mophun^{TM} Assembly Reference

Contents

Chapter 1

Introduction

This document describes the PIP2[1] architecture used by the mophun Software Platform. It describes the architecture as well as the instruction set and assembler syntax.

1.1 Architecture

The PIP2 architecture is a software based system for executing programs in a platform independent fashion within a sandbox environment[2]. It is primarly 32-bit in design, although a certain number of 8- and 16-bit operations exist to make it efficient even in very limited environments.

The memory adress space is 32 bits in length although software may only access memory within it's private address space. Memory accesses on quantities larger than 8 bits are strictly aligned to ensure portability.

1.2 Assembler syntax

1.2.1 Registers

The PIP2 architecture has 32 general purpose registers. Some registers by convention have special meanings, such as the stack pointer.

Register	**Alias**	**Description**
$0	$zero	Defined as zero.
$1	$sp	Stack pointer.
$2	$ra	Subroutine return address.
$3	$fp	Frame pointer.
$4-$11	$s0-$s7	Saved registers, must be preserved across function calls.
$12-$15	$p0-$p3	Subroutine parameter registers.

[1] Platform Independent Processor

[2] The software executes within a protected space

\$16-29	\$g0-\$g13	Temporary general purpose registers. Not preserved across fuction calls.
\$30-\$31	\$r0-\$r1	Contains function return values. Not preserved across function calls.

1.2.2 Instruction syntax

The general instruction syntax is

```
[LABEL:] INSN OPERAND0, OPERAND1...
```

For more information on the syntax of the assembler refer to the GNU assembler manual.

1.3 ABI Conventions

This section describes the conventions used by mophun applications running on the PIP2 VM, such as calling conventions and register usage.

1.3.1 Subroutine Arguments

The first four arguments are passed in the *\$p0-\$p3* registers, remaining arguments as well as arguments to stdarg (ellipsis) functions are passed on the stack.

1.3.2 Function return values

Function values are returned in the *\$r0* register, if a function returns a multi-word value the second word is returned in *\$r1*.

1.3.3 Register usage

- The *\$sp* register is used as the stack pointer and points to the current lowest address used on the stack, its value must be preserved across subroutine calls.
- The *\$fp* register is similar to *\$sp*, but contains the stack frame-pointer in subroutines that need one. The value of *\$fp* must be preserved across subroutine calls but may be used for any purpose.
- The *\$ra* register is used to hold the address to which a subroutine should return. It's value is implicitly set by a *call* instruction. If a subroutine calls another subroutine it must save the value of *\$ra,* preferably by executing a *store* instruction at the start of the subroutine and executing a matching *restore* instruction upon exit.
- The registers *\$s0-\$s7* are referred to as saved registers, their values are guaranteed to be preserved across subroutine calls. It is the responsibility of the called subroutine to save the necessary registers on the stack.

- The four argument registers $p0-$p3 are used to pass arguments to subroutines. The values are not guaranteed to be preserved across subroutine calls.
- The *$r0* and *$r1* registers hold function return values. The contents may be destroyed by a function call.
- Temporary values of expression evaluation may be computed and store in the registers *$g0-$g15*, the values are however not guaranteed to be preserved across subroutine calls.

1.3.4 Stack structure

The stack is pointed to by the *$sp* register. As data is accumulated on the stack the value of *$sp* decreases, i.e the stack is growing downwards. At all times the value of the stack pointer must be 32 bit aligned, or else an alignment error might occur. It is the responsibility of each subroutine to always adjust the stack in 4 byte quantities.

If a subroutine call other subroutines and/or keep local data on the stack it may need a stack-frame pointer. The stack-frame should be initialized upon subroutine entry. Local data are stored at negative offsets from the stack-frame pointer.

Here's an example of subroutine prologue and epilogue assembly code to manage a 16 byte long stack frame:

```
store $ra,$fp   # Preserve $ra and $fp
sub $sp,$sp,16  # allocate a 16 byte stack frame
add $fp,$sp,24  # point $fp to argu-
ments ($sp+16+$ra+$fp) one
                # word above the last stack slot ...
call foo        # invoke subroutine ...
stb $0, $fp,-20 # store 0 in locals[4]
add $sp,$sp,16  # Restore the stack pointer
ret $ra,$fp     # Restore $ra and $fp and re-
turn to caller
```

Chapter 2

Instruction Set

This chapter documents the PIP2 instruction set.

