Chapter 1: Instructions

1.1 Instruction Word Format

All instructions for BARNDLES are assembled into fixed-length 24-bit instructions. The most significant byte of each instruction is composed of an op code, an immediate flag `I`, a CPSR update flag `S`, and an invert flag `N`. The following bytes vary depending on the type of the instruction. Non-immediate instructions are composed of 4-bits of reserved space, a first operand register `Rm`, a second operand register `Rn`, and a destination register `Rd`. Immediate instructions are composed of a 12-bit immediate value `Imm12` and a destination register `Rd`.

23 – 19	18	17	16	15 – 12	11 – 8	7 – 4	3 – 0
Op Code	I	S	N	Res.	Rm	Rn	Rd
5	1	1	1	4	4	4	4

Figure 1.1: The BARNDLES non-immediate instruction format

23 – 19	18	17	16	15 – 4	3 – 0
Op Code	I	S	N	Imm12	Rd
5	1	1	1	12	4

Figure 1.2: The BARNDLES immediate instruction format

Op Code

The op code specifies which operation the microprocessor should execute on its operands.

Immediate Flag (`I`)

The immediate flag specifies whether the instruction should be decoded as an immediate instruction or as a non-immediate instruction. If the immediate flag is set, the instruction will be decoded as an immediate instruction. Otherwise, it will be decoded as a non-immediate instruction.

CPSR Update Flag (`S`)

The CPSR update flag `S` specifies whether or not the current program status register `CPSR` should be updated to store the ALU status flags set by the execution of the instruction. If the CPSR update flag is set, the `CPSR` will be updated on writeback. Otherwise, the `CPSR` will not be updated.

Invert Flag (`N`)

The invert flag `N` specifies whether or not the value output by the ALU after the execution of the instruction should be inverted before writeback. If the invert flag is set, the ALU output will be inverted. Otherwise, it will not be affected.

First Operand Register (`Rm`)

The first operand register is the register that contains the first operand for the operation. This field is ignored if the instruction is in the immediate format (the immediate flag `I` is set) or if the instruction does not require a first operand register to be specified.

Second Operand Register (`Rn`)

The second operand register is the register that contains the second operand for the operation. This field is ignored if the instruction is in the immediate format (the immediate flag `I` is set) or if the instruction does not require a second operand register to be specified.

Destination Register (`Rd`)

The destination register is the register that the output from the operation will be stored in. This field is ignored if the instruction does not require a destination register to be specified.

12-bit Immediate (`Imm12`)

The 12-bit immediate field holds a 12-bit immediate value that can be used as an operand in immediate instructions. This field is ignored if the instruction is in the non-immediate format (the immediate flag `I` is cleared) or if the instruction does not require a 12-bit immediate to be specified.

1.2 Assembly Instruction Format

1.2.1 Optional Suffixes

Optional suffixes can be optionally appended to the end of an instruction to add additional functionality to it. Any number of optional suffixes can be appended to a single instruction. The order that the optional suffixes are appended in does not matter to the assembler, but convention suggests that suffixes be appended in alphabetical order. The optional suffixes provided by BARNDLES ASM are as follows.

Invert Flag (`N`)
The invert flag will cause the output of the ALU to be inverted before writeback. This is useful for creating additional operations from existing functions. For example, you can convert the `MOV` instruction to a move not instruction `MOVN` or the `AND` instruction to a NAND instruction `ANDN`.

CPSR Update Flag (`S`)
The CPSR update flag will update the current program status register `CPSR` to reflect the ALU status flags produced by the instruction that the update flag is appended to. This allows the programmer to optimize their code by partially eliminating the need for compare instructions `CPS` and `CPA`.

1.2.2 Flexible Address Operand

Flexible address operands allow the programmer to use one of multiple formats for specifying a memory address. Upon assembly, the flexible operand is replaced with a 12-bit immediate containing the memory address that the programmer's flexible operand references. Given they are converted to immediates, flexible address operands should only be used in immediate instructions. The flexible address operands provided by BARNDLES ASM are as follows.

Label (`label`)
The label operand allows you to reference the memory location of a label in the assembly program.

12-bit Immediate (`#Imm12`)
The immediate operand allows you to directly specify a memory location by its 12-bit address.

