AMO

Architecture Manual

32-bit RISC

Yechan Hong

[github.com/hyahong](https://github.com/hyahong)

Kwangwoon University Capstone

CONTENTS

CHAPTER 1 | AMO ARCHITECTURE

* 1. Order 15
  2. Mode 15
  3. Register 15

1.3.1 General Register 15

1.3.2 System Register 15

* 1. Pin 15
  2. Memory Map 15

1.5 Monitor 15

1.5.1 VGA 15

1.5.2 Color 15

built-in graphics

CHAPTER 2 | Motherboard

1.5 Monitor 15

1.5.1 VGA 15

1.5.2 Color 15

1.6 Interrupt 15

1.6.1 Interrupt Service Routine 15

1.6.2 Keyboard 15

CHAPTER 2 | AMO INSTRUCTION SET

2.1 Instruction Set Summary 15

2.1.1 Format Summary 15

2.2 Instruction 15

CHAPTER 3 | AMO TOOLCHAIN

2.1 Tool Chain For amo 15

CHAPTER1

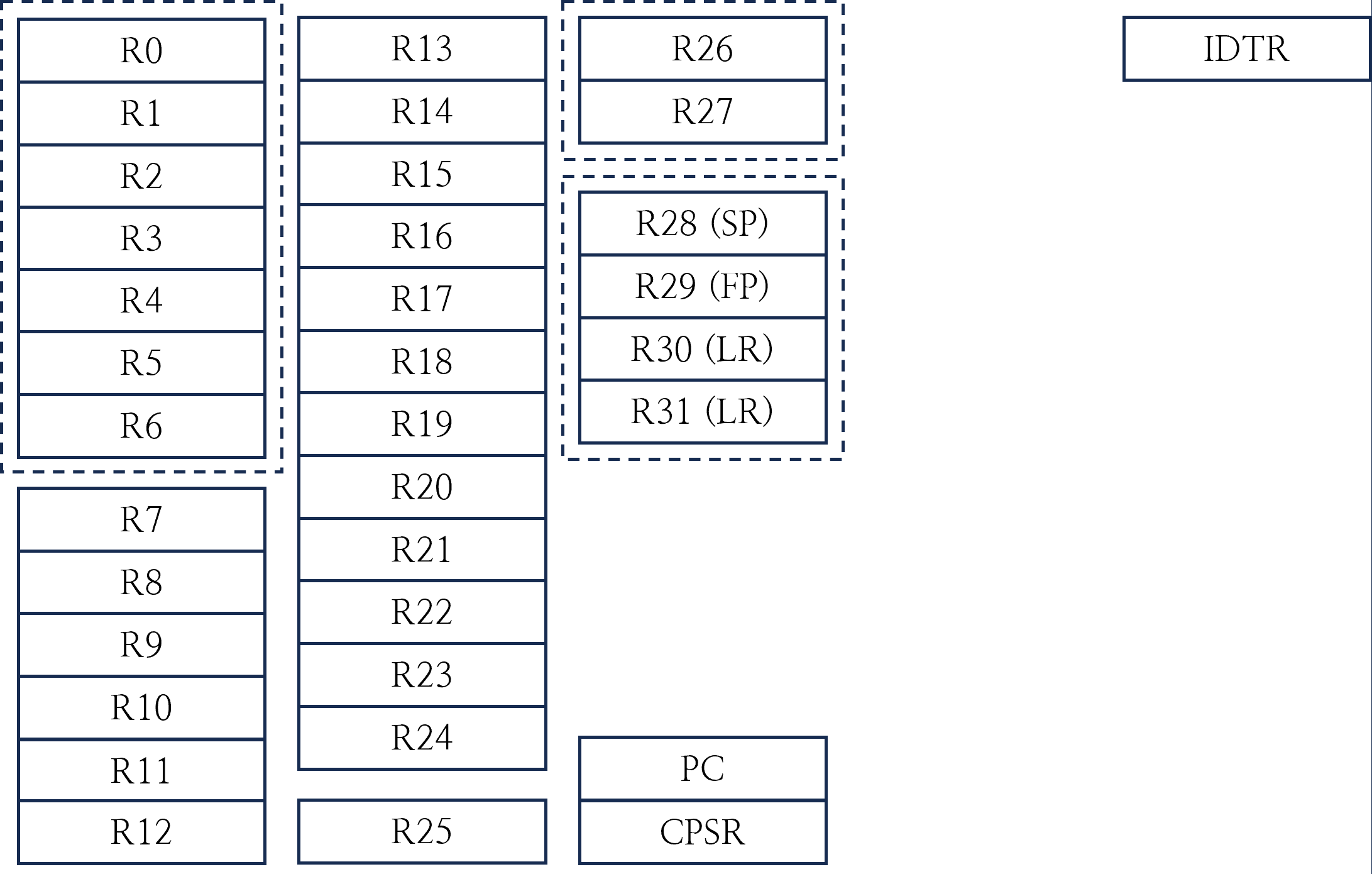
AMO ARCHITECTURE

This chapter describes the architecture of AMO. AMO is a 32-bit Central Processing Unit (CPU) with a Reduced Instruction Set Computing (RISC) architecture. This CPU was designed for capstone project, “Computer from Scratch”.

* 1. Order

This CPU uses Little Endian byte ordering. It is important to keep this in mind when developing programs for this CPU.