2.1 Abbreviations and conventions

This section describes how to read the instruction set reference and also explains the abbreviations used in the following sections.

List of abbreviations

rd	Destination register operand.
rd8	8 bit destination register.
rd16	16 bit destination register.
rs	Source register.
rs8	8 bit source register.
rs16	16 bit source register.
rt	Second register operand.
rt8	8 bit secondary register.
rt16	16 bit secondary register.
imm32	32 bit immediate operand. This can be a numeric constant or a symbol or label reference.
imm16	16 bit immediate operand.
immq	8 bit immediate operand.
LABEL	An assembler label.

2.2 Instruction set

2.2.1 ADD - Add

Syntax: `add` *`rd,rs,rt/imm32`*
Operation: `rd <- rs + rt`
Operation: `rd <- rs + imm32`

The *ADD* instruction computes the sum of the operands and stores the result in *rd*.

2.2.2 ADDB - Add byte

Syntax: `addb` *`rd8,rs8,rt8/immq`*
Operation: `rd8 <- rs8 + rt8`
Operation: `rd8 <- rs8 + immq`

The *ADDB* instruction computes the 8-bit sum of the source operands and stores the result in *rd8*.

2.2.3 ADDH - Add half-word

Syntax: `addh` *`rd16,rs16,rt16/immq`*
Operation: `rd16 <- rs16 + rt16`
Operation: `rd16 <- rs16 + sign_extend(immq)`

The *ADDH* instruction computes the 16-bit sum of the source operands and stores the result in *rd16*.

2.2.4 ADDQ - Add sign-extended byte

Syntax: `addq` *`rd,rs,immq`*
Operation: `rd <- rs + sign_extend(immq)`

The *ADDQ* instruction computes the sum of the source register and the sign-extended 8-bit immediate and stores the result in *rd*.

2.2.5 AND - Bitwise and

Syntax: `and` *`rd,rs,rt/imm32`*
Operation: `rd <- rs & rt`
Operation: `rd <- rs & imm32`

The *AND* instruction computes the bitwise and value of the input operands and stores the result in *rd*.

2.2.6 ANDB - Bitewise and byte

Syntax: `andb` *`rd8,rs8,rt8/immq`*
Operation: `rd8 <- rs8 & rt8/immq`

The *ANDB* instruction performs the bitwise and operation on the 8-bit input operands and stores the result in *rd8*.

2.2.7 ANDH - Bitewise and half-word

Syntax: `andb` *`rd16,rs16,rt16/immq`*
Operation: `rd8 <- rs16 & rt16`
Operation: `rd8 <- rs16 & zero_extend(immq)`

The *ANDH* instruction performs the bitwise and operation on the 16-bit input operands and stores the result in *rd*16.

2.2.8 BEQ - Branch Equal

Syntax: `beq` *`rd,rs,LABEL`*
Operation: `if (rd == rs) goto LABEL`

Syntax: `beqi` *`rd,immq,LABEL`*
Operation: `if (rd == sign_extend(immq)) goto LA-BEL`

Syntax: `beqib` *`rd8,immq,LABEL`*
Operation: `if (rd8 == immq) goto LABEL`

Branch to *LABEL* if the value of *rd* is equal to the second operand.

2.2.9 BGE - Branch Greater or Equal

Syntax: `bge` *`rd,rs,LABEL`*
Operation: `if (rd >= rs) goto LABEL`

Syntax: `bgei` *`rd,immq,LABEL`*
Operation: `if (rd >= sign_extend(immq)) goto LA-BEL`

Syntax: `bgeib` *`rd8,immq,LABEL`*
Operation: `if (rd8 >= immq) goto LABEL`

Branch to *LABEL* if the value of *rd* is greater or equal to second operand.

2.2.10 BGEU - Branch Unsigned Greater or Equal

Syntax: `bgeu` *rd,rs,LABEL*
Operation: `if (rd >= rs) goto LABEL`

Syntax: `bgeui` *rd,immq,LABEL*
Operation: `if (rd >= zero_extend(immq)) goto LABEL`

Syntax: `bgeuib` *rd8,immq,LABEL*
Operation: `if (rd >= immq) goto LABEL`

Branch to *LABEL* if the unsigned difference between value of *rd* and second operand is greater than or equal to zero.

