประกาศ

ขอกราบขอบพระคุณบุคคลที่มีรายนามดังต่อไปนี้ที่ให้ความช่วยเหลือในการทำโครงงานนี้ให้สำเร็จลุล่วง

1. อาจารย์อภิเนตร อูนากุล

   สอน Interface , Computer Sciences เป็นต้น

   ให้คำปรึกษาและแนวทางในการทำโครงงานนี้มาโดยตลอด

2. อาจารย์นวพร วรรณวิมลศรี

   สอน Switching เป็นต้น

   ให้คำปรึกษา และตรวจงานทุกสัปดาห์

3. ห้องปฏิบัติการ ESL

   เอื้อเฟื้อสถานที่ทำงาน ที่นอน ที่รับประทาน และเครื่องคอมพิวเตอร์

4. นายพหล ศิริเหลืองทอง

   วิศวกรรมศาสตร์ไฟฟ้า จุฬาลงกรณ์มหาวิทยาลัย

   ให้ความช่วยเหลือในการทดสอบการทำงาน และเป็นกำลังใจให้เสมอมา

5. บุคคลทั่วไป

   ผู้ที่ให้กำลังใจทั้งที่ตั้งใจและไม่ตั้งใจทุกท่าน

6. ภาควิชาวิศวกรรมคอมพิวเตอร์

เอกสารอ้างอิง

1. Steve Carlson , "Introduction To HDL-BASED Design Using VHDL" , Synopsys Inc.,1991
2. www.arm.com , "Data Sheet ARM"
3. Douglas L.Perry, "VHDL",McGRAW-HILL,second edition
4. Roger lipsett ,Carl F. schaefer,cary ussery , "VHDL:Hardware Description and Design" , Kluwer Academic Publishers.