Sol Boucher and Evan Klei CSCI-453-01 04/28/14

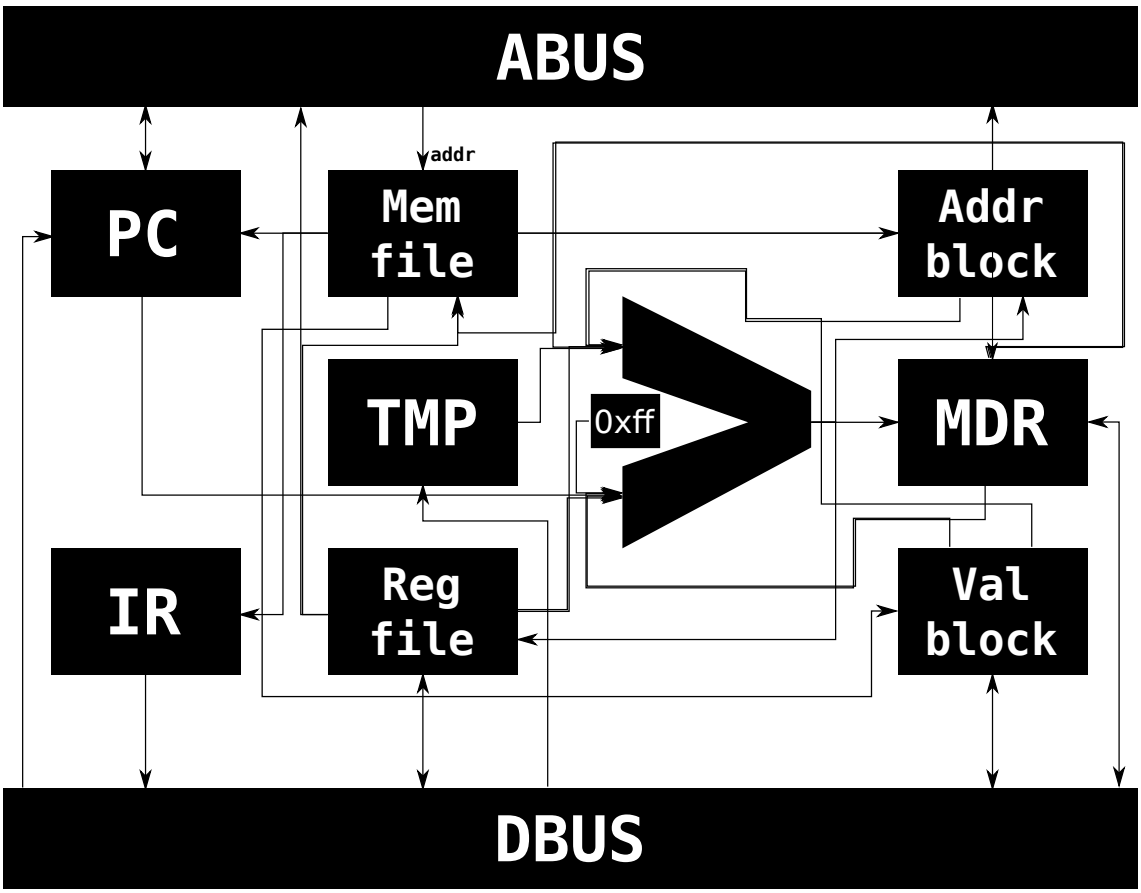
FabComp: Hardware specification

1 Hardware

The computer is composed of a largely isolated data unit and control unit, which are only connected by a couple of direct buses.