2.2.11 BGT - Branch Greater Than

Syntax: `bgt` *rd,rs,LABEL*
Operation: `if (rd > rs) goto LABEL`

Syntax: `bgti` *rd,immq,LABEL*
Operation: `if (rd > sign_extend(immq)) goto LABEL`

Syntax: `bgtib` *rd8,immq,LABEL*
Operation: `if (rd8 > immq) goto LABEL`

Branch to *LABEL* if the value of *rd* is greater than the second operand.

2.2.12 BGTU - Branch Unsigned Greater Than

Syntax: `bgtu` *rd,rs,LABEL*
Operation: `if (rd > rs) goto LABEL`

Syntax: `bgtui` *rd,immq,LABEL*
Operation: `if (rd > zero_extend(immq)) goto LABEL`

Syntax: `bgtuib` *rd8,immq,LABEL*
Operation: `if (rd > immq) goto LABEL`

Branch to *LABEL* if the unsigned difference between value of *rd* and the second operand is greater than zero.

2.2.13 BLE - Branch Less or Equal

Syntax: `ble` *rd,rs,LABEL*
Operation: `if (rd <= rs) goto LABEL`

Syntax: `blei` *rd,immq,LABEL*
Operation: `if (rd <= sign_extend(immq)) goto LABEL`

Syntax: `bleib` *rd8,immq,LABEL*
Operation: `if (rd8 <= immq) goto LABEL`

Branch to *LABEL* if the value of *rd* is less than or equal to the second operand.

2.2.14 BLEU - Branch Unsigned Less or Equal

Syntax: `bleu` *rd,rs,LABEL*
Operation: `if (rd <= rs) goto LABEL`

Syntax: `bleui` *rd,immq,LABEL*
Operation: `if (rd <= zero_extend(immq)) goto LABEL`

Syntax: `bleuib` *rd8,immq,LABEL*
Operation: `if (rd8 <= immq) goto LABEL`

Branch to *LABEL* if the value of *rd* is less than or equal to the second operand.

2.2.15 BLT - Branch Less Than

Syntax: `blt` *rd,rs,LABEL*
Operation: `if (rd < rs) goto LABEL`

Syntax: `blti` *rd,immq,LABEL*
Operation: `if (rd < sign_extend(immq)) goto LABEL`

Syntax: `blti` *rd8,immq,LABEL*
Operation: `if (rd8 < immq) goto LABEL`

Branch to *LABEL* if the value of *rd* is less than the second operand.

2.2.16 BLTU - Branch Unsigned Less Than

Syntax: `bltu` *rd,rs,LABEL*
Operation: `if (rd < rs) goto LABEL`

Syntax: `bltui` *rd,immq,LABEL*
Operation: `if (rd < zero_extend(immq)) goto LABEL`

Syntax: `bltuib` *rd8,immq,LABEL*
Operation: `if (rd8 < immq) goto LABEL`

Branch to *LABEL* if the the unsigned value of *rd* is less than the second operand.

2.2.17 BNE - Branch Not Equal short

Syntax: `bne` *`rd,rs,LABEL`*
Operation: `if (rd != rs) goto LABEL`

Syntax: `bnei` *`rd,immq,LABEL`*
Operation: `if (rd != sign_extend(immq)) goto LA-BEL`

Syntax: `bneib` *`rd8,immq,LABEL`*
Operation: `if (rd8 != immq) goto LABEL`

Branch to *LABEL* if the value of *rd* is not equal to the second operand.

2.2.18 CALL - Call subroutine

Syntax: `call` *`rd/LABEL`*
Operation: `$ra <- address_of_next_instruction; goto rd/LABEL`

The *CALL* instruction stores the address of the following instruction in the *$ra* register and then jumps to the specified address.

2.2.19 DIV - Signed divide

Syntax: `div` *`rd,rs,rt/imm32`*
Operation: `rd <- rs` **`div`** `rt`
Operation: `rd <- rs` **`div`** `imm32`

The *DIV* instruction performs a signed 32-bit division of the source operands and stores the quotient in *rd*. Division by zero is undefined.

