EE511 Handouts

KAIST RISC PROCESSOR 2.0 (KRP)
USER'S MANUAL

2022 FALL

INTEGRATED COMPUTER SYSTEMS LAB.

School of EE, KAIST

1. Register Sets and Memory

1.1. Special-Purpose Registers

1.1.1. PC <31..0> (program counter)

PC holds the address to fetch the next instruction in the program memory. It increases by 4 after an instruction is fetched. Instructions that are allowed to access or modify **PC** are J, JL, BR, BRL, RFI, LDR, STR, and LEA. The **PC** value used as an operand (`currentPC`) in these instructions is [the memory address of those instructions + 4].

1.1.2. IR <31..0> (instruction register)

IR holds the current instruction to execute. No instruction is allowed to modify it directly.

1.1.3. IE (interrupt enable)

IE is a 1-bit flag and indicates whether the control may recognize exceptions (1) or not (0). Since KRP cannot process nested exceptions, as the control enters the exception service routine it clears **IE** so that no more exceptions are recognized. Instructions that are allowed to modify **IE** are EEN and EDI.

1.1.4. IPC <31..0> (interrupted PC)

IPC contains the **PC** value of the interrupted process so that the control may return from the exception service routine by copying **IPC** to **PC**. IPC is mapped to GPR[31].

1.2. General-Purpose Registers

KRP has 32 general-purpose registers (GPRs), all 32-bit long. They are not included in the data memory address space, thus they cannot be accessed using memory addresses. Instead, most instructions are given one or more 5-bit fields to contain the indices of GPRs, so that they may address the registers explicitly.

1.3. Memory

KRP has separate address spaces for program and data memory. The program memory can only be read while the data memory can be read and written. Both memories can hold up to 4 GBytes of directly addressable capacity. They are byte-addressable; although there is no instruction that is capable of access them in a byte-wise manner. The size of a word is 32 bits. (See Fig. 1)

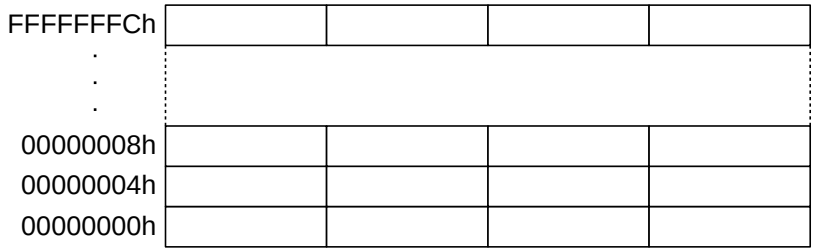


Fig. 1 Main memory map

2. Interface Signals

Fig. 2 depicts the input, output, and bi-directional signals of KRP. In the figure, wide arrows mean busses, in other words, they occupy more than one bit. The arrows pointing inward the box represent input signals, those pointing outward represent output signals, and that with heads on both directions represents a bi-directional signal.

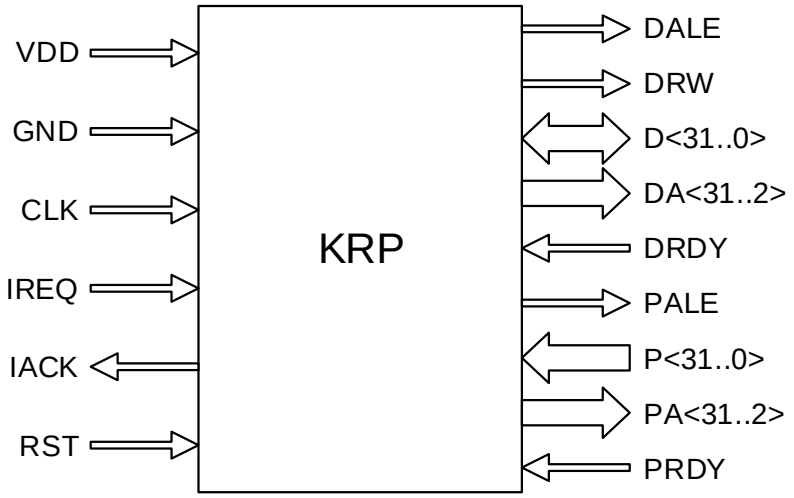


Fig. 2 Interface signals of KRP

The reason why the address lines consist only of 30 bits, rather than full 32 bits, is that the data of both program and data memories are accessed only in words. This means, in turn, that the one who implements KRP may enhance the address lines if he or she adds instructions that byte-addresses memories and if the number of pins of the chip package allows that.