1.1 Data path

All components of the data path have both 16-bit word size and address length. They are connected as such:



The system's storage components adhere to these specifications:

data path storage objects		
name	type	purpose
PC	counter register	program counter
IR	shift register[†]	instruction register
MDR	shift register	main data register
TMP	register	temporary register
Mem	memory	main memory (program and data)
Reg	16-register bank	general-purpose registers (*see* ISA document, section 5)
Val	3-register bank	storage of immediate values
Addr	3-register bank	storage of effective addresses

1.2 Control path

The word size and address widths within the control path are component-specific:

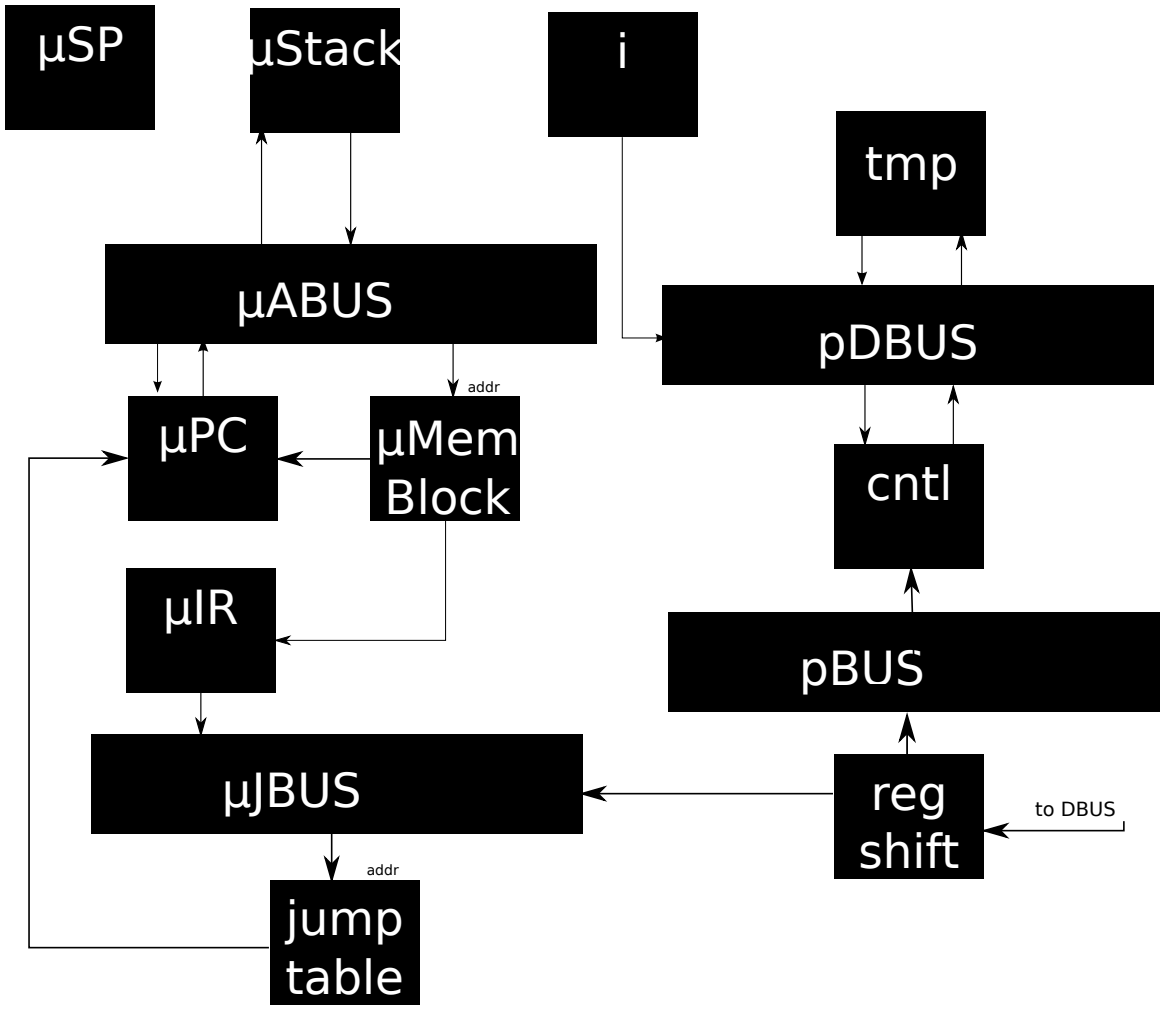
control path storage objects			
name	type	word size	purpose
μPC	counter register	12-bit	microprogram counter
μIR	register	16-bit	microinstruction register
cntl	3–register bank[†]	5-bit	operand control flags
tmp	register	5-bit	temporary control flags
i	counter register	2-bit	track current working operand
regshift	shift register	16-bit	transfer register IDs from data path
μSP	counter register	2-bit	micro–stack pointer
μStack	3-register bank	12-bit	subprocedure return addresses
μMem	memory	16-bit data, 12-bit addr	contains hardcoded microprogram
μJumpTab	memory	12-bit data, 7-bit addr	hardcoded jump table