2.2.20 DIVU - Unsigned divide

Syntax: `divu` *`rd,rs,rt/imm32`*
Operation: `rd <- rs` **`div`** `rt`
Operation: `rd <- rs` **`div`** `imm32`

The *DIVU* instruction performs an unsigned 32-bit division of the source operands and stores the quotient in *rd*. Division by zero is undefined.

2.2.21 EXSB - Sign-extend byte

Syntax: `exsb` *`rd,rs8`*
Operation: `rd <- sign_extend(rs8)`

The *EXSB* instruction sign extends the 8 least significant bits in the source register (*rs8*) and stores the 32-bit result in *rd*.

2.2.22 EXSH - Sign-extend half-word

Syntax: `exsh` *rd,rs16*
Operation: `rd <- sign_extend(rs16)`

The *EXSH* instruction sign extends the 16 least significant bits in the source register (*rs16*) and stores the 32-bit result in *rd*.

2.2.23 JP - Unconditional jump

Syntax: `jp` *rd/LABEL*
Operation: `goto rd/LABEL`

The *JP* instruction unconditionally jumps to the specified label or an address stored in *rd*.

2.2.24 KILLTASK - Terminate task

Syntax: `killtask`

Terminate the currently running task and schedule the next task.

2.2.25 LDB - Load byte

Syntax: `ldb` *rd,LABEL*
Operation: `rd < sign_extend([LABEL])`
Syntax: `ldb` *rd,rs,imm32*
Operation: `rd <- sign_extend([rs + imm32])`

Load the 8-bit value stored at the memory address specified by the sum of the base register *rs* and the 32-bit displacement. If no base register is specified *LABEL* specifies the memory address.

2.2.26 LDBU - Load unsigned byte

Syntax: `ldb` *rd,LABEL*
Operation: `rd < zero_extend([LABEL])`
Syntax: `ldb` *rd,rs,imm32*
Operation: `rd <- zero_extend([rs + imm32])`

Load the 8-bit value stored at the memory address specified by the sum of the base register *rs* and the 32-bit displacement. If no base register is specified *LABEL* specifies the memory address.

2.2.27 LDH - Load half-word

Syntax: `ldh` *rd,LABEL*
Operation: `rd < sign_extend([LABEL])`
Syntax: `ldh` *rd,rs,imm32*
Operation: `rd <- sign_extend([rs + imm32])`

Load the 16-bit value stored at the memory address specified by the sum of the base register *rs* and the 32-bit displacement. If no base register is specified *LABEL* specifies the memory address.

2.2.28 LDHU - Load unsigned half-word

Syntax: `ldhu` *rd,LABEL*
Operation: `rd < zero_extend([LABEL])`
Syntax: `ldh` *rd,rs,imm32*
Operation: `rd <- zero_extend([rs + imm32])`

Load the 16-bit value stored at the memory address specified by the sum of the base register *rs* and the 32-bit displacement. If no base register is specified *LABEL* specifies the memory address.

2.2.29 LDI - Load immediate

Syntax: `ldi` *rd,imm32/LABEL*
Operation: `rd <- imm32/LABEL`

The *LDI* instruction loads a 32 bit value into a register. The value can be either a numeric constant or the address of a label.

2.2.30 LDW - Load word

Syntax: `ldw` *rd,LABEL*
Operation: `rd < [LABEL]`
Syntax: `ldw` *rd,rs,imm32*
Operation: `rd <- [rs + imm32]`

Load the 32-bit value stored at the memory address specified by the sum of the base register *rs* and the 32-bit displacement. If no base register is specified *LABEL* specifies the memory address.

2.2.31 LDQ - Load 16-bit immediate

Syntax: `ldq` *rd,imm16*
Operation: `rd <- sign_extend(imm16)`

Load the sign-extended 16-bit immediate into the destination register.

2.2.32 MOVB - Move byte register

Syntax: `movb` *rd8,rs8*
Syntax: `moveb` *rd8,rs8*
Operation: `rd8 <- rs8`

The *MOVB* instruction moves the 8-bit contents of the source register to the destination register.

2.2.33 MOVH - Move half-word register

Syntax: `movh` *rd16,rs16*
Syntax: `moveh` *rd16,rs16*
Operation: `rd16 <- rs16`

The *MOVH* instruction moves the 16-bit contents of the source register to the destination register.

2.2.34 MUL - Multiply

Syntax: `mul` *rd,rs,rt/imm32*
Operation: `rd <- rs * rt`
Operation: `rd <- rs * imm32`

The *MUL* instruction computes the product the source operands and stores the 32 bit result in *rd*. Overflow is ignored.