* 1. Register



* + 1. General Register

Registers below can be used in all instructions as Rs (Source), Rn (Operand) and Rd (Destination).

|  |  |  |
| --- | --- | --- |
| Register | Special | Role |
| R0 |  | Caller-saved/Result register |
| R1-R15 |  | Caller-saved registers |
| R16-R25 |  | Callee-saved registers |
| R26-R27 |  | Interrupt registers |
| R28 |  | Argument register |
| R29 | FP | Frame pointer |
| R30 | SP | Stack pointer |
| R31 | LR | Link register |

* + 1. System Register

Registers below are system registers that directly control the CPU AMO.

|  |  |
| --- | --- |
| Register | Role |
| CPSR | Current Program Status Register |
| IDTR | Interrupt Descriptor Table Register |

* + - 1. Current Program Status Register



Current Program Status Register (CPSR) contains flag information and interrupt handling details for the currently executing program.

* + - 1. Interrupt Descriptor Table Register



Interrupt Descriptor Table Register (IDTR) points to the location of the vector table. When an interrupt occurs, the CPU jumps to the location that is the IDTR value plus the offset. The offsets are reset 0x0, undefined 0x4, interrupt (trap) 0x8, and interrupt (HW) 0xC, respectively. IDTR is set to 0 at boot time.

* 1. Memory Map
  2. Monitor

AMO has built-in graphics card to support monitor. To use monitor, user connects the VGA cable. CPU has a total of two modes and determines the mode to use with ‘Graphics Mode Bit’ of CR0. If it is zero, CPU uses Text Mode, otherwise use Graphic Mode.