control path buses		
name	length	purpose
μABUS	12	Moves the address of microcode
μJBUS	7	Moves around the information for jumping
pBUS	4	Moves the register number into the cntl registers
pDBUS	5	Moves the cntl values between cntl registers and tmp

The 3 control registers are used to represent the source and destination operand locations:

control registers encoding/flags		
msb	trailing bits	significance
0	4-bit register identifier	this operand/destination is located in the specified GPR
1	0000	this operand is not used by the current operation
1	0001	this operand is located in register Val[1]
1	0010	this operand is located in register Val[2]
1	0011	this operand is located in the MDR
1	0100	this operand/destination is located at memory address Addr[0]
1	0101	this operand is located at memory address Addr[1]
1	0110	this operand is located at memory address Addr[2]

[†]These flippable shift registers support hardware-based toggling of individual bits.



2 Register transfer language

Here is the sequence of hardware actions performed by each phase of instruction processing:

2.1 Fetch phase

```
IR <- Mem[PC] # the instruction word
PC <- PC + 1
```

2.2 Decode phase

```
i <- 00
loop
        if i != 00 AND IR(imm)(i) = 1 then
                Val[i] <- Mem[PC]
                cntl[i] <- 10000 | i
                PC <- PC + 1
        elif IR(ami) = 00 then
```

```
        cntl[i] <- 10000
elif IR(ami) = 01 or 10 then
        Addr[i] <- Mem[PC]
        cntl[i] <- 10100 | i
        PC <- PC + 1
elif IR(ami) = 11 then
        MDR <- Mem[PC] # the R-type immediate word
        cntl[i] <- 10100 | i
        PC <- PC + 1
        if MDR(sam) = 000 then # register value
                regshift <- MDR
                regshift <- regshift >> 8 # reg0
                cntl[i] <- 00000 | regshift
        elif MDR(sam) = 001 then # register indirect
                Addr[i] <- Reg[MDR(reg0)]
        elif MDR(sam) = 010 then # scaled
                Val[i] <- MDR
                MDR <- Mem[PC] # the S-type immediate
                PC <- PC + 1
                MDR <- MDR >> 8
                Addr[i] <- Reg[Val[i](reg1)] << MDR
                Addr[i] <- Reg[Val[i](reg0)] + Addr[i]
        elif MDR(sam) = 011 then # doubly scaled
                Val[i] <- MDR
                MDR <- Mem[PC]
                TMP <- MDR
                PC <- PC + 1
                MDR <- MDR >> 8
                Addr[i] <- Reg[Val[i](reg1)] << MDR
                Addr[i] <- Reg[Val[i](reg0)] + Addr[i]
                MDR <- Mem[Addr[i]]
                Addr[i] <- MDR
                MDR <- TMP & 0xff
                MDR <- Reg[Val[i](reg2)] << MDR
                Addr[i] <- Addr[i] + MDR
        elif MDR(sam) = 100 then # auto increment
                Val[i] <- MDR
                Addr[i] <- Reg[Val[i](reg0)]
                Reg[Val[i](reg0)] <- Reg[Val[i](reg0)] + 1
        elif MDR(sam) = 101 then # auto decrement
                Val[i] <- MDR
                Addr[i] <- Reg[Val[i](reg0)]
                Reg[Val[i](reg0)] <- Reg[Val[i](reg0)] - 1
        elif MDR(sam) = 110 then # scaled displacement
                Val[i] <- MDR
                MDR <- Mem[PC] # the S-type immediate
                PC <- PC + 1
```

```
                        MDR <- MDR >> 8
                        Addr[i] <- Reg[Val[i](reg0)] << MDR
                        MDR <- Mem[PC] # the I-type immediate
                        PC <- PC + 1
                        Addr[i] <- Addr[i] + MDR
                elif MDR(sam) = 111 then # doubly scaled displacement
                        Val[i] <- MDR
                        MDR <- Mem[PC] # the S-type immediate
                        PC <- PC + 1
                        TMP <- MDR
                        MDR <- MDR >> 8
                        Addr[i] <- Reg[Val[i](reg0)] << MDR
                        MDR <- Mem[PC] # the I-type immediate
                        PC <- PC + 1
                        Addr[i] <- Addr[i] + MDR
                        MDR <- Mem[Addr[i]]
                        Addr[i] <- MDR
                        MDR <- TMP & 0xff
                        MDR <- Reg[Val[i](reg1)] << MDR
                        Addr[i] <- Addr[i] + MDR
                fi
        fi
        i <- i + 1
until i = 11 repeat

i <- 00
loop
        if IR(ami) = 10 then # PC-relative
                Addr[i] <- Addr[i] + PC
        fi
        i <- i + 1
until i = 11 repeat
```