1.3 Instruction Set

1.3.1 Do Nothing (DON)

The `DON` instruction does nothing.

Non-Immediate Syntax
`DON`

Immediate Syntax
`DON`

Non-Immediate Operation
`None`

Immediate Operation
`None`

1.3.2 Move (MOV)

The `MOV` instruction moves an integer into a register or between registers.

Non-Immediate Syntax
`MOV{N,S} Rd, Rm`

Immediate Syntax
`MOV{N,S} Rd, #Imm12`

Non-Immediate Operation
`Rd <= Rm`

Immediate Operation
`Rd <= Imm12`

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.3 Add (ADD)

The `ADD` instruction adds one integer to another integer and stores the sum in a register.

Non-Immediate Syntax
```
ADD{N,S} Rd, Rm, Rn
```

Immediate Syntax
```
ADD{N,S} Rd, #Imm12
```

Non-Immediate Operation
```
Rd <= Rm + Rn
```

Immediate Operation
```
Rd <= Rd + Imm12
```

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.4 Subtract (SUB)

The `SUB` instruction subtracts one integer from another integer and stores the difference in a register.

Non-Immediate Syntax
```
SUB{N,S} Rd, Rm, Rn
```

Immediate Syntax
```
SUB{N,S} Rd, #Imm12
```

Non-Immediate Operation
```
Rd <= Rm - Rn
```

Immediate Operation
```
Rd <= Rd - Imm12
```

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.5 Multiply (MUL)

The `MUL` instruction multiplies one integer by another integer and stores the product in a register.

Non-Immediate Syntax
```
MUL{N,S} Rd, Rm, Rn
```

Immediate Syntax
```
MUL{N,S} Rd, #Imm12
```

Non-Immediate Operation
```
Rd <= Rm × Rn
```

Immediate Operation
```
Rd <= Rd × Imm12
```

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.6 Divide (DIV)

The `DIV` instruction divides one integer by another integer and stores the product in a register.

Non-Immediate Syntax
```
DIV{N,S} Rd, Rm, Rn
```

Immediate Syntax
```
DIV{N,S} Rd, #Imm12
```

Non-Immediate Operation
```
Rd <= Rm ÷ Rn
```

Immediate Operation
```
Rd <= Rd ÷ Imm12
```

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.7 Bitwise AND (AND)

The `AND` instruction bitwise ANDs one integer by another integer and stores the result in a register.

Non-Immediate Syntax
```
AND{N,S} Rd, Rm, Rn
```

Immediate Syntax
```
AND{N,S} Rd, #Imm12
```

Non-Immediate Operation
```
Rd <= Rm & Rn
```

Immediate Operation
```
Rd <= Rd & Imm12
```

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.8 Bitwise OR (ORR)

The `ORR` instruction bitwise ORs one integer by another integer and stores the result in a register.

Non-Immediate Syntax
```
ORR{N,S} Rd, Rm, Rn
```

Immediate Syntax
```
ORR{N,S} Rd, #Imm12
```

Non-Immediate Operation
```
Rd <= Rm | Rn
```

Immediate Operation
```
Rd <= Rd | Imm12
```

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.9 Bitwise XOR (XOR)

The `ORR` instruction bitwise XORs one integer by another integer and stores the result in a register.

Non-Immediate Syntax
```
XOR{N,S} Rd, Rm, Rn
```

Immediate Syntax
```
XOR{N,S} Rd, #Imm12
```

Non-Immediate Operation
```
Rd <= Rm ^ Rn
```

Immediate Operation
```
Rd <= Rd ^ Imm12
```

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.10 Logical AND (LGA)

The `LGA` instruction logically ANDs one integer by another integer and stores the result in a register.

Non-Immediate Syntax
```
LGA{N,S} Rd, Rm, Rn
```

Immediate Syntax
```
LGA{N,S} Rd, #Imm12
```

Non-Immediate Operation
```
Rd <= Rm && Rn
```

Immediate Operation
```
Rd <= Rd && Imm12
```

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.11 Logical OR (LGO)

The `LGO` instruction logically ORs one integer by another integer and stores the result in a register.