The signal descriptions are as follows:

Signal name	In/Out	Width [bits]	Meaning
CLK	In	1	Clock
RST	In	1	Reset
IREQ	In	1	Interrupt request
IACK	Out	1	Interrupt acknowledge
DALE	Out	1	Data address latch enable
DRW	Out	1	0:read / 1:write
D	In/Out	32	Data
DA	Out	30	Data address
DRDY	In	1	Data ready
PALE	Out	1	Program address latch enable
P	In	32	Program
PA	Out	30	Program address
PRDY	In	1	Program ready

3. Reset Procedures

The reset signal, **Rst**, is external.

When the **Rst** signal goes high while the machine is running, the reset procedure is invoked to

make the following changes:

- Interrupt enable is set to zero.

  IE ← 0

- **PC** is set to zero.

  PC ← 00000000h

When the **Rst** signal goes low, the machine starts running by fetching the first instruction indicated by the PC.

4. Instruction Sets

NOTICE

In this instruction set summary, the notations and representations are similar with those of C language and Verilog HDL and thus are different with those in the reference[†]. The used operators are as follows:

Operator	Meaning
+	add
-	subtract or 2's complement
&	logical AND
|	logical OR
^	logical exclusive-OR
~	logical inversion (1's complement)
=	substitution
==	equal to
>	greater than
<	smaller than
>=	greater than or equal
<=	smaller than or equal
!=	Not equal
<<	shift left
>>	shift right
signExt()	sign-extension to 32 bits
zeroExt()	zero-extension to 32 bits

However, the bit-field notation is the same as that of reference, since Verilog's ***bit-field notation*** is likely to be confused with the array representation, e.g. R[rb]. Some examples are presented below.

R[4]<8> : The bit 8 of register 4.
M[1020]<4:0> : From bit 4 to bit 0 of the memory word at address 1020.

Furthermore, ***when a bit is repeated*** for arbitrary times, it is represented like below, according to the convention of Verilog-HDL.

{n{R[3]<29>}} : The bit 29 of register 3 is repeated n times.

Concatenation of bits is also represented after the Verilog notation, i.e. each comprising bit field distinguished by commas and overall bits embraced by

[†] V. P. Heuring and H. F. Jordan, "Computer Systems Design and Architecture," Prentice Hall, 2003

bracelets. The following is an example.

{a<3..0>, b<7..4>} : Concatenation of lower nibble of a and higher nibble of b.

The switch modes that are used in data-processing operands are as follows:

Switch modes	**Meaning**
SHL(signExt(**A**), shamt)	Shift left the sign extended **A** by shamt
LSR(signExt(**A**), shamt)	Logical shift right the sign extended **A** by shamt
ASR(signExt(**A**), shamt)	Arithmetic shift right the sign extended **A** by shamt
ROR(signExt(**A**), shamt)	Rotate right the sign extended **A** by shamt

The condition code used in branch instructions is as follows:

Condition	**Meaning**
NV	Never
AL	Always
EQ	Equal zero
NE	Not equal zero
GE	Greater or equal than zero
LT	Less than zero

ADDI (Add Immediate)

ASSEMBLY

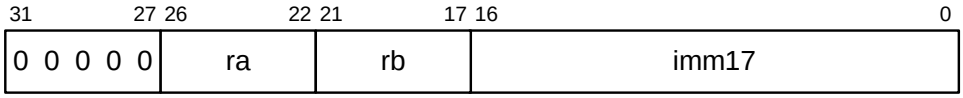
ADDI ra, rb, imm17
ADDI ra, rb, shift(imm10, shamt)

DESCRIPTION

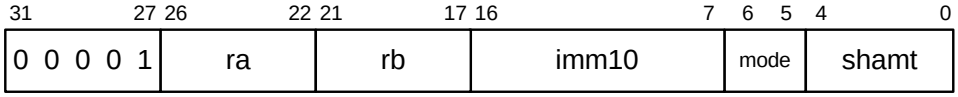
The ADDI instruction performs a 2's complement signed addition where one operand is a general-purpose register value and the other is a 32-bit immediate value. Since the addition is *signed*, the shorter operand is sign-extended to match the bit-length of longer one. The result of the addition is stored in a general-purpose register, ra.

FORMAT

Type 1)



Type 2)



OPERATION

Type 1)

```
R[ra] = R[rb] + signExt(imm17);
```

Type 2)

```
switch(mode) {
   case 2'b00: R[ra] = R[rb] + SHL(signExt(imm10), shamt};
   case 2'b01: R[ra] = R[rb] + LSR(signExt(imm10), shamt};
   case 2'b10: R[ra] = R[rb] + ASR(signExt(imm10), shamt};
   case 2'b11: R[ra] = R[rb] + ROR(signExt(imm10), shamt};
}
```

EXAMPLE

ADDI r1, r2, #2
ADDI r1, r2, SHL(#0x3, #5)

ORI (Or Immediate)