2.3 Memory Load

```
i <- 00
loop
        if cntl[i](4) = 1 AND cntl[i](2) = 1 then
                Val[i] <- Mem[Addr[i]]
                if i != 00 then
                        cntl[i](2) <- 0
                fi
        fi
        i <- i + 1
until i = 11 repeat
```

2.4 Execute

```
# call function at uJumpTab label IR(opc)
```

2.5 Writeback

```
if cntl[0](4) = 0 then
        Reg[cntl[0](3..0)] <- MDR
elif cntl[0](2) = 1 then
        Mem[Addr[cntl[0](3..0)]] <- MDR
fi
# jump to the very beginning
```

2.6 Supporting Functions

```
halt:
        # bail out
and:
        # call validator_one
        MDR <- op1 & op2
        # return
or:
        # call validator_one
        MDR <- op1 | op2
        # return
xor:
        # call validator_one
        MDR <- op1 ^ op2
        # return
lsft:
        # call validator_one
        MDR <- op1 << op2
        # return
nand:
        # call validator_one
        # call and
        cntl[1] <- 10011
        cntl[2] <- 10000
        # call not
        # return
nor:
        # call validator_one
        # call or
        cntl[1] <- 10011
        cntl[2] <- 10000
        # call not
        # return
xnor:
```

```
        # call validator_one
        # call xor
        cntl[1] <- 10011
        cntl[2] <- 10000
        # call not
        # return
rsft:
        # call validator_one
        MDR <- op1 >> op2
        # return
# logical goes here
# logical goes here
# logical goes here
rasft:
        # call validator_one
        MDR <- op1 >>> op2
        # return
# illogical goes here
# illogical goes here
# illogical goes here
slt:
        # call validator_one
        # call sub
        if MDR(15) = 1 then
                MDR <- 1
        else
                MDR <- 0
        fi
        # return
sgt:
        # call validator_one
        MDR <- op2 - op1
        if MDR(15) = 1 then
                MDR <- 1
        else
                MDR <- 0
        fi
        # return
seq:
        # call validator_one
        # call sub
        if MDR = 0 then
                MDR <- 1
        else
                MDR <- 0
        fi
        # return
```

```
sne:
        # call validator_one
        # call seq
        cntl[1] <- 10011
        cntl[2] <- 10000
        # call siz
        # return
sle:
        # call validator_one
        # call sgt
        cntl[1] <- 10011
        cntl[2] <- 10000
        # call siz
        # return
sge:
        # call validator_one
        # call slt
        cntl[1] <- 10011
        cntl[2] <- 10000
        # call siz
        # return
add:
        # call validator_one
        MDR <- op1 + op2
        # return
sub:
        # call validator_one
        MDR <- op1 - op2
        # return
# compbranch goes here
# compbranch goes here
# compbranch goes here
# compbranch goes here
# compbranch goes here
# compbranch goes here
prnt:
        # call validator_three
        # print out op1
        # return
# siz goes here
siz: # handles lnot and siz
        # call validator_four
        if op1 = 0 then
                MDR <- 1
        else
                MDR <- 0
        fi
```

```
        # return
snz:
        # call validator_four
        # call siz
        cntl[1] <- 10011
        # call siz
        # return
not:
        # call validator_four
        MDR <- ~ op1
        # return
neg:
        # call validator_four
        MDR <- 0 - op1
        # return
# simpbranch goes here
# simpbranch goes here
incr:
        # call validator_six
        # call add
        # return
decr:
        # call validator_six
        # call sub
        # return
jmp:
        # call validator_seven
        PC <- op0
        cntl[0] <- 10011
        # return
jal:
        # call validator_seven
        Reg[15] <- PC
        PC <- op0
        cntl[0] <- 10011
        # return
call:
        # call validator_seven
        # call jal
        Reg[14] <- Reg[14] - 1
        Mem[Reg[14]] <- Reg[15]
        # return
ret:
