

## CPYPTN, CPYMTN, CPYETN

Memory Copy, reads and writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTN, then CPYMTN, and then CPYETN.

CPYPTN performs some preconditioning of the arguments suitable for using the CPYMTN instruction, and performs an implementation defined amount of the memory copy. CPYMTN performs an implementation defined amount of the memory copy. CPYETN performs the last part of the memory copy.

---

### Note

The inclusion of implementation defined amounts of memory copy allows some optimization of the size that can be performed.

---

For CPYPTN, the following saturation logic is applied:

If  $X_n < 63:55 > \neq 000000000$ , the copy size  $X_n$  is saturated to  $0x007FFFFFFFFFFFFFFF$ .

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If  $(X_s > X_d) \ \&\& \ (X_d + \text{saturated } X_n) > X_s$ , then direction = forward

Elsif  $(X_s < X_d) \ \&\& \ (X_s + \text{saturated } X_n) > X_d$ , then direction = backward

Else direction = implementation defined choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is implementation defined.

---

### Note

Portable software should not assume that the choice of algorithm is constant.

---

After execution of CPYPTN, option A (which results in encoding  $PSTATE.C = 0$ ):

- $PSTATE.\{N,Z,V\}$  are set to  $\{0,0,0\}$ .
- If the copy is in the forward direction, then:
  - $X_s$  holds the original  $X_s + \text{saturated } X_n$ .
  - $X_d$  holds the original  $X_d + \text{saturated } X_n$ .
  - $X_n$  holds  $-1 * \text{saturated } X_n + \text{an implementation defined number of bytes copied}$ .
- If the copy is in the backward direction, then:
  - $X_s$  and  $X_d$  are unchanged.
  - $X_n$  holds the saturated value of  $X_n - \text{an implementation defined number of bytes copied}$ .

After execution of CPYPTN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an implementation defined number of bytes copied.
  - Xd holds the original Xd + an implementation defined number of bytes copied.
  - Xn holds the saturated Xn - an implementation defined number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an implementation defined number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an implementation defined number of bytes copied.
  - Xn holds the saturated Xn - an implementation defined number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1\* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from - Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1\* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from - Xn+1.
  - Xd holds the highest address that the copy is copied to - Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction ( $PSTATE.N == 0$ ), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction ( $PSTATE.N == 1$ ), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYETN, option A (encoded by  $PSTATE.C = 0$ ), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds  $-1 \times$  the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from - Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from - Xn+1.
  - Xd holds the highest address that the copy is copied to - Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

#### Integer (FEAT\_MOPS)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sz		0	1	1	1	0	1	op1		0	Rs				1	1	1	1	0	1	Rn				Rd						
																op2															

#### Epilogue (op1 == 10)

CPYETN [[<Xd>](#)]!, [[<Xs>](#)]!, [<Xn>](#)!

#### Main (op1 == 01)

CPYMTN [[<Xd>](#)]!, [[<Xs>](#)]!, [<Xn>](#)!

#### Prologue (op1 == 00)

CPYPTN [[<Xd>](#)]!, [[<Xs>](#)]!, [<Xn>](#)!

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';
```

```

MOPSSStage stage;
case op1 of
  when '00' stage = MOPSSStage_Prologue;
  when '01' stage = MOPSSStage_Main;
  when '10' stage = MOPSSStage_Epilogue;
  otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
  assert c IN {Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

```

For information about the constrained unpredictable behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *Memory Copy and Memory Set CPY\**.

## Assembler Symbols

- <Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
- <Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
- <Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.
- For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.
- For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

## Operation

```

constant integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(8*N) readdata;

boolean implements_option_a = CPYOptionA();
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessP
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessP

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp\_LOAD, rprivileged,
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp\_STORE, wprivileged,

if stage == MOPSTage\_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFFFFF<

    boolean forward = IsMemCpyForward(toaddress, fromaddress, cpysize);

    if implements_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
        else
            if !forward then
                // Copy in the reverse direction offsets the arguments.
                toaddress = toaddress + cpysize;
                fromaddress = fromaddress + cpysize;
                nzcv = '1010';
            else
                nzcv = '0010';
        else
            CheckMemCpyParams(stage, implements_option_a, nzcv, options, d, s,

bits(64) stagecpysize = MemCpyStageSize(stage, toaddress, fromaddress,

if implements_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While
        // implementations might make this constant, that is not assume
        constant integer B = CPYSizeChoice(toaddress, fromaddress, cpy

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);
            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, raccdesc];
            Mem[toaddress+cpysize, B, waccdesc] = readdata<B*8-1:0>;

```

```

        if stage != MOPSStage\_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While
        // implementations might make this constant, that is not assume
        constant integer B = CPYSizeChoice(toaddress, fromaddress, cpys
        assert B <= UInt(stagecpysize);

        if nzcvc<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, raccdesc];
            Mem[toaddress, B, waccdesc] = readdata<B*8-1:0>;

            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, raccdesc];
            Mem[toaddress-B, B, waccdesc] = readdata<B*8-1:0>;

            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage\_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage\_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcvc;

```

[Base  
Instructions](#)

[SIMD&FP  
Instructions](#)

[SVE  
Instructions](#)

[SME  
Instructions](#)

[Index by  
Encoding](#)

[Sh  
Pseud](#)

Internal version only: isa v33.64, AdvSIMD v29.12, pseudocode  
no\_diffs\_2023\_09\_RC2, sve v2023-06\_rel ; Build timestamp: 2023-09-18T17:56

Copyright Â© 2010-2023 Arm Limited or its affiliates. All rights reserved. This  
document is Non-Confidential.