TODO: explain how to change the graphics mode

* + 1. VGA

12-bit DAC

4-bit 4-bit 4-bit

* + 1. Color

Foreground Color | Backgroud Color

4-bit | 4-bit

16 =

0: Black 0 0 0 | 0 0 0

1: Blue 0 0 170 | 0 0 11

2: Green 0 170 0 | 0 11 0

3: Cyan 0 170 170 | 0 11 11

4: Red 170 0 0 | 11 0 0

5: Magenta 170 0 170 | 11 0 11

6: Brown 170 85 0 | 11 5 0

7: White 170 170 170 | 11 11 11

8: Gray 85 85 85 | 5 5 5

9: Light Blue 85 85 255 | 5 5 15

A: Light Green 85 255 85 | 5 15 5

B: Light Cyan 85 255 255 | 5 15 15

C: Light Red 255 85 85 | 15 5 5

D: Light Magenta 255 85 255 | 15 5 15

E: Yellow 255 255 85 | 15 15 5

F: Bright White 255 255 255 | 15 15 15

텍스트, 다채로움, 스크린샷, 마젠타이(가) 표시된 사진

자동 생성된 설명

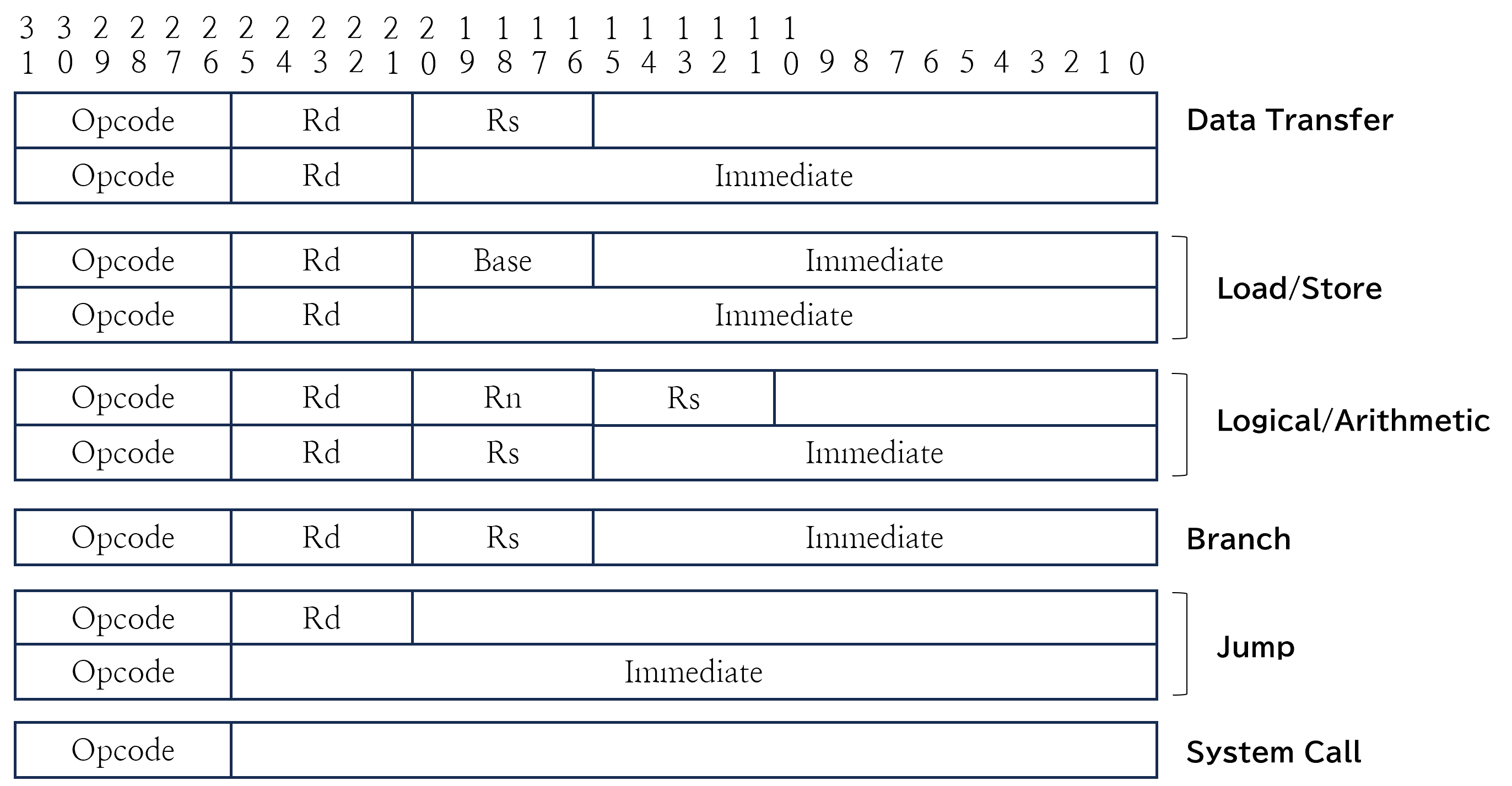
CHAPTER2

AMO INSTRUCTION SET

This chapter describes an instruction set of the AMO architecture and provides a brief overview before examining each section in detail.

* 1. Instruction Set Summary
     1. Format Summary

The AMO instruction set formats are shown below.



* + 1. Instruction Summary

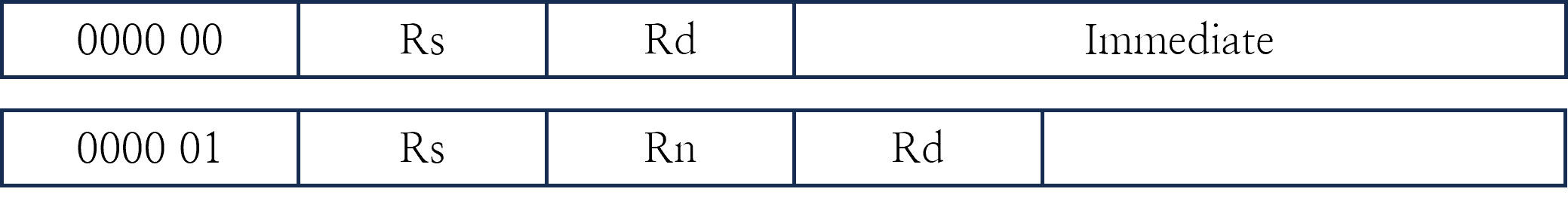
**DEFAULT**

|  |  |  |
| --- | --- | --- |
| Mnemonic | Example | Instruction |
| MOV | mov Rd, Rs/imm21  mov Rd, imm32/symbol | Rd ← Rs/imm21  literal pool (pseudo) |
| LDR | ldr Rd, [base, offset]  ldr Rd, [relative] | Rd ← mem[base + offset]  Rd ← mem[pc + relative] |
| STR | str [base, offset], Rs  str [relative], Rs | mem[base + offset] ← Rs  mem[pc + relative] ← Rs |
| LDRB | ldrb Rd, [base, offset]  ldrb Rd, [relative] | Rd ← mem[base + offset]  Rd ← mem[pc + relative] |
| STRB | strb [base, offset], Rs  strb [relative], Rs | mem[base + offset] ← Rs  mem[pc + relative] ← Rs |
| LDRH | ldrh Rd, [base, offset]  ldrh Rd, [relative] | Rd ← mem[base + offset]  Rd ← mem[pc + relative] |
| STRH | strh [base, offset], Rs  strh [relative], Rs | mem[base + offset] ← Rs  mem[pc + relative] ← Rs |
| ADD | add Rd, Rn, Rs/imm16 | Rd ← Rn + Rs/imm16 |
| ADC | adc Rd, Rn, Rs/imm16 | Rd ← Rn + Rs/imm16 + Carry |
| SUB | sub Rd, Rn, Rs/imm16 | Rd ← Rn - Rs/imm16 |
| AND | and Rd, Rn, Rs/imm16 | Rd ← Rn AND Rs/imm16 |
| OR | or Rd, Rn, Rs/imm16 | Rd ← Rn OR Rs/imm16 |
| XOR | xor Rd, Rn, Rs/imm16 | Rd ← Rn XOR Rs/imm16 |
| NOT | not Rd, Rs/imm16 | Rd ← NOT Rs/imm16 |
| LSL | lsl Rd, Rn, Rs/imm16 | Rd ← Rn << Rs/imm16 |
| LSR | lsr Rd, Rn, Rs/imm16 | Rd ← Rn >> Rs/imm16 |
| ASR | asr Rd, Rn, Rs/imm16 | Rd ← Rn >>> Rs/imm16 |
| BEQ | beq Rd, Rs, imm16 | PC ← PC + (imm16 << 2) if Rd == Rs |
| BNE | bne Rd, Rs, imm16 | PC ← PC + (imm16 << 2) if Rd != Rs |
| BLT | blt Rd, Rs, imm16 | PC ← PC + (imm16 << 2) if Rd < Rs |
| BLE | ble Rd, Rs, imm16 | PC ← PC + (imm16 << 2) if Rd <= Rs |
| BLTU | bltu Rd, Rs, imm16 | PC ← PC + (imm16 << 2) if Rs < Rn |
| BLEU | bleu Rd, Rs, imm16 | PC ← PC + (imm16 << 2) if Rs <= Rn |
| JMP | jmp Rs/imm26 | PC = Rs/(imm26 << 2) |
| JAL | jal Rs/imm26 | PC = (imm26 << 2)  LR = PC |
| SWI | swi imm6 | Jump to Interrupt Vector (Trap) |
| EXT | ext Rd, Rs, Opt | Rd ← [Sign/Unsign]Extend (Rs) |
| SETVT | setvt Rs, Type | Set the Vector Table  Type 0: Interrupt Vector Table |
| RET | ret Rs | PC = Rs/(imm26 << 2)  InterruptBlocking = false |
| LOCK | lock | InterruptBlocking = !InterruptBlocking  (for atomic operation) |