        # call validator_eight
        Addr[0] <- Mem[Reg[14]]
        cntl[0] <- 10100
        # call jmp
```

```
        Reg[14] <- Reg[14] + 1
        # return
move:
        # call validator_nine
        MDR <- op1
        # return

logical: # handles land, lor, lxor
        tmp <- cntl[2]
        cntl[2] <- 10000
        # call snz
        Val[1] <- MDR
        cntl[1] <- tmp
        cntl[2] <- 10000
        # call snz
        cntl[1] <- 10001
        cntl[2] <- 10011
        IR(opc)(3) <- 0 # IR(opc) - 8
        # call function at uJumpTab label IR(opc)
        # return

illogical: # handles lnand, lnor, lxnor
        IR(opc)(2) <- 0 # IR(opc) - 4
        # call logical
        cntl[1] <- 10011
        cntl[2] <- 10000
        # call siz
        # return

compbranch: # handles blt, bgt, beq, bne, ble, bge
        # call validator_two
        IR(opc)(3) <- 0 # IR(opc) - 8
        # call function at uJumpTab label IR(opc)
        if MDR = 1 then
                PC <- op0
        fi
        cntl[0] <- 10011
        # return

simpbranch: # handles biz, bnz
        # call validator_five
        IR(opc)(2) <- 0 # IR(opc) - 4
        # call function at uJumpTab label IR(opc)
        if MDR = 1 then
                PC <- op0
        fi
        cntl[0] <- 10011
```

```
        # return

validator_one:
        # Check for 2-3 ops, and detect/handle shorthand form
        if cntl[0] = 10000 then
                # bail out loudly
        fi
        if cntl[1] = 10000 then
                # bail out loudly
        fi
        if cntl[2] = 10000 then
                cntl[2] <- cntl[1]
                cntl[1] <- cntl[0]
                if cntl[1](4) = 1 AND cntl[1](2) = 1 then
                        cntl[1](2) <- 0
                fi
        fi
        # return

validator_two:
        # Check for 3 ops, op0 is not a register
        if cntl[0] = 10000 then
                # bail out loudly
        fi
        if cntl[1] = 10000 then
                # bail out loudly
        fi
        if cntl[2] = 10000 then
                # bail out loudly
        fi
        if cntl[0](4) = 0 then
                # bail out loudly
        fi
        # return

validator_three:
        # Check for 1 op, and shuttle it into position 1
        if cntl[0] = 10000 then
                # bail out loudly
        fi
        cntl[1] <- cntl[0]
        cntl[0] <- 10011
        if cntl[1](4) = 1 AND cntl[1](2) = 1 then
                cntl[1](2) <- 0
        fi
        # return
```

```
validator_four:
        # Check for 1-2 ops, set cntl[1] if it was empty
        if cntl[0] = 10000 then
                # bail out loudly
        fi
        if cntl[1] = 10000 then
                cntl[1] <- cntl[0]
                if cntl[1](4) = 1 AND cntl[1](2) = 1 then
                        cntl[1](2) <- 0
                fi
        fi
        # return

validator_five:
        # Check for 2 ops, op0 is not a register
        if cntl[0] = 10000 then
                # bail out loudly
        fi
        if cntl[1] = 10000 then
                # bail out loudly
        fi
        if cntl[0](4) = 0 then
                # bail out loudly
        fi
        # return

validator_six:
        # Check for 1 op, and prepare for binary operation with 1
        if cntl[0] = 10000 then
                # bail out loudly
        fi
        MDR <- 1
        cntl[1] <- 10011
        # return

validator_seven:
        # Check for 1 op, op0 is not a register
        if cntl[0] = 10000 then
                # bail out loudly
        fi
        if cntl[0](4) = 0 then
                # bail out loudly
        fi
        # return

validator_eight:
        # Check for 0 op
```

```
        # return

validator_nine:
        # Check for 2 ops
        if cntl[0] = 10000 then
                # bail out loudly
        fi
        if cntl[1] = 10000 then
                # bail out loudly
        fi
        # return
```