Non-Immediate Syntax
```
LGO{N,S} Rd, Rm, Rn
```

Immediate Syntax
```
LGO{N,S} Rd, #Imm12
```

Non-Immediate Operation
```
Rd <= Rm || Rn
```

Immediate Operation
```
Rd <= Rd || Imm12
```

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.12 Logical XOR (LGX)

The `LGX` instruction logically XORs one integer by another integer and stores the result in a register.

Non-Immediate Syntax
```
LGX{N,S} Rd, Rm, Rn
```

Immediate Syntax
```
LGX{N,S} Rd, #Imm12
```

Non-Immediate Operation
```
Rd <= Rm ^^ Rn
```

Immediate Operation
```
Rd <= Rd ^^ Imm12
```

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.13 Logical Shift Left (LSL)

The `LSL` instruction logically shifts an integer left by another integer and stores the result in a register.

Non-Immediate Syntax
```
LSL{N,S} Rd, Rm, Rn
```

Immediate Syntax
```
LSL{N,S} Rd, #Imm12
```

Non-Immediate Operation
```
Rd <= Rm LSL Rn
```

Immediate Operation
```
Rd <= Rd LSL Imm12
```

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.14 Logical Shift Right (LSR)

The `LSR` instruction logically shifts an integer right by another integer and stores the result in a register.

Non-Immediate Syntax
```
LSR{N,S} Rd, Rm, Rn
```

Immediate Syntax
```
LSR{N,S} Rd, #Imm12
```

Non-Immediate Operation
```
Rd <= Rm LSR Rn
```

Immediate Operation
```
Rd <= Rd LSR Imm12
```

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.15 Arithmetic Shift Right (ASR)

The `ASR` instruction arithmetically shifts an integer right by another integer and stores the result in a register.

Non-Immediate Syntax
```
ASR{N,S} Rd, Rm, Rn
```

Immediate Syntax
```
ASR{N,S} Rd, #Imm12
```

Non-Immediate Operation
```
Rd <= Rm ASR Rn
```

Immediate Operation
```
Rd <= Rd ASR Imm12
```

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.16 Branch (BCH)

The `BCH` instruction assigns a new value to the program counter register `PC`.

Non-Immediate Syntax
```
BCH Rd
```

Immediate Syntax
```
BCH <AddrOp>
```

Non-Immediate Operation
```
PC <= Rd
```

Immediate Operation
```
PC <= <AddrOp>
```

Where:

- `Rd` is the destination register.
- `<AddrOp>` is the flexible address operand. See Flexible Address Operand for more information.

1.3.17 Branch and Link (BAL)

The `BAL` instruction stores the current value of the program counter register `PC` in the link register `LR` and assigns a new value to the program counter register.

Non-Immediate Syntax

```
BAL Rd
```

Immediate Syntax

```
BAL <AddrOp>
```

Non-Immediate Operation

```
LR <= PC
PC <= Rd
```

Immediate Operation

```
LR <= PC
PC <= <AddrOp>
```

Where:

- `Rd` is the destination register.
- `<AddrOp>` is the flexible address operand. See Flexible Address Operand for more information.

1.3.18 Branch If Equal (BEQ)

The `BEQ` instruction assigns a new value to the program counter register `PC` if the last instruction to update the current program status register `CPSR` produced a zero result.

Non-Immediate Syntax

```
BEQ Rd
```

Immediate Syntax

```
BEQ <AddrOp>
```

Non-Immediate Operation

```
if CPSR[Z]:
PC <= Rd
```

Immediate Operation

```
if CPSR[Z]:
PC <= <AddrOp>
```

Where:

- `Rd` is the destination register.
- `<AddrOp>` is the flexible address operand. See Flexible Address Operand for more information.
- `Z` is the ALU's zero status flag as read from the current program status register `CPSR`.

1.3.19 Branch If Not Equal (BNE)

The `BNE` instruction assigns a new value to the program counter register `PC` if the last instruction to update the current program status register `CPSR` produced a non-zero result.

Non-Immediate Syntax

```
BNE Rd
```

Immediate Syntax

```
BNE <AddrOp>
```

Non-Immediate Operation

```
if not CPSR[Z]:
PC <= Rd
```

Immediate Operation

```
if not CPSR[Z]:
PC <= <AddrOp>
```

Where:

- `Rd` is the destination register.
- `<AddrOp>` is the flexible address operand. See Flexible Address Operand for more information.
- `Z` is the ALU's zero status flag as read from the current program status register `CPSR`.