**KERNEL**

* 1. Instruction
     1. NOP/ADD

|  |  |  |
| --- | --- | --- |
| ADD | add Rd, Rn, Rs/imm16 | Rd ← Rn + Rs/imm16 |



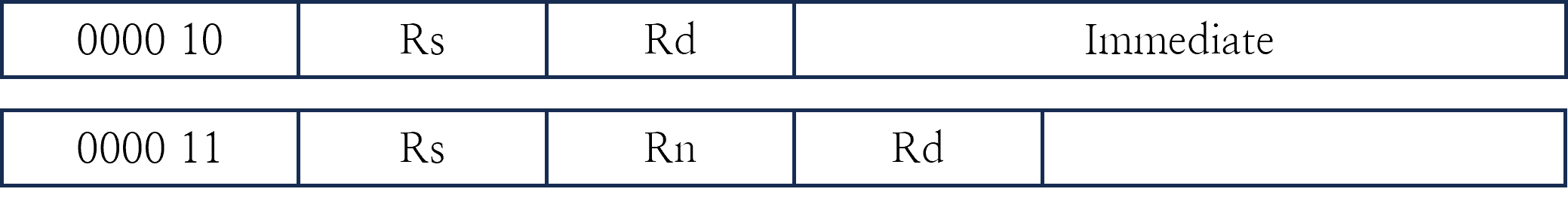
00: Add imm16 to Rs and store it in Rd.

01: Add a value of Rs to Rn and store it in Rd.

|  |
| --- |
| NOTE |
| For NOP, all bits are 0, so the CPU does R0 ← R0 + 0 |

* + 1. ADC

|  |  |  |
| --- | --- | --- |
| ADC | adc Rd, Rn, Rs/imm16 | Rd ← Rn + Rs/imm16 + Carry |



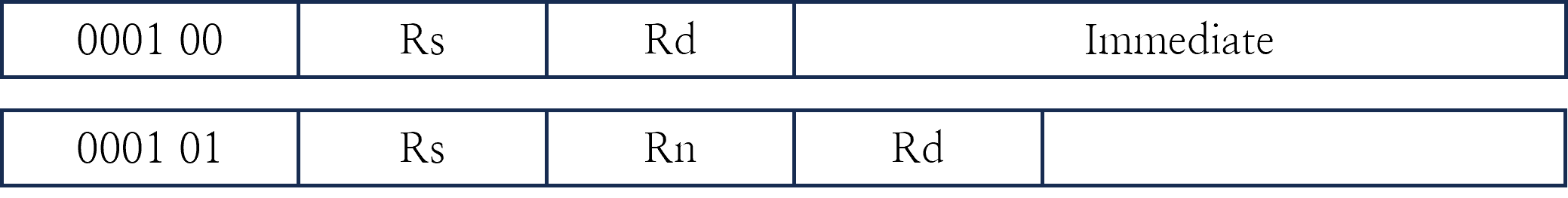
02: Add imm16 to Rs with carry and store it in Rd.

03: Add a value of Rs to Rn with carry and store it in Rd.

|  |
| --- |
| NOTE |
| Carry is set by previous operation. |

* + 1. SUB

|  |  |  |
| --- | --- | --- |
| SUB | sub Rd, Rn, Rs/imm16 | Rd ← Rs - Rn/imm16 |

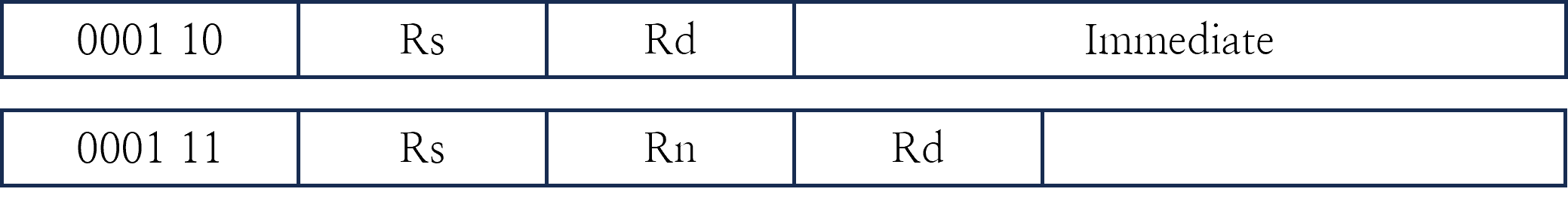


04: Subtract imm16 from Rs and store it in Rd.

05: Subtract a value of Rs from Rn and store it in Rd.