3 Microinstruction format

Each microinstruction is encoded as a single word, with one of two possible formats:

C-type word format		
0000	control points	
15 12	11	0

J-type word format		
type	condition	jump index
15 12	11 7	6 0

3.1 C-type microinstructions

This microinstruction type is used to set control points and move data around in the data and control paths. The encoding is entirely vertical and therefore supports no parallelism beyond that encoded into the discrete control point identifiers themselves.

control point settings	
encoding	equivalent RTL
0x00	Addr[0] $\leftarrow$ Mem[Reg[14]]
0x01	Addr[i] $\leftarrow$ Addr[i] + MDR
0x02	Addr[i] $\leftarrow$ Addr[i] + PC
0x03	Addr[i] $\leftarrow$ MDR
0x04	Addr[i] $\leftarrow$ Mem[PC]
0x05	Addr[i] $\leftarrow$ Reg[MDR(reg0)]
0x06	Addr[i] $\leftarrow$ Reg[Val[i](reg0)]
0x07	Addr[i] $\leftarrow$ Reg[Val[i](reg0)] + Addr[i]
0x08	Addr[i] $\leftarrow$ Reg[Val[i](reg0)] $<<$ MDR
0x09	Addr[i] $\leftarrow$ Reg[Val[i](reg1)] $<<$ MDR
0x0a	IR $\leftarrow$ Mem[PC]
0x0b	IR(opc)(2) $\leftarrow$ 0
0x0c	IR(opc)(3) $\leftarrow$ 0
0x0d	MDR $\leftarrow$ 0
0x0e	MDR $\leftarrow$ 0 – op1
0x0f	MDR $\leftarrow$ 1
0x10	MDR $\leftarrow$ MDR $>>$ 8
0x11	MDR $\leftarrow$ Mem[Addr[i]]
0x12	MDR $\leftarrow$ Mem[PC]
0x13	MDR $\leftarrow$ Reg[Val[i](reg1)] $<<$ MDR