1.3.20 Branch If Greater Than (BGT)

The `BGT` instruction assigns a new value to the program counter register `PC` if the last instruction to update the current program status register `CPSR` produced a non-negative and non-zero result.

Non-Immediate Syntax

```
BGT Rd
```

Immediate Syntax

```
BGT <AddrOp>
```

Non-Immediate Operation

```
if not CPSR[N] and not CPSR[Z]:
PC <= Rd
```

Immediate Operation

```
if not CPSR[N] and not CSPR[Z]:
PC <= <AddrOp>
```

Where:

- `Rd` is the destination register.
- `<AddrOp>` is the flexible address operand. See Flexible Address Operand for more information.
- `Z` is the ALU's zero status flag as read from the current program status register `CPSR`.
- `N` is the ALU's negative status flag as read from the current program status register `CPSR`.

1.3.21 Branch If Less Than (BLT)

The `BLT` instruction assigns a new value to the program counter register `PC` if the last instruction to update the current program status register `CPSR` produced a negative result.

Non-Immediate Syntax

```
BLT Rd
```

Immediate Syntax

```
BLT <AddrOp>
```

Non-Immediate Operation

```
if CPSR[N]:
PC <= Rd
```

Immediate Operation

```
if CPSR[N]:
PC <= <AddrOp>
```

Where:

- `Rd` is the destination register.
- `<AddrOp>` is the flexible address operand. See Flexible Address Operand for more information.
- `N` is the ALU's negative status flag as read from the current program status register `CPSR`.

1.3.22 Branch If Greater Than or Equal To (BGE)

The `BGE` instruction assigns a new value to the program counter register `PC` if the last instruction to update the current program status register `CPSR` produced a non-negative result.

Non-Immediate Syntax

```
BGE Rd
```

Immediate Syntax

```
BGE <AddrOp>
```

Non-Immediate Operation

```
if not CPSR[N]:
PC <= Rd
```

Immediate Operation

```
if not CPSR[N]:
PC <= <AddrOp>
```

Where:

- `Rd` is the destination register.
- `<AddrOp>` is the flexible address operand. See Flexible Address Operand for more information.
- `N` is the ALU's negative status flag as read from the current program status register `CPSR`.

1.3.23 Branch If Less Than or Equal To (BLE)

The `BLE` instruction assigns a new value to the program counter register `PC` if the last instruction to update the current program status register `CPSR` produced a zero or negative result.

Non-Immediate Syntax

```
BLE Rd
```

Immediate Syntax

```
BLE <AddrOp>
```

Non-Immediate Operation

```
if CPSR[N] or CPSR[Z]:
PC <= Rd
```

Immediate Operation

```
if CPSR[N] or CPSR[Z]:
PC <= <AddrOp>
```

Where:

- `Rd` is the destination register.
- `<AddrOp>` is the flexible address operand. See Flexible Address Operand for more information.
- `CPSR[Z]` is the ALU's zero status flag as read from the current program status register `CPSR`.
- `CPSR[N]` is the ALU's negative status flag as read from the current program status register `CPSR`.

1.3.24 Compare Add (CPA)

The `CPA` instruction compares two integers by adding them then updating the `CPSR`

Non-Immediate Syntax

```
CPA Rd, Rm
```

Immediate Syntax

```
CPA Rd, #Imm12
```

Non-Immediate Operation

```
Rd + Rm, Update CPSR
```

Immediate Operation

```
Rd + Imm12, Update CPSR
```

Where:

- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.25 Compare Subtract (CPS)

The `CPS` instruction compares two integers by subtracting them then updating the `CPSR`.

Non-Immediate Syntax

```
CPS Rd, Rm
```

Immediate Syntax

```
CPS Rd, #Imm12
```

Non-Immediate Operation

```
Rd - Rm, Update CPSR
```

Immediate Operation

```
Rd - Imm12, Update CPSR
```

Where:

- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.

1.3.26 Load Register (LDR)

The `LDR` instruction loads a word from an offset in read-write data memory and stores it in a register.