ASSEMBLY

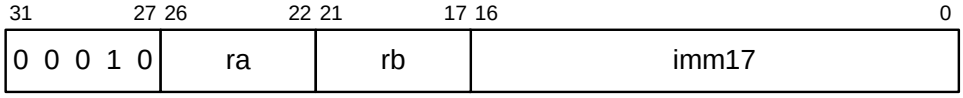
ORI ra, rb, imm17
ORI ra, rb, shift(imm10, shamt)

DESCRIPTION

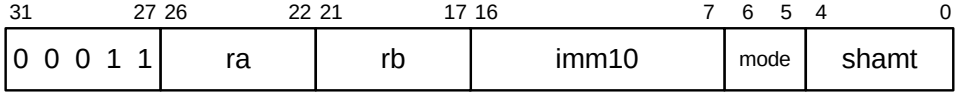
The ORI instruction performs a logical bit-wise OR operation where one operand is a general-purpose register value and the other is a 32-bit immediate value. The immediate operand is sign-extended to match the bit-length of the GPR operand. The result of the addition is stored in a general-purpose register, ra.

FORMAT

Type 1)



Type 2)



OPERATION

Type 1)

```
R[ra] = R[rb] | signExt(imm17);
```

Type 2)

```
switch(mode) {
   case 2'b00: R[ra] = R[rb] | SHL(signExt(imm10), shamt};
   case 2'b01: R[ra] = R[rb] | LSR(signExt(imm10), shamt};
   case 2'b10: R[ra] = R[rb] | ASR(signExt(imm10), shamt};
   case 2'b11: R[ra] = R[rb] | ROR(signExt(imm10), shamt};
}
```

EXAMPLE

ORI r10, r11, #0x11
ORI r10, r11, ASR(#0x330, #4)

ANDI (And Immediate)

ASSEMBLY

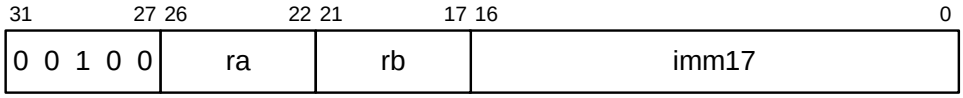
ANDI ra, rb, imm17
ANDI ra, rb, shift(imm10, shamt)

DESCRIPTION

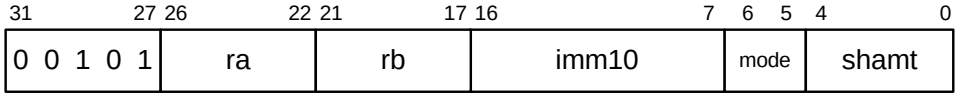
The ANDI instruction performs a logical bit-wise AND operation where one operand is a general-purpose register value and the other is a 32-bit immediate value. The immediate operand is sign-extended to match the bit-length of the GPR operand. The result of the addition is stored in a general-purpose register, ra.

FORMAT

Type 1)



Type 2)



OPERATION

Type 1)

```
R[ra] = R[rb] & signExt(imm17);
```

Type 2)

```
switch(mode) {
   case 2'b00: R[ra] = R[rb] & SHL(signExt(imm10), shamt};
   case 2'b01: R[ra] = R[rb] & LSR(signExt(imm10), shamt};
   case 2'b10: R[ra] = R[rb] & ASR(signExt(imm10), shamt};
   case 2'b11: R[ra] = R[rb] & ROR(signExt(imm10), shamt};
}
```

EXAMPLE

ANDI r1, r2, #2
ANDI r1, r2, LSR(#0xFF, #8)

MOVI (Move Immediate)

ASSEMBLY

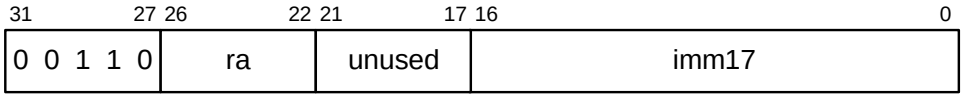
MOVI ra, imm17
MOVI ra, shift(imm10, shamt)

DESCRIPTION

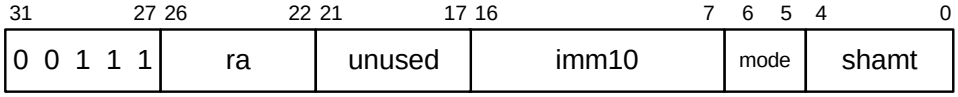
The MOVI instruction moves the 32-bit immediate value into a general-purpose register, ra. The immediate operand is sign-extended to match the bit-length of the destination GPR.