2.2.35 MULQ - Multiply zero-extended byte

Syntax: `mulq` *rd,rs,immq*
Operation: `rd <- rs * zero_extend(immq)`

The *MULQ* instruction computes the product of the source register and the zero-extended immediate and stores the result in *rd*. Overflow is ignored.

2.2.36 NEG - Negate

Syntax: `neg` *rd,rs*
Operation: `rd = -rs`

The *NEG* instructions negates the signed value of the source operand (*rs*) and stores the result in *rd*.

2.2.37 NOT - Bitwise Complement

Syntax: `not` *rd,rs*
Operation: `rd <- ~rs`

The *NOT* instruction computs the bitwise complement of the source operand (*rs*) and stores the result in *rd*.

2.2.38 OR - Bitwise or

```
Syntax: or rd,rs,rt/imm32
Operation: rd <- rs | rt
Operation: rd <- rs | imm32
```

The *OR* instruction computes the result of the 32-bit bitwise or operation on the source operands and stores the result in *rd*.

2.2.39 ORB - Bitwise or byte

```
Syntax: orb rd8,rs8,rt8/immq
Operation: rd8 <- rs8 | rt8
Operation: rd8 <- rs8 | immq
```

The *ORB* instruction performs the bitwise or operation on the 8-bit source operands and stores the result in *rd8*.

2.2.40 ORH - Bitwise or half-word

```
Syntax: orh rd16,rs16,rt16
Operation: rd16 <- rs16 | rt16
```

The *ORH* instruction performs the bitwise or operation on the 16-bit source operands and stores the result in *rd*16.

2.2.41 RESTORE - Pop registers from stack

```
Syntax: restore rd,rs
Operation: for (r = rs; r >= rd; r--) $r = $sp++;
```

Pop the range of registers specified by *rd* and *rs* (inclusive) from the stack order.

2.2.42 SLEEP - Yield task

```
Syntax: sleep
```

Yield from the current task and schedule the next task.

2.2.43 SLL - Shift Left Logical

```
Syntax: sll rd,rs,rt/immq
Operation: rd <- rs << rt
Operation: rd <- rs << immq
```

The *SLL* instruction shifts the bits in *rs* left the number of times specified by the second operand (*rt* or *immq*) and inserts zeros in the least significant bits. If the shift amount is greater than 31 the result is undefined. The result is stored in *rd*.

2.2.44 SLLB - Shift Left Logical byte

> **Syntax:** `sllb` *`rd8,rs8,immq`*
> **Operation:** `rd8 <- rs8 << immq`

The *SLLB* instruction shifts the bits in *rs8* left the number of times specified by the immediate operand and inserts zeros in the least significant bits. The result is stored in *rd*8.

2.2.45 SLLH - Shift Left Logical half-word

> **Syntax:** `sllh` *`rd16,rs16,immq`*
> **Operation:** `rd16 <- rs16 << immq`

The *SLLH* instruction shifts the bits in *rs16* left the number of times specified by the immediate operand and inserts zeros in the least significant bits. The result is stored in *rd*16.

2.2.46 SRA - Shift Right Arithmetic

> **Syntax:** `sra` *`rd,rs,rt/immq`*
> **Operation:** `rd <- rs_signed >> rt`
> **Operation:** `rd <- rs_signed >> immq`

The *SRA* instruction shift the bits in *rs* to the right the number of times specified by the second operand (*rt* or *immq*). The sign is set or cleared according to the original value of the sign. If the shift amount is greater than 31 the result is undefined. The result is stored in *rd*.

2.2.47 SRAB - Shift Right Arithmetic byte

> **Syntax:** `srab` *`rd8,rs8,immq`*
> **Operation:** `rd8 <- rs8_signed >> immq`

The *SRAB* instruction shift the bits in *rs8* to the right the number of times specified by the immediate operand. The sign is set or cleared according to the original value of the sign. The result is stored in *rd*8.

2.2.48 SRAH - Shift Right Arithmetic half-word

> **Syntax:** `srah` *`rd16,rs16,immq`*
> **Operation:** `rd16 <- rs16_signed >> immq`

The *SRAH* instruction shift the bits in *rs16* to the right the number of times specified by the immediate operand. The sign is set or cleared according to the original value of the sign. The result is stored in *rd*16.