Non-Immediate Syntax

```
LDR Rd, [Rm]
```

Immediate Syntax

```
LDR Rd, [<AddrOp>]
```

Non-Immediate Operation

```
Rd <= MEM[Rm]
```

Immediate Operation

```
Rd <= MEM[<AddrOp>]
```

Where:

- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.
- `MEM` is read-write data memory.

1.3.27 Store Register (STR)

The `STR` instruction takes a word stored in a register and stores it at an offset in read-write data memory.

Non-Immediate Syntax

```
STR{N,S} Rd, [Rm]
```

Immediate Syntax

```
STR{N,S} Rd, [<AddrOp>]
```

Non-Immediate Operation

```
MEM[Rm] <= Rd
```

Immediate Operation

```
MEM[<AddrOp>] <= Rd
```

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `Rm` is the first operand register.
- `Imm12` is a 12-bit immediate.
- `MEM` is read-write data memory.

1.3.28 Push (PSH)

The `PSH` instruction increments the offset in the stack pointer register `SP`. Then it takes a word stored in a register and stores it in stack memory at the new offset.

Non-Immediate Syntax

```
PSH{N,S} Rd
```

Immediate Syntax

```
PSH{N,S} #Imm12
```

Non-Immediate Operation

```
SP <= SP + 1
STACK[SP] <= Rd
```

Immediate Operation

```
SP <= SP + 1
STACK[SP] <= Imm12
```

Where:

- `N` is the invert flag. See Optional Suffixes for more information.
- `S` is the CPSR update flag. See Optional SuffixesChapter 1: for more information.
- `Rd` is the destination register.
- `SP` is the stack pointer register.
- `STACK` is stack memory.

1.3.29 Pop (POP)

The `POP` instruction takes the word stored in stack memory at the offset specified by the stack pointer register `SP` and stores it in a register. Then it decrements the offset in the stack pointer register.

Non-Immediate Syntax

```
POP Rd
```

Immediate Syntax

```
POP [<AddrOp>]
```

Non-Immediate Operation

```
Rd <= STACK[SP--]
```

Immediate Operation

```
DMEM[<AddrOp>] <= STACK
```

Where:

- `Rd` is the destination register.
- `SP` is the stack pointer register.
- `STACK` is stack memory.