FORMAT

Type 1)



Type 2)



OPERATION

Type 1)

```
R[ra] = signExt(imm17);
```

Type 2)

```
switch(mode) {
   case 2'b00: R[ra] = SHL(signExt(imm10), shamt};
   case 2'b01: R[ra] = LSR(signExt(imm10), shamt};
   case 2'b10: R[ra] = ASR(signExt(imm10), shamt};
   case 2'b11: R[ra] = ROR(signExt(imm10), shamt};
}
```

EXAMPLE

MOVI ra, #0x11
MOVI ra, ROR(#0x1, 1)

ADD

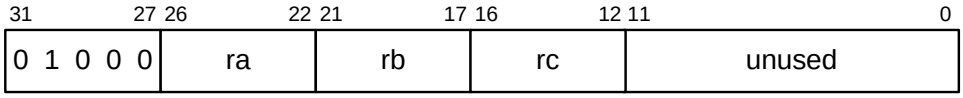
ASSEMBLY

ADD ra, rb, rc

DESCRIPTION

The ADD instruction performs a 2's complement signed addition for two general-purpose register values, rb and rc. The result is stored in a general-purpose register, ra.

FORMAT



OPERATION

```
R[ra] = R[rb] + R[rc];
```

EXAMPLE

ADD r1, r2, r3

SUB (Subtract)

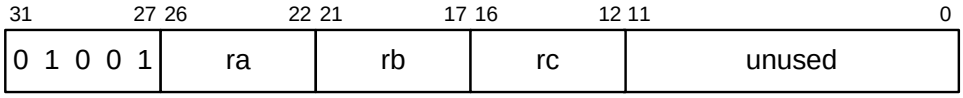
ASSEMBLY

SUB ra, rb, rc

DESCRIPTION

The SUB instruction performs a 2's complement signed subtraction for two general-purpose register values, rb and rc. The result is stored in a general-purpose register, ra. More precisely, rc is subtracted from rb.

FORMAT



OPERATION

```
R[ra] = R[rb] - R[rc];
```

EXAMPLE

SUB r2, r3, r4

NOT

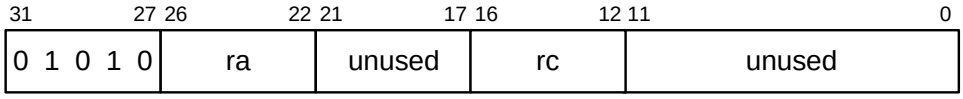
ASSEMBLY

NOT ra, rc

DESCRIPTION

The NOT instruction performs a 1's complement inversion for a general-purpose register value, rc. The result is stored in a general-purpose register, ra.

FORMAT



OPERATION

```
R[ra] = ~R[rc];
```

EXAMPLE

NOT r22, r7

NEG (Negate)

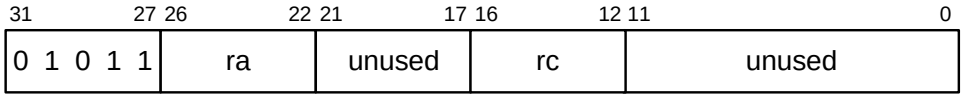
ASSEMBLY

NEG ra, rc

DESCRIPTION

The NEG instruction performs a 2's complement negation for a general-purpose register value, rc. The result is stored in a general-purpose register, ra.

FORMAT



OPERATION

```
R[ra] = −R[rc];
```

EXAMPLE

NEG r12, r16

OR

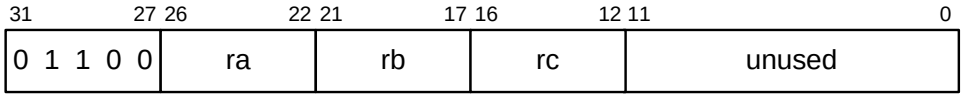
ASSEMBLY

OR ra, rb, rc

DESCRIPTION

The OR instruction performs a logical bit-wise or operation for two general-purpose register values, rb and rc. The result is stored in a general-purpose register, ra.

FORMAT



OPERATION

```
R[ra] = R[rb] | R[rc];
```

EXAMPLE

OR r10, r11, r12

AND

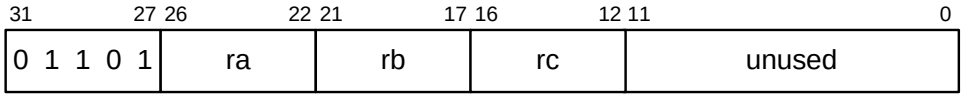
ASSEMBLY

AND ra, rb, rc

DESCRIPTION

The AND instruction performs a logical bit-wise and operation for two general-purpose register values, rb and rc. The result is stored in a general-purpose register, ra.

FORMAT



OPERATION

```
R[ra] = R[rb] & R[rc];
```

EXAMPLE

AND r1, r2, r3

XOR (Exclusive Or)

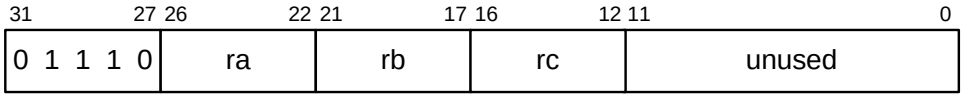
ASSEMBLY

XOR ra, rb, rc

DESCRIPTION

The XOR instruction performs a logical bit-wise exclusive-or operation for two general-purpose register values, rb and rc. The result is stored in a general-purpose register, ra.