* + 1. AND

|  |  |  |
| --- | --- | --- |
| AND | and Rd, Rn, Rs/imm16 | Rd ← Rn AND Rs/imm16 |

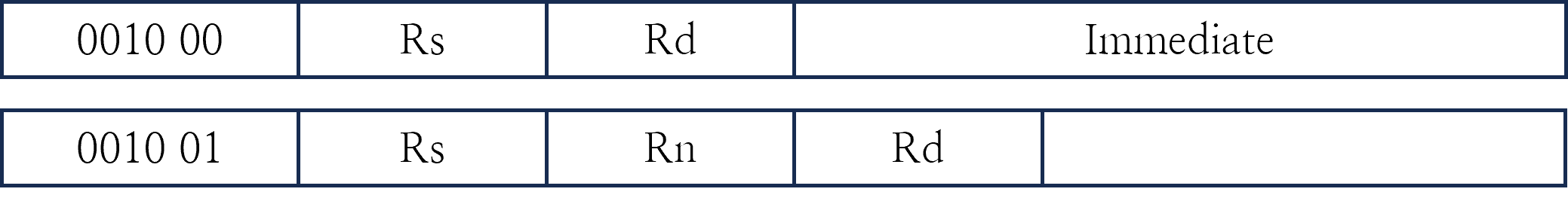


06: Perform the AND operation with imm16 and Rs, and store it in Rd.

07: Perform the AND operation with Rs and Rn, and store it in Rd.

* + 1. OR

|  |  |  |
| --- | --- | --- |
| OR | or Rd, Rn, Rs/imm16 | Rd ← Rn OR Rs/imm16 |

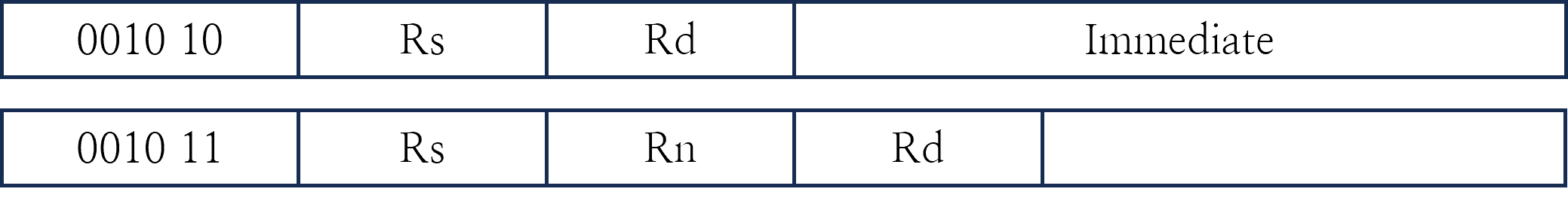


08: Perform the OR operation with imm16 and Rs, and store it in Rd.

09: Perform the OR operation with a value of Rs and Rn, and store it in Rd.

* + 1. XOR

|  |  |  |
| --- | --- | --- |
| XOR | xor Rd, Rn, Rs/imm16 | Rd ← Rn XOR Rs/imm16 |

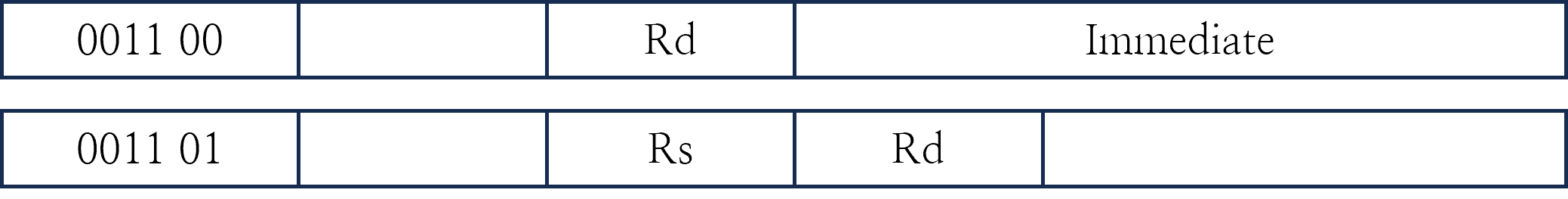


0A: Perform the XOR operation with imm16 and Rs, and store it in Rd.

0B: Perform the XOR operation with a value of Rs and Rn, and store it in Rd.

* + 1. NOT

|  |  |  |
| --- | --- | --- |
| NOT | not Rd, Rs/imm16 | Rd ← NOT Rs/imm16 |

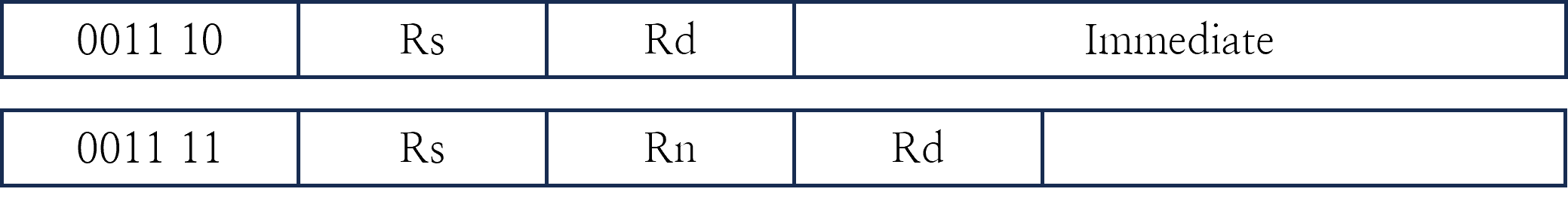


0C: Perform the NOT operation on imm16 and store it in Rd.

0D: Perform the NOT operation on a value of Rs and store it in Rd.