1.4 Table of Instructions (Quick Reference)

Instruction Name	Non-Immediate Syntax	Non-Immediate Operation	Immediate Syntax	Immediate Operation
Do Nothing	DON	NONE	DON	NONE
Move	MOV{N,S} Rd, Rm	Rd <= Rm	MOV{N,S} Rd, #Imm12	Rd <= Imm12
Add	ADD{N,S} Rd, Rm, Rn	Rd <= Rm + Rn	ADD{N,S} Rd, #Imm12	Rd <= Rd + Imm12
Subtract	SUB{N,S} Rd, Rm, Rn	Rd <= Rm - Rn	SUB{N,S} Rd, #Imm12	Rd <= Rd - Imm12
Multiply	MUL{N,S} Rd, Rm, Rn	Rd <= Rm × Rn	MUL{N,S} Rd, #Imm12	Rd <= Rd × Imm12
Divide	DIV{N,S} Rd, Rm, Rn	Rd <= Rm ÷ Rn	DIV{N,S} Rd, #Imm12	Rd <= Rm ÷ Imm12
Bitwise AND	AND{N,S} Rd, Rm, Rn	Rd <= Rm & Rn	AND{N,S} Rd, #Imm12	Rd <= Rd & Imm12
Bitwise OR	ORR{N,S} Rd, Rm, Rn	Rd <= Rm | Rn	ORR{N,S} Rd, #Imm12	Rd <= Rd | Imm12
Bitwise XOR	XOR{N,S} Rd, Rm, Rn	Rd <= Rm ^ Rn	XOR{N,S} Rd, #Imm12	Rd <= Rd ^ Imm12
Logical AND	LGA{N,S} Rd, Rm, Rn	Rd <= Rm && Rn	LGA{N,S} Rd, #Imm12	Rd <= Rm && Imm12
Logical OR	LGO{N,S} Rd, Rm, Rn	Rd <= Rm || Rn	LGO{N,S} Rd, #Imm12	Rd <= Rm || Imm12
Logical XOR	LGX{N,S} Rd, Rm, Rn	Rd <= Rm ^^ Rn	LGX{N,S} Rd, #Imm13	Rd <= Rm ^^ Imm12
Logical Shift Left	LSL{N,S} Rd, Rm, Rn	Rd <= Rm LSL Rn	LSL{N,S} Rd, #Imm12	Rd <= Rd LSL Imm12
Logical Shift Right	LSR{N,S} Rd, Rm, Rn	Rd <= Rm LSR Rn	LSR{N,S} Rd, #Imm12	Rd <= Rd LSR Imm12
Arithmetic Shift Right	ASR{N,S} Rd, Rm, Rn	Rd <= Rm ASR Rn	ASR{N,S} Rd, #Imm12	Rd <= Rd ASR Imm12
Compare Add	CPA Rd, Rm	Rd - Rm, Update CPSR	CPA Rd, #Imm12	Rd - Imm12, Update CPSR
Compare Subtract	CPS Rd, Rm	Rd - Rm, Update CPSR	CPS Rd, #Imm12	Rd - Imm12, Update CPSR
Branch	BCH Rd	PC <= Rd	BCH <AddrOp>	PC <= <BrOp>
Branch and Link	BAL Rd	LR <= PC, PC <= Rd	BAL <AddrOp>	LR <= PC, PC <= <AddrOp>
Branch if Equal	BEQ Rd	PC <= Rd	BEQ <AddrOp>	PC <= <AddrOp>
Branch if Not Equal	BNE Rd	PC <= Rd	BNE <AddrOp>	PC <= <AddrOp>
Branch if Greater Than	BGT Rd	PC <= Rd	BGT <AddrOp>	PC <= <AddrOp>
Branch if Less Than	BLT Rd	PC <= Rd	BLT <AddrOp>	PC <= <AddrOp>
Branch if Greater Than or Equal To	BGE Rd	PC <= Rd	BGE <AddrOp>	PC <= <AddrOp>
Branch if Less Than or Equal To	BLE Rd	PC <= Rd	BLE <AddrOp>	PC <= <AddrOp>
Load Register	LDR Rd, Rm	Rd <= DMEM[Rm]	LDR Rd, <AddrOp>	Rd <= DMEM[<AddrOp>]
Store Register	STR Rd, Rm	DMEM[Rm] <= Rd	STR Rd, <AddrOp>	DMEM[<AddrOp>] <= Rd
Push	PSH Rd	STACK[++SP] <= Rd	PSH <AddrOp>	STACK[++SP] <= DMEM[<AddrOp>]
Pop	POP Rd	Rd <= STACK[SP--]	POP <AddrOp>	DMEM[<AddrOp>] <= STACK[SP--]

Flexible Address Operand <AddrOp>	
label	References the memory location of a label
#Imm12	References a memory location by its 12-bit address

Optional Suffixes	
N	Inverts the ALU output before writeback
S	Updates the CPSR after instruction execution

Figure 1.3: Complete table of BARNDLES ASM instructions and special terminology.

Chapter 2: Memory

2.1 Memory Encoding

2.2 Memory Image

Because Blender's shader editor was designed to create shaders and not microprocessors, it is exclusively composed of forward processing logic. In conventional use cases this approach is efficient and effective, but it poses an issue when you're trying to develop digital memory.

Digital memory requires a feedback loop, a circuitry device that takes signals from one point of a circuit and routes them back into the circuit at an earlier position. Unfortunately, the forward processing nature of Blender's shader editor makes the implementation of feedback loops impossible, so a work-around had to be developed. The solution is to create a feedback loop outside of the shader editor in an element called the memory image.