FORMAT



OPERATION

```
R[ra] = R[rb] ^ R[rc];
```

EXAMPLE

XOR r2, r3, r22

ASR (Arithmetic Shift Right)

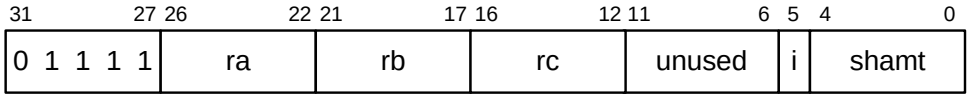
ASSEMBLY

ASR ra, rb, shamt
ASR ra, rb, rc

DESCRIPTION

The ASR performs shifts the content of a general-purpose register, rb, arithmetically to the right by a given amount and store the result in a general-purpose register, ra. If the i bit is 0, the shift amount is an immediate value, specified in the shamt field, ranging from 0 to 31. Otherwise the shift amount is the least significant 5-bit value in a general-purpose register, rc.
Note that this shift operation is *arithmetic*, thus sign-extension is performed.

FORMAT



OPERATION

```
if(i == 0)
     R[ra] = {{(shamt){R[rb]<31>}}, R[rb]<31..shamt>};
else
     R[ra] = {{(R[rc]<4..0>){R[rb]<31>}},
               R[rb]<31..R[rc] <4.. 0>>};
```

EXAMPLE

ASR r1, r2, #4
ASR r1, r2, r3

LSR (Logical Shift Right)

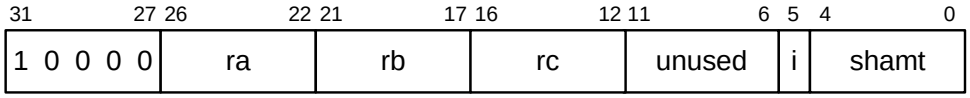
ASSEMBLY

LSR ra, rb, shamt
LSR ra, rb, rc

DESCRIPTION

The LSR shifts the content of a general-purpose register, rb, logically to the right by a given amount and store the result in a general-purpose register, ra. If the i bit is 0, the shift amount is an immediate value, specified in the shamt field, ranging from 0 to 31. Otherwise the shift amount is the least significant 5-bit value in a general-purpose register, rc.
Note that this shift operation is *not* arithmetic, thus no sign-extension is performed.

FORMAT



OPERATION

```
if(i == 0)
     R[ra] = {{(shamt){0}}, R[rb]<31..shamt>};
else
     R[ra] = {{(R[rc]<4:0>){0}}, R[rb]<31..R[rc]<4:0>>};
```

EXAMPLE

LSR r1, r2, #4
LSR r1, r2, r3

SHL (Shift Left)

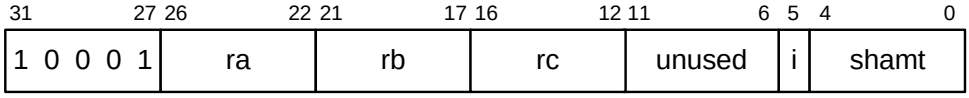
ASSEMBLY

SHL ra, rb, shamt
SHL ra, rb, rc

DESCRIPTION

The SHL shifts the content of a general-purpose register, rb, to the left by a given amount and store the result in a general-purpose register, ra. If the i bit is 0, the shift amount is an immediate value, specified in the shamt field, ranging from 0 to 31. Otherwise the shift amount is the least significant 5-bit value in a general-purpose register, rc.

FORMAT



OPERATION

```
if(I == 0)
    R[ra] = {R[rb]<31-shamt..0>, {(shamt){0}}};
else
    R[ra] = {R[rb]<31-R[rc]<4..0>..0>, {(R[rc]<4..0>){0}}};
```

EXAMPLE

SHL r1, r2, #4
SHL r1, r2, r3

ROR (Rotate Right)

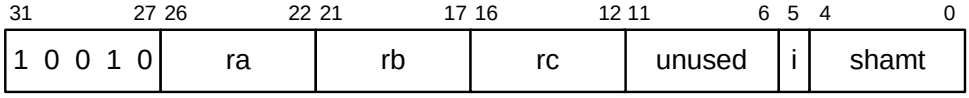
ASSEMBLY

ROR ra, rb, shamt
ROR ra, rb, rc

DESCRIPTION

The ROR rotates the content of a general-purpose register, rb, to the right by a given amount and store the result in a general-purpose register, ra. If the i bit is 0, the rotating amount is an immediate value, specified in the shamt field, ranging from 0 to 31. Otherwise the rotating amount is the least significant 5-bit value in a general-purpose register, rc.