* + 1. LSL

|  |  |  |
| --- | --- | --- |
| LSL | lsl Rd, Rn, Rs/imm16 | Rd ← Rn << Rs/imm16 |

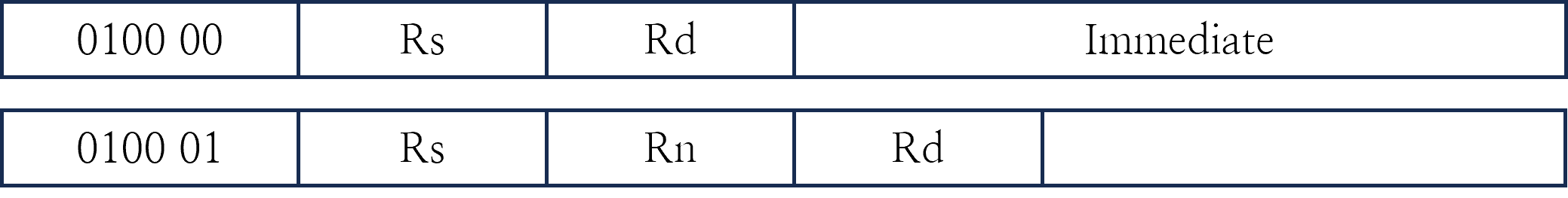


0E: Shift Rs to the left by imm16 and store it in Rd.

0F: Shift Rn to the left by Rs and store it in Rd.

* + 1. LSR

|  |  |  |
| --- | --- | --- |
| LSR | lsr Rd, Rn, Rs/imm16 | Rd ← Rn >> Rs/imm16 |



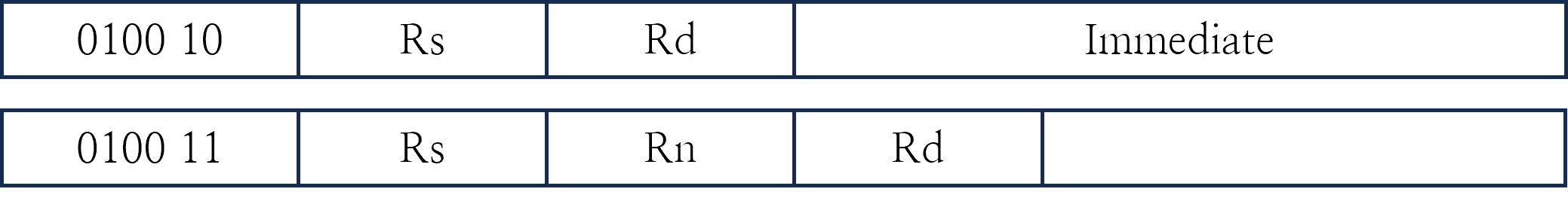
10: Shift Rs to the right by imm16 and store it in Rd.

11: Shift Rn to the right by Rs and store it in Rd.

|  |
| --- |
| NOTE |
| In this operation, MSB is set to zero |

* + 1. ASR

|  |  |  |
| --- | --- | --- |
| ASR | asr Rd, Rn, Rs/imm16 | Rd ← Rn >>> Rs/imm16 |



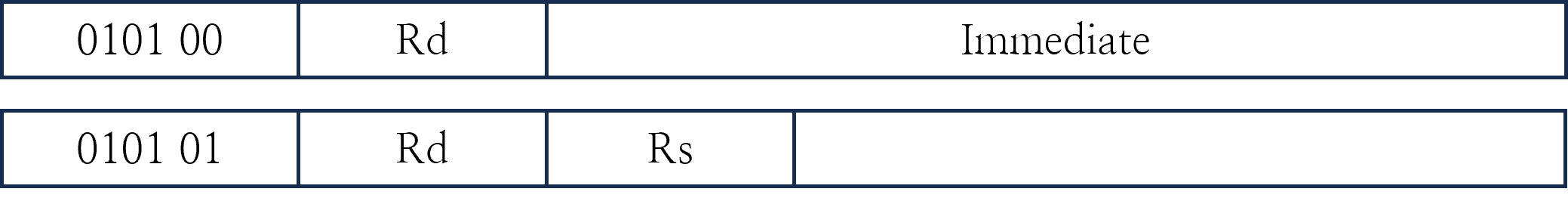
12: Shift Rs to the right by imm16 and store it in Rd.

13: Shift Rn to the right by Rs and store it in Rd.

|  |
| --- |
| NOTE |
| In this operation, MSB is set to pre-MSB |

* + 1. MOV

|  |  |  |
| --- | --- | --- |
| MOV | mov Rd, Rs/imm21  mov Rd, imm32/symbol | Rd ← Rs/imm21  \*literal pool (pseudo) |

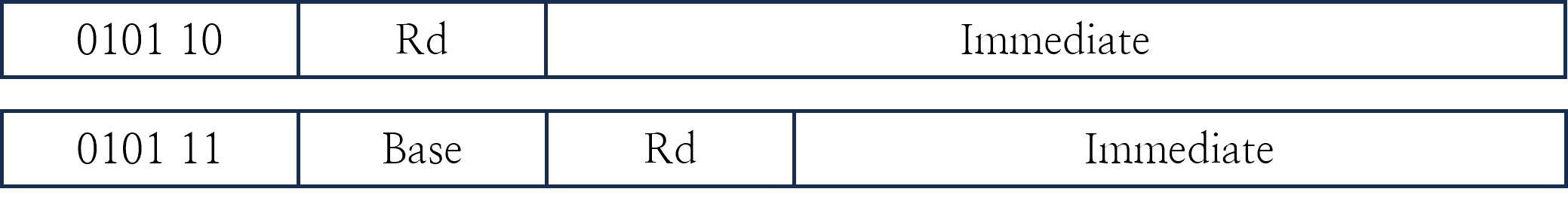


14: Copy imm21 to Rd.

15: Copy a value of Rs to Rd.