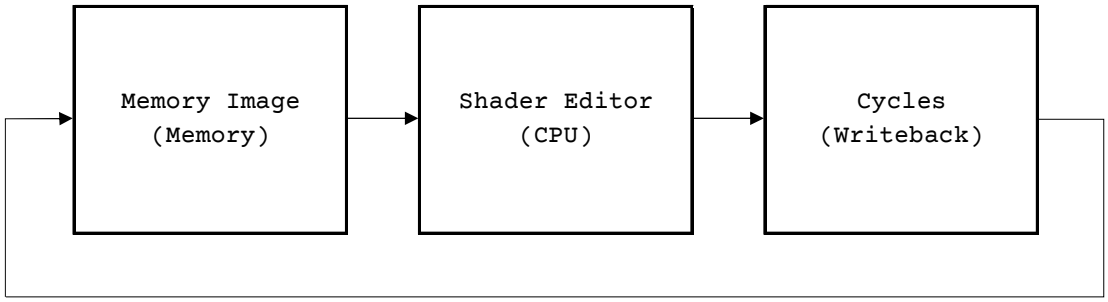

Figure 2.1: The feedback loop constructed for the BARNDLES memory system.

In short, the memory image is a 64px by 64px image rendered with Cycles after every clock cycle. Each pixel stores a 24-bit word in memory which means that there is a total of 4,096 words or 12,288 bytes of memory available to the system. Each of these words can be accessed via an address between `0x000` and `0xFFF`.

The memory image contains every piece and level of memory available to BARNDLES, so it is divided into sections that serve specific purposes. In its current configuration, the memory image allocates 2,560 words to text memory, 512 words to read-only data memory, 512 words to read/write data memory, 496 words to stack memory, and 16 words to the register file.

Memory Name	Size [words]	Size [bytes]	Address Range
Text Memory	2560	7680	0x600-0xFFF
Read-Only Data Memory	512	1536	0x400-0x5FF
Read/Write Data Memory	512	1536	0x200-0x3FF
Stack Memory	496	1488	0x010-0x1FF
Register File	16	48	0x000-0x00F

Figure 2.2: The sizes and address ranges of the memory image's sections.

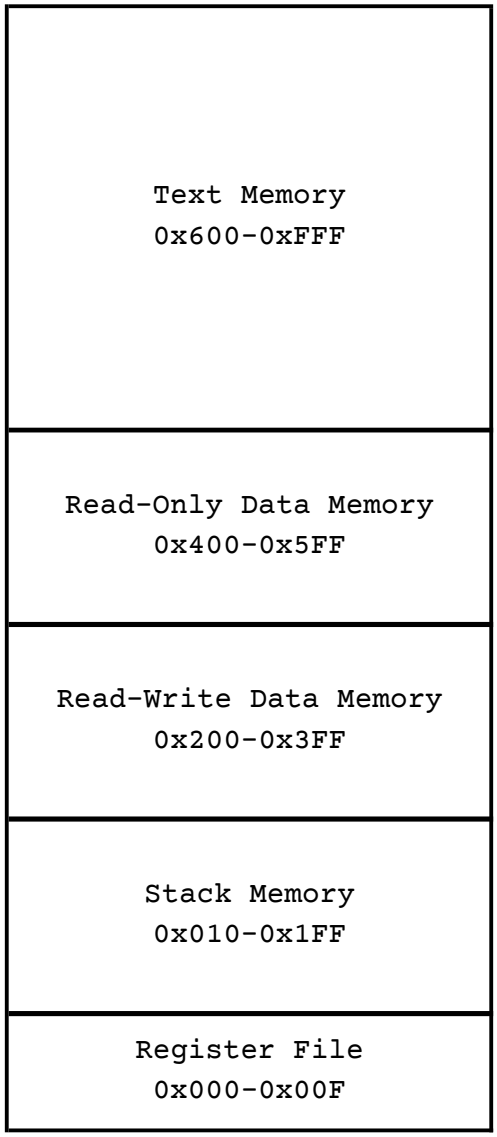

Figure 2.3: Visual display of the sections of the memory image.

In future versions of BARNDLES, it is possible to expand the size of the memory image. Given the system's 24-bit word size, it can address a up to 2^{24} memory locations with a single word. This rounds out to about 16.8 million words or 50.3 megabytes of potential memory. To take full advantage of this, the memory image would have to be scaled up to a resolution of 4,096px by 4,096px.

The only real challenge in increasing the size of the memory image is the increase in time required to render the memory image using Cycles. The path-tracing nature of Cycles puts a massive bottleneck on the system, so some sort of optimization or workaround will have to be set in place should higher resolution memory images be implemented. Until then, low-resolution memory images will suffice.