0x14	MDR ← Reg[Val[i](reg2)] < < MDR
0x15	MDR ← TMP & 0xff
0x16	MDR ← op1
0x17	MDR ← op1 & op2
0x18	MDR ← op1 + op2
0x19	MDR ← op1 – op2
0x1a	MDR ← op1 < < op2
0x1b	MDR ← op1 > > op2
0x1c	MDR ← op1 > > > op2
0x1d	MDR ← op1 ^ op2
0x1e	MDR ← op1 | op2
0x1f	MDR ← op2 – op1
0x20	MDR ← ~ op1
0x21	Mem[Addr[cntl[0](3..0)]] ← MDR
0x22	Mem[Reg[14]] ← Reg[15]
0x23	PC ← PC + 1
0x24	PC ← op0
0x25	Reg[14] ← Reg[14] + 1
0x26	Reg[14] ← Reg[14] – 1
0x27	Reg[15] ← PC
0x28	Reg[Val[i](reg0)] ← Reg[Val[i](reg0)] + 1
0x29	Reg[Val[i](reg0)] ← Reg[Val[i](reg0)] – 1
0x2a	Reg[cntl[0](3..0)] ← MDR
0x2b	TMP ← MDR
0x2c	Val[1] ← MDR
0x2d	Val[i] ← MDR
0x2e	Val[i] ← Mem[Addr[i]]
0x2f	Val[i] ← Mem[PC]
0x30	cntl[0] ← 10011
0x31	cntl[0] ← 10100
0x32	cntl[1] ← 10001
0x33	cntl[1] ← 10011
0x34	cntl[1] ← cntl[0]
0x35	cntl[1] ← tmp
0x36	cntl[1](2) ← 0
0x37	cntl[2] ← 10000
0x38	cntl[2] ← 10011
0x39	cntl[2] ← cntl[1]
0x3a	cntl[i] ← 00000 | regshift
0x3b	cntl[i] ← 10000
0x3c	cntl[i] ← 10000 | i
0x3d	cntl[i] ← 10100 | i
0x3e	cntl[i](2) ← 0
0x3f	i ← 00
0x40	i ← i + 1
0x41	regshift ← MDR
0x42	regshift ← regshift > > 8

0x43	tmp ← cntl[2]

3.2 J-type microinstructions

This microinstruction format is used for goto operations altering the micro–program counter and microstack. The type field determines what type of control flow change is occurring, as well as which of the immediates will actually be used.

meaning of the type field		
encoding	function	condition and jump fields
0x0	(C-type microinstruction)	N/A
0x1	jump to jump table label	only jump used
0x2	jump to beginning of the microprogram	both ignored
0x3	call function at jump table label	only jump used
0x4	call function at jump table address IR(opc)	both ignored
0x5	return from a call	both ignored
0x6	halt system normally	both ignored
0x7	halt system due to failed operand validation	both ignored
0x8	print contents of op1	both ignored
0x9	(invalid)	N/A
0xa	branch to jump table label	both used
0xb	branch to beginning of the microprogram	both ignored
0xc	conditionally call function by jump table	both used
0xd	conditionally call function at IR(opc)	only condition used
0xe	conditionally return from a call	only condition used

If the goto is conditional, the condition bits determine the sufficient clause as follows:

possible conditional sufficient clauses	
encoding	expansion
0x00	IR(ami) = 00
0x01	IR(ami) = 01 or 10
0x02	IR(ami) = 10
0x03	IR(ami) = 11
0x04	MDR = 0
0x05	MDR = 1
0x06	MDR(15) = 1
0x07	MDR(sam) = 000
0x08	MDR(sam) = 001
0x09	MDR(sam) = 010
0x0a	MDR(sam) = 011
0x0b	MDR(sam) = 100
0x0c	MDR(sam) = 101
0x0d	MDR(sam) = 110
0x0e	MDR(sam) = 111
0x0f	cntl[0] = 10000
0x10	cntl[0](2) = 1
0x11	cntl[0](4) = 0
0x12	cntl[1] = 10000
0x13	cntl[1](4) = 1 AND cntl[1](2) = 1
0x14	cntl[2] = 10000
0x15	cntl[i](4) = 1 AND cntl[i](2) = 1
0x16	i != 00
0x17	i != 00 AND IR(imm)(i) = 1
0x18	i = 11
0x19	op1 = 0

Jump destinations are encoded as addresses in the jump table, which is stored in a dedicated memory module within the control unit. Each location therein is analogous to a label, and contains a microprogram memory address. The precise number and ordering of labels within this table are unspecified, except that the lowest addresses are to be used for the user-facing instruction opcodes in the exact order enumerated under section 3.1 of the ISA document. This requirement is imposed to allow efficient decoding of user-generated instruction words.