* + 1. LDR

|  |  |  |
| --- | --- | --- |
| LDR | ldr Rd, [base, offset]  ldr Rd, [relative] | Rd ← mem[base + offset]  Rd ← mem[pc + relative] |

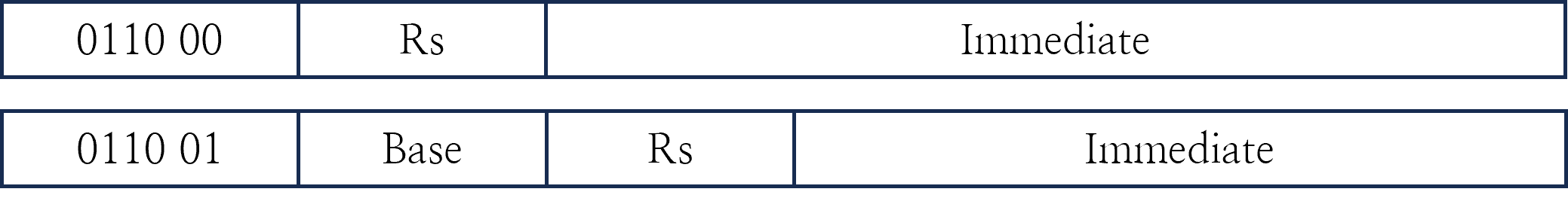


16: Read from memory at ‘pc + imm21’ and store the result in Rd.

17: Read from memory at ‘the value of base + imm16’ and store the result in Rd.

* + 1. STR

|  |  |  |
| --- | --- | --- |
| STR | str [base, offset], Rs  str [relative], Rs | mem[base + offset] ← Rs  mem[pc + relative] ← Rs |

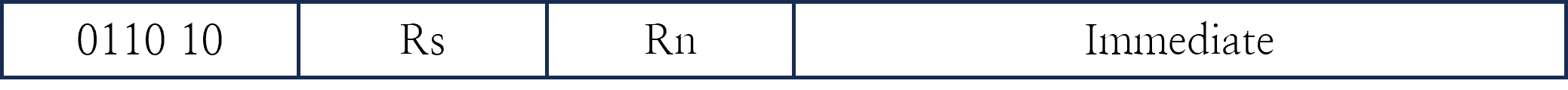


18: Write the value of Rd to memory at ‘pc + imm21’.

19: Write the value of Rd to memory at ‘the value of base + imm16’.

* + 1. BEQ

|  |  |  |
| --- | --- | --- |
| BEQ | beq Rs, Rn, imm16 | PC ← PC + (imm16 << 2) if Rs == Rn |

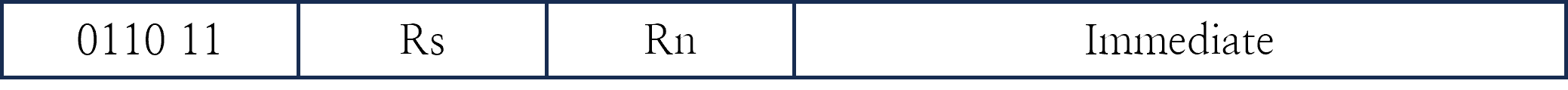


1A: Add imm16 << 2 to PC if Rs is equal to Rn.

|  |
| --- |
| NOTE |
| branch instruction shifts imm16 for memory address alignment and operates a signed addition. |

* + 1. BNE

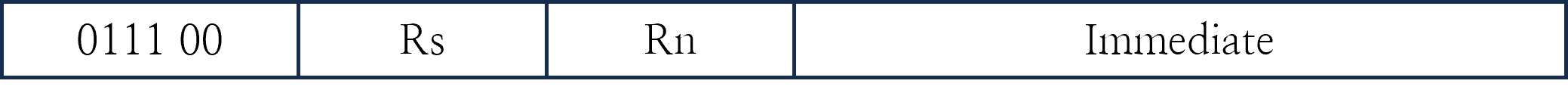
|  |  |  |
| --- | --- | --- |
| BNE | bne Rd, Rs, imm16 | PC ← PC + (imm16 << 2) if Rs != Rn |



1B: Add imm16 << 2 to PC if Rs is not equal to Rn.

* + 1. BLT

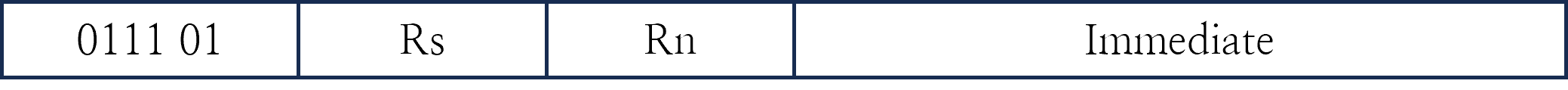
|  |  |  |
| --- | --- | --- |
| BLT | blt Rd, Rs, imm16 | PC ← PC + (imm16 << 2) if Rs < Rn |



1C: Add imm16 << 2 to PC if Rs is less than Rd.