2.3 Register File

The BARNDLES register file is composed of 16 registers that each hold a 24-bit word. Like all other memory, the BARNDLES register file is part of the memory image spanning from `0x000` to `0x00F`. The registers are organized sequentially within this section of memory such that `R0` is stored at `0x000`, `R1` is stored at `0x001`, and so on until we reach the `CPSR` which is stored at `0x00F`.

Of the 16 registers, 12 are general-purpose registers, and the other 4 are system registers.

Type	Register	Function
General Purpose Registers	R0	Return Register
	R1	First Parameter Register
	R2	Second Parameter Register
	R3	Third Parameter Register
	R4	Fourth Parameter Register
	R5	
	R6	
	R7	
	R8	
	R9	
	R10	
	R11	
System Registers	SP	Stack Pointer
	LR	Link Register
	PC	Program Counter
	CPSR	Current Program Status Register

Figure 2.4: The BARNDLES registers and their functions.

2.3.1 General-Purpose Registers

BARNDLES has 12 general-purpose registers `R0-R11`. General-purpose registers are designed to be used and modified by the programmer. This means that no system event will ever modify the values within these registers, apart from system resets.

Although Figure 1.1 details specific functions for general-purpose registers `R0-R4`, the registers are functionally identical to all other general-purpose registers in the register file. These functions are calling conventions that the programmer can choose to follow if they wish to. Breaking these conventions will have no negative side effects on the architecture nor the execution of instructions.

Return Register

The return register `R0` holds return values from subroutines, so they are easily accessible to the calling routine.

Parameter Registers

The parameter registers `R1-R4` hold parameter values, so they can be reliably passed into subroutines.

2.3.2 System Registers

The other 4 registers `SP`, `LR`, `PC`, and `CPSR` are system registers. Although system registers maintain the same read and write accessibility as general-purpose registers, they should not be written to by the programmer. Many components within the Blender Microprocessor's architecture are dependent on the values stored in the system registers. Writing to them or otherwise altering their values can have serious side effects on the way the processor executes instructions, determines conditionals, and more.

Each of the system registers have a specific function. Unlike general-purpose registers, these functions are not conventions. They are defined by hardware and must be acknowledged by the programmer.

Stack Pointer

The stack pointer `SP` stores the offset from the stack base to the position of the last element stored in the stack.

Link Register

The link register `LR` stores the last value of the program counter when the last branch and link instruction was executed.

Program Counter

The program counter `PC` stores the offset from the instruction memory base to the position that the next instruction should be fetched from.

Current Program Status Register

The current program status register `CPSR` stores the ALU status flags C, V, N, and Z.

2.4 Other Memory

The read-write memory section is partitioned into two smaller subsections of memory, stack memory and read-write data memory. These subsections serve specific purposes to the system and the programmer, so their locations, functions, and conventions are defined in full below.

2.4.1 Stack Memory

Stack memory is a section of memory managed used to temporarily store register values.

Stack memory follows the guidelines of the stack data structure. Elements are pushed and popped from the top of the stack in first-in-last-out order, and elements within the body of the stack are inaccessible unless the elements on top of it are popped out first.

The current offset into stack memory is stored in the Stack Pointer register `SP` within the BARNDLES register file. Every time a PUSH

While this value is accessible to the programmer, it should not be used to access stack memory manually, nor should it ever be modified by the programmer.

As the name suggests, stack memory is a stack data structure, so elements can only be pushed and popped off the stack in first-in-last-out order. Elements within the stack cannot be randomly accessed.

2.4.2 Read-Write Data Memory

The system's read-write data memory (RWDM) is intended to store modifiable data for programs and occupies the memory image from address `0x200` to `0x3FF`. RWDM is allocated by the assembler when the program is assembled and cannot be reallocated at runtime, but the values stored within these allocated portions can be written to or otherwise modified.

RWDM should be used to store

2.4.3 Read-Only Data Memory

The system's read-only data memory (RODM) is intended to store unmodifiable data for programs and occupies the memory image from address `0x400` to `0x5FF`. RODM is allocated by the assembler when the program is assembled and cannot be reallocated nor written to at runtime.

RODM should be used to store unchanging values

2.4.4 Instruction Memory