FORMAT



OPERATION

```
if(I == 0)
     R[ra] = {R[rb]<shamt-1..0>, R[rb]<31..shamt>};
else
     R[ra] = {R[rb]<R[rc]<4..0>-1..0>,
               R[rb]<31..R[rc]<4..0>>};
```

EXAMPLE

ROR r1, r2, #4
ROR r1, r2, r3

BR (Branch)

ASSEMBLY

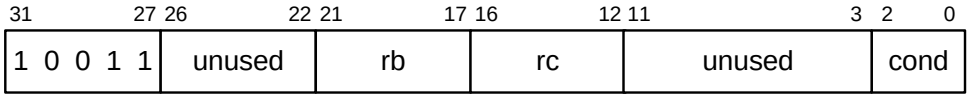
BR{cond} rb{, rc}

DESCRIPTION

The BR instruction changes the program flow based on the specified condition. The target address of the branch is the value in a general-purpose register rb and the value to be tested for condition evaluation is in a general-purpose register rc.

There are six conditions, {never, always, zero, nonzero, plus and minus}, each encoded as 3'b000, 3'b001, 3'b010, 3'b011, 3'b100 and 3'b101, respectively. These are specified in the 3-bit condition field of the instruction format. With the condition satisfied, one more instruction after BR is executed before actual branch occurs, i.e. the BR instruction has 1 delay slot. The default condition is 'always'.

FORMAT



OPERATION

```
if(cond == 0)       { /* Never */
      do_nothing;
}
else if(cond == 1) { /* Always */
      PC = R[rb];
}
else if(cond == 2) { /* Zero */
      if(R[rc] == 0)
            PC = R[rb];
}
else if(cond == 3) { /* Nonzero */
      if(R[rc] != 0)
            PC = R[rb];
}
else if(cond == 4) { /* Plus */
      if(R[rc] >= 0)
            PC = R[rb];
}
else if(cond == 5) { /* Minus */
      if(R[rc] < 0)
            PC = R[rb];
}
```

EXAMPLE

BR r1
BRNZ r4, r6

BRL (Branch and Link)

ASSEMBLY

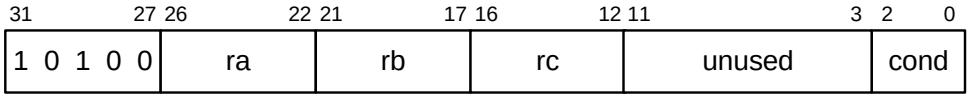
BRL{cond} ra, rb{, rc}

DESCRIPTION

The BRL instruction changes the program flow based on the specified condition while storing the current value of PC into a link register. The link address is mainly used to return from a subroutine or a procedure. The target address of the branch is the value in a general-purpose register rb, the value to be tested for condition evaluation is in a general-purpose register rc and the link register is ra.

Branch conditions are the same as those of BR. With the condition satisfied, one more instruction after BRL is executed before actual branch occurs, i.e. the BRL instruction has 1 delay slot. Note that the link register is updated regardless of whether the condition is satisfied or not.

FORMAT



OPERATION

```
R[ra] = currentPC;
if(cond == 0)       { /* Never */
      do_nothing;
}
else if(cond == 1) { /* Always */
      PC = R[rb];
}
else if(cond == 2) { /* Zero */
      if(R[rc] == 0)
            PC = R[rb];
}
else if(cond == 3) { /* Nonzero */
      if(R[rc] != 0)
            PC = R[rb];
}
else if(cond == 4) { /* Plus */
      if(R[rc] >= 0)
            PC = R[rb];
}
else if(cond == 5) { /* Minus */
      if(R[rc] < 0)
            PC = R[rb];
}
```

EXAMPLE

BRL r1, r2
BRLGE r4, r6, r11

J (Jump)

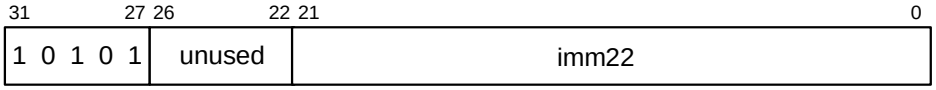
ASSEMBLY

J imm22

DESCRIPTION

The J instruction changes the program flow unconditionally. The 22 bit immediate (imm22) is sign-extended and the target address is generated by adding this extended value to the currentPC (currentPC = Address of J instruction + 4). One more instruction after J is executed before actual jump occurs, i.e. the J instruction has 1 delay slot.