* + 1. BLE

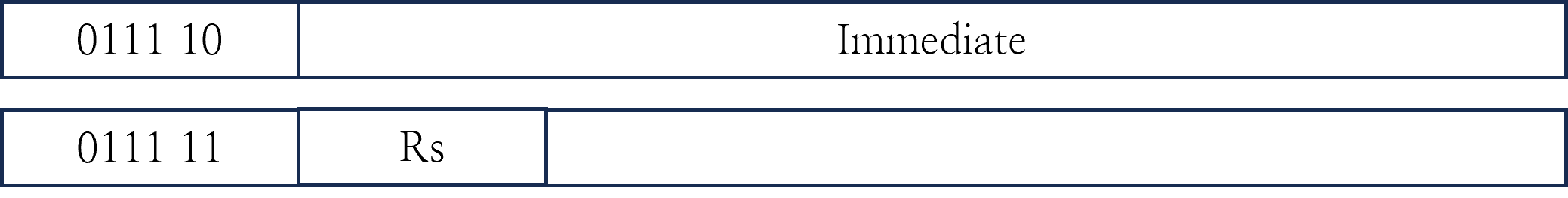
|  |  |  |
| --- | --- | --- |
| BLE | ble Rd, Rs, imm16 | PC ← PC + (imm16 << 2) if Rs <= Rn |



1D: Add imm16 << 2 to PC if Rs is less than or equal to Rd.

* + 1. J

|  |  |  |
| --- | --- | --- |
| JMP | jmp Rs/imm26 | PC = Rs/(imm26 << 2) |

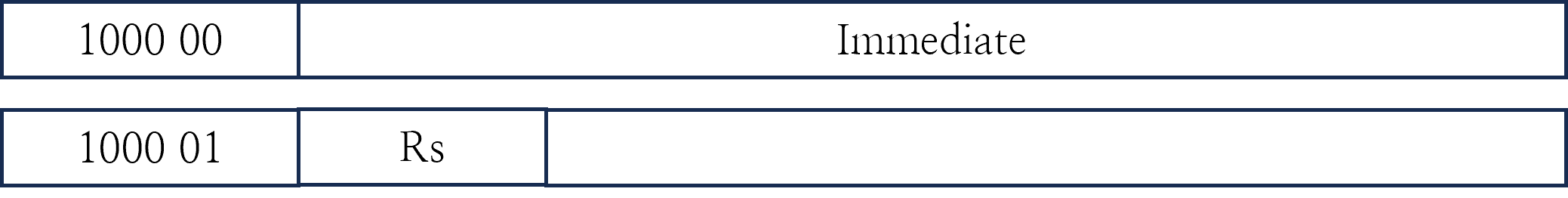


1E: Jump to an absolute location. the address is composed of the upper 4 bits of the PC and imm26 << 2.

1F: Jump to the value of Rs. since the value is 32-bits, this can be used to jump for any address location.

* + 1. JAL

|  |  |  |
| --- | --- | --- |
| JAL | jal Rs/imm26 | PC = (imm26 << 2)  LR = PC + 4 |



20: Jump to an absolute location and store PC + 4 to LR. the address is composed of the upper 4 bits of the PC and imm26 << 2.

21: Jump to the value of Rs and store PC + 4 to LR. since the value is 32-bits, this can be used to jump for any address location.

* + 1. SWI

|  |  |  |
| --- | --- | --- |
| SWI | swi imm8 | Jump to Interrupt Vector |

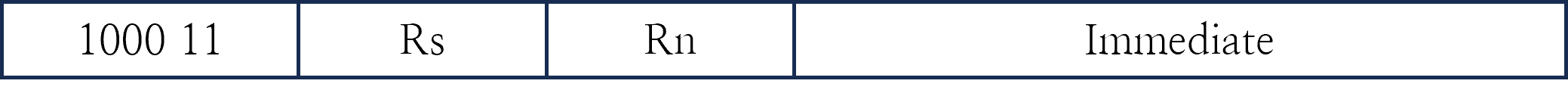
스크린샷, 라인, 직사각형, 텍스트이(가) 표시된 사진

자동 생성된 설명

22: Jump to System Call Routine. imm6 should be handled in a syscall routine.

* + 1. BLTU

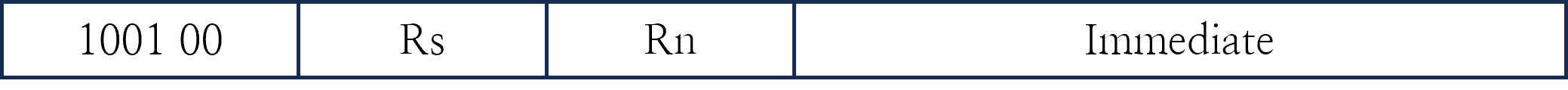
|  |  |  |
| --- | --- | --- |
| BLTU | bltu Rd, Rs, imm16 | PC ← PC + (imm16 << 2) if Rs < Rn |



23: Add imm16 << 2 to PC if Rs is less than Rd.

* + 1. BLEU

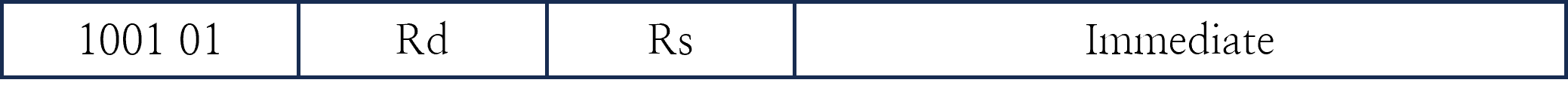
|  |  |  |
| --- | --- | --- |
| BLEU | bleu Rd, Rs, imm16 | PC ← PC + (imm16 << 2) if Rs <= Rn |



24: Add imm16 << 2 to PC if Rs is less than or equal to Rd.

* + 1. EXT

|  |  |  |
| --- | --- | --- |
| EXT | ext Rd, Rs, Opt | Rd ← Extend (Rs) |



25: Extension

Imm 0: 8-bit logical extension

Imm 1: 8-bit arithmetic extension

Imm 2: 16-bit logical extension

Imm 3: 16-bit arithmetic extension