2.2.49 SRL - Shift Right Logical

Syntax: `srl` *rd,rs,rt/immq*
Operation: `rd <- rs >> rt`
Operation: `rd <- rs >> immq`

The *SRL* instruction shifts the bits in *rs* to the right the number of times specified by the second operand (*rt* or *immq*) and inserts zeros in the most significant bits. If the shift amount is greater than 31 the result is undefined. The result is stored in *rd*.

2.2.50 SRLB - Shift Right Logical byte

Syntax: `srlb` *rd8,rs8,immq*
Operation: `rd8 <- rs8 >> immq`

The *SRLB* instruction shifts the bits in *rs8* to the right the number of times specified by the immediate operand and inserts zeros in the most significant bits. The result is stored in *rd*8.

2.2.51 SRLH - Shift Right Logical half-word

Syntax: `srlh` *rd16,rs16,immq*
Operation: `rd16 <- rs16 >> immq`

The *SRLH* instruction shifts the bits in *rs16* to the right the number of times specified by the immediate operand and inserts zeros in the most significant bits. The result is stored in *rd*16.

2.2.52 STB - Store byte

Syntax: `stb` *rd8,LABEL*
Operation: `[LABEL] <- rd8`
Syntax: `stb` *rd8,rs,imm32*
Operation: `[rs + imm32] <- rd`

Store the value of *rd8* to the memory address specified by the sum of the base register *rs* and the 32-bit displacement. If no base register is specified *LABEL* specifies the memory address.

2.2.53 STH - Store half-word

Syntax: `stb` *rd16,LABEL*
Operation: `[LABEL] <- rd16`
Syntax: `stb` *rd16,rs,imm32*
Operation: `[rs + imm32] <- rd`

Store the value of *rd16* to the memory address specified by the sum of the base register *rs* and the 32-bit displacement. If no base register is specified *LABEL* specifies the memory address.

2.2.54 STW - Store word

> **Syntax:** `stb` *`rd,LABEL`*
> **Operation:** `[LABEL] <- rd`
> **Syntax:** `stb` *`rd,rs,imm32`*
> **Operation:** `[rs + imm32] <- rd`

Store the value of *rd* to the memory address specified by the sum of the base register *rs* and the 32-bit displacement. If no base register is specified *LABEL* specifies the memory address.

2.2.55 STORE - Store registers on stack

> **Syntax:** `store rd,rs`
> **Operation:** `for (r = rd; r <= rs; r++) $--sp = $r;`

Push the range of registers specified by *rd* and *rs* (inclusive) onto the stack.

2.2.56 SUB - Subtract

> **Syntax:** `sub` *`rd,rs,rt`*
> **Operation:** `rd <- rs - rt`

The *SUB* instruction subtracts *rt* from *rs* and stores the result in *rd.*

2.2.57 SUBB - Subtract byte

> **Syntax:** `subb` *`rd8,rs8,rt8`*
> **Operation:** `rd8 <- rs8 - rt8`

The *SUBB* instruction subtracts the 8-bit value in *rt8* from the 8-bit value in *rs8* and stores the result in *rd8*.

2.2.58 SUBH - Subtract half-word

> **Syntax:** `subh` *`rd16,rs16,rt16`*
> **Operation:** `rd16 <- rs16 - rt16`

The *SUBH* instruction subtracts the16-bit value in *rt16* from the 16-bit value in *rs16* and stores the result in *rd*16.

2.2.59 SYSCPY - memory block copy

> **Syntax:** `syscpy` *`rd,rs,rt`*

Perform a memory block copy from the address specified by *rs* to the address specified by *rd*. The number of bytes to copy is specified by *rt*.

2.2.60 SYSSET - memory block fill

Syntax: `sysset` *`rd,rs8,rt`*

Fill the memory block pointed to by *rd* with the value specified in *rs8*. The number of bytes to set is specified by *rt*.

2.2.61 XOR - Bitwise exclusive or

Syntax: `xor` *`rd,rs,rt/imm32`*
Operation: `rd <- rs ^ rt`
Operation: `rd <- rs ^ imm32`

The *XOR* instruction computes the result of the 32-bit exclusive or operation on the source operands and stores the result in *rd*.