FORMAT



OPERATION

```
PC = currentPC + signExt(imm22);
```

EXAMPLE

J #12

JL (Jump and Link)

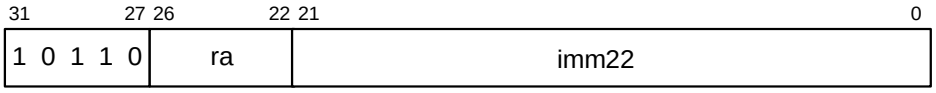
ASSEMBLY

JL ra, imm22

DESCRIPTION

The JL instruction changes the program flow unconditionally while storing the currentPC (currentPC = Address of J instruction + 4) into a link register(R[ra]). The link address is mainly used to return from a subroutine or a procedure. The 22 bit immediate (imm22) is sign-extended and the target address is generated by adding this extended value to the currentPC. One more instruction after JL is executed before actual jump occurs, i.e. the JL instruction has 1 delay slot.

FORMAT



OPERATION

```
R[ra] = currentPC;
PC = currentPC + signExt(imm22);
```

EXAMPLE

JL r1, #20

LD (Load)

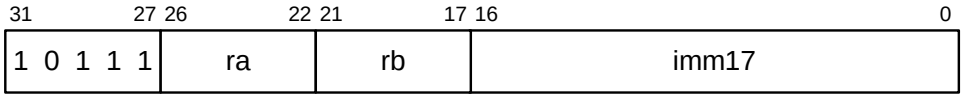
ASSEMBLY

LD ra, imm17
LD ra, imm17(rb)

DESCRIPTION

The LD instruction loads a word, or 4 bytes, of data from the given address of memory into a general-purpose register, ra. The address may be either an absolute address or a displacement address.
An absolute address given by a 17-bit immediate value (imm17) is sign extended. Thus, the memory area that the absolute address can handle ranges from 00000000h to 0000FFFFh and from FFFF0000h to FFFFFFFFh. When a displacement address is used, it is formed by adding a 17-bit immediate value and the value stored in the register, rb. The field rb being 31 indicates that the address is absolute.

FORMAT



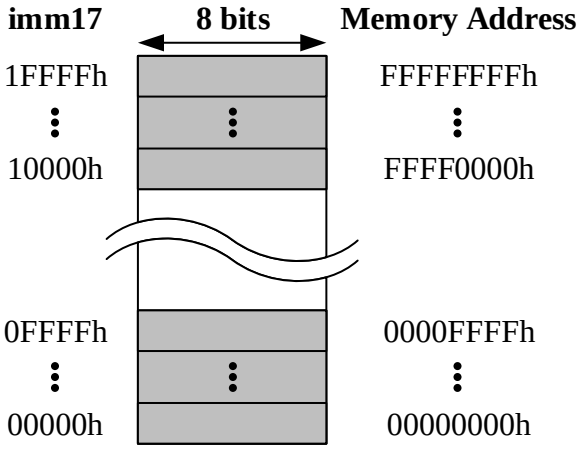
OPERATION

```
if(rb == 5'b11111)
     R[ra] = M[signExt(imm17)];
else
     R[ra] = M[signExt(imm17) + R[rb]];
```

EXAMPLE

LD r1, #0x1000
LD r3, #8(r17)

ADDRESS RANGE



LDR (Load PC Relative)

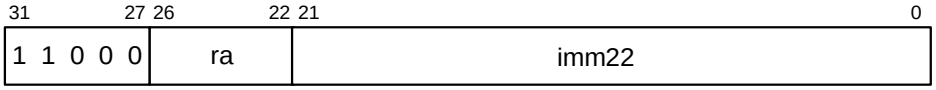
ASSEMBLY

LDR ra, imm22

DESCRIPTION

The LDR instruction loads a word, or 4 bytes, of data from the given relative address of memory into a general-purpose register, ra. The relative address is calculated by adding a 22-bit immediate value (imm22) to the current value of PC. Note that, since this addition is signed, imm22 is sign-extended to 32 bits before conducting the addition as an operand.

FORMAT



OPERATION

```
R[ra] = M[currentPC + signExt(imm22)];
```

EXAMPLE

LDR r1, #12

ST (Store)

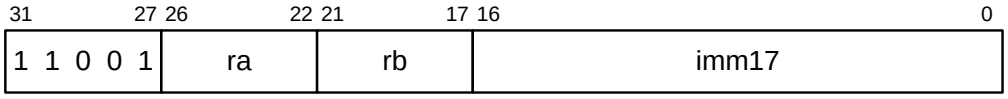
ASSEMBLY

ST ra, imm17
ST ra, imm17(rb)

DESCRIPTION

The ST instruction stores a word, or 4 bytes, of data from a general-purpose register, ra, into the given address of memory. The address may be either an absolute address or a displacement address.
An absolute address given by a 17-bit immediate value (imm17) is sign extended. Thus, the memory area that the absolute address can handle ranges from 00000000h to 0001FFFFh or from FFFE0000h to FFFFFFFFh. When a displacement address is used, it is formed by adding imm17 and the value stored in the register, rb. The field rb being 31 indicates that the address is absolute.

FORMAT



OPERATION

```
if(rb == 5'b11111)
     M[signExt(imm17)] = R[ra];
else
     M[R[rb] + signExt(imm17)] = R[ra];
```

EXAMPLE

ST r1, #4
ST r2, #4(r4)

STR (Store PC Relative)

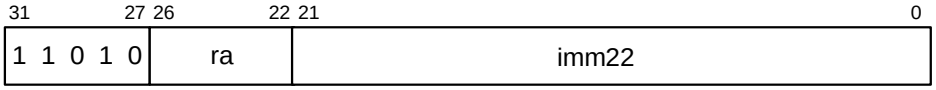
ASSEMBLY

STR ra, imm22

DESCRIPTION

The STR instruction stores a word, or 4 bytes, of data from a general-purpose register ra into the given relative address of memory. The relative address is calculated by adding a 22-bit immediate value (imm22) to the current value of PC. Note that, since this addition is signed, imm22 is sign-extended to 32 bits before conducting the addition as an operand.

FORMAT



OPERATION

```
M[currentPC + signExt(imm22)] = R[ra];
```

EXAMPLE

STR r1, #40

LEA (Load Effective Address)

ASSEMBLY

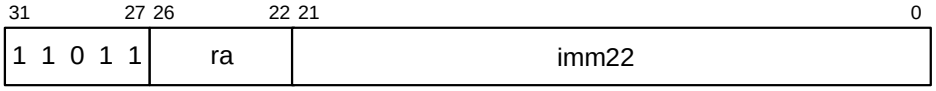
LEA ra, imm22

DESCRIPTION

The LEA instruction loads a general-purpose register ra with a relative address, which is calculated by adding a 22-bit immediate value imm22 to the current value of PC. Since imm22 is shorter than PC and the addition is signed, it is sign-extended.

This instruction is mostly used for setting a branch target address prior to executing a branch instruction. Note that, in this case, the branch range is limited by 00400000h.

FORMAT



OPERATION

```
R[ra] = currentPC + signExt(imm22);
```

EXAMPLE

LEA r1, #0x20

IEN (Interrupt Enable)

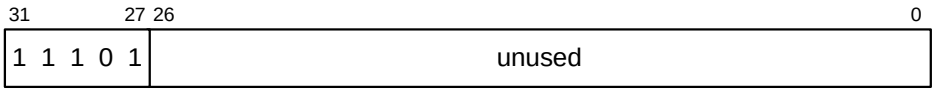
ASSEMBLY

IEN

DESCRIPTION

The IEN instruction enables the recognition of exceptions and sets the interrupt enable flag, IE, to 1.

FORMAT



OPERATION

```
IE = 1;
```

IDS (Interrupt Disable)

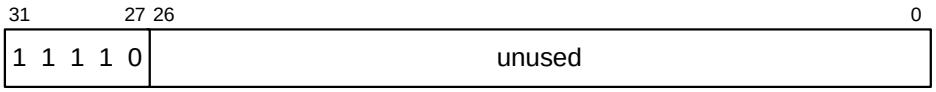
ASSEMBLY

IDS

DESCRIPTION

The IDS instruction disables the recognition of exceptions and clears the interrupt enable flag, IE, to 0.

FORMAT



OPERATION

```
IE = 0;
```

RFI (Return from Interrupt)

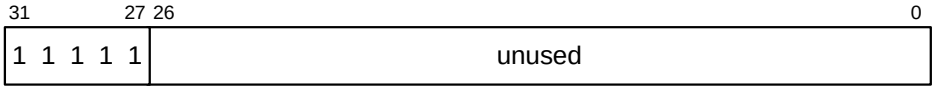
ASSEMBLY

RFI

DESCRIPTION

The RFI instruction returns control to the interrupted process from the interrupt service routine by copying IPC to PC and setting the interrupt enable flag, IE, to 1.

FORMAT



OPERATION

```
PC = IPC;
IE = 1;
```

Opcode Summary

Instruction	opcode	Instruction	opcode
ADDI (type1)	0	LSR	16
ADDI (type2)	1	SHL	17
ORI (type1)	2	ROR	18
ORI (type2)	3	BR	19
ANDI (type1)	4	BRL	20
ANDI (type2)	5	J	21
MOVI (type1)	6	JL	22
MOVI (type2)	7	LD	23
ADD	8	LDR	24
SUB	9	ST	25
NOT	10	STR	26
NEG	11	LEA	27
OR	12	-	-
AND	13	IEN	29
XOR	14	IDS	30
ASR	15	RFI	31

5. Pipeline

KRP has 5 pipeline stages which are:

- Instruction Fetch
- Decode and operand read
- ALU operation
- Memory access
- Register write

As the branch target is known in the decode stage, the instruction located after J, JL, BR, or BRL is always executed before actual branch occurs, even though the branch condition is true, i.e. the instruction has 1 delay slot.