Function Calling, Message Passing, Stack-Group Switching

****

This file is confidential. Don't show it to anybody, don't hand it out to people, don't give it to customers, don't hardcopy and leave it lying around, don't talk about it on airplanes, don't use it as sales material, don't give it as background to TSSEs, don't show it off as an example of our (erodable) technical lead, and don't let our competition, potential competition, or even friends learn all about it. Yes, this means you. This notice is to be replaced by the real notice when someone defines what the real notice is.

****

Stack Groups on the I Machine

A stack group is the object of computation. It contains the memory image of a process. This includes many things, all of which eventually need to be enumerated. For now, the list includes the following:

- Control Stack
  - Control Stack Base
  - Control Stack Pointer
  - Control Stack Limit
  - Control Stack Extra Limit
- Frame Pointer
- Local Pointer
- Binding Stack
  - Binding Stack Base
  - Binding Stack Pointer
  - Binding Stack Limit
- Data Stack
  - Data Stack Base
  - Data Stack Pointer
  - Data Stack Limit
- Catch Block Head
- PC

- Control Register
- Continuation Register
- Floating Point State?
  - Mode - rounding, underflow-to-0, ?
  - Status - sticky-over/underflow, ?

*

****

Notes: What to do with these items: OK-to-run bit maintained in wired storage by paging system. Different set of known-by-microcode fields? More registers to be saved as part of the context.

****

Stacks

The architecture defines three stacks:

- control stack,
- binding stack, and
- data stack.

Each type of stack is described in the sections that follow. All the stacks grow in the direction of increasing memory addresses. A stack pointer addresses the top word on a stack. A stack limit is the address of the highest location that can be used. A stack base register addresses the lowest word in the stack. Stack limit and stack base registers are not hardware registers, just software slots in the stack group.

Control Stack

The control stack holds control information necessary on a per function invocation basis. It also holds the arguments and local and temporary variables of a function.

Control Stack Frames

The environment of an executing function is stored in a frame on the control stack. A control stack frame consists of a two-word header, the arguments, and then the local variables and temporaries. Note that there are no separate copies of the arguments for caller and callee; in this respect the I Machine architecture is like the LM-2 and unlike the 3600.

See Figure CONTROL-STACK.SCH.

[Figure caption: An I-machine control stack frame.]

The first word in a control stack frame header contains a saved copy of the caller's Continuation register. This is either the caller's caller's PC or the address of a function the caller is going to call later. The second word in a frame header contains a saved copy of the caller's Control register.

When a function returns, the saved values are restored into the Continuation and Control registers. At the same time, the caller's PC is restored from the previous contents of the Continuation register. When a function is first entered, the contents of the Continuation register normally points at the next instruction after a finish-call instruction, except in a trap handler, where it points either at the instruction that trapped or at the following instruction, depending on the type of trap.

Note that the Continuation and Control registers stored in a frame header belong to the caller's frame, not to the frame where they are stored. The values for the current frame are kept in live (hardware) registers instead of the stack because special hardware uses them. The maximum size of a control stack frame is 256 words.

Base Registers

There are three base registers that point to the current control stack frame. These can be used to calculate instruction operand addresses. See the section "Macroinstruction Set".

The frame pointer (FP) points to the first word of the frame header. This register is used to locate the function's arguments, which start at a fixed offset past FP. The local pointer (LP) points after the spread arguments. (Spread arguments are arguments that are not part of a **&rest** parameter.) It is used to locate local variables to the function. The stack pointer (SP) points to the highest word in the frame. SP is incremented or decremented as execution proceeds and pushes or pops the stack. These registers are discussed further in another section. See the section "Registers Important to Function Calling and Returning".

Binding Stack

Binding is the temporary replacement of a memory cell's contents. The Binding Stack saves the address and contents of memory cells that have been bound so the original contents can later be restored. Note that binding affects only the contents of a cell, not its cdr code.

Entries on the binding stack are two words long. The fields of an entry are as follows:

Word	*Position*	*Field*	*Comments*
0	<38>	Binding-stack-chain-bit	=1 if the previous entry is for the same frame.
0	<37:0>	Binding-stack-cell	Locative to the memory cell that is bound.
1	<37:0>	Binding-stack-contents	Saved contents of bound cell.

The binding-stack-cell field contains a **user::dtp-locative** pointer to the memory

cell that is bound. This indicates which location has had its contents temporarily replaced. In the case of a *dynamic closure,* however, a new memory cell is created, and the old value cell is loaded with a **user::dtp-external-value-cell-pointer** to this new cell. The new cell is referenced by the closure.

The binding-stack-contents field contains the contents of the bound cell. Bindings do not persist across stack groups, and must be undone when control is transferred to another group. Binding-stack-contents contains the "former" contents of the cell when the binding stack belongs to the currently executing stack group; otherwise it contains the "current" contents of the cell. See the section "Stack-Group Switching".

The binding-stack-chain-bit is 1 if the previous entry on the binding stack is associated with the same function invocation as this entry. This bit is set by the **user::bind** instruction, and groups entries on the binding stack into frames associated with a function. Binding stack frames are removed at function return time.

The Binding Stack Pointer points to the top of the binding stack (word 1 of the topmost entry]) There is also a Binding Stack Limit register.

Bindings are performed by the **user::bind-locative** or **user::bind-locative-to-value** instruction. A bind instruction checks the Control register binding cleanup bit. If this bit is 0, then this binding is the first associated with the current frame. The instruction will set the binding cleanup bit in the Control register, and set the chain bit for the entry on the binding stack to 0. If the cleanup bit is 1, then there are already bindings associated with the current frame. The instruction will set the chain bit for the entry to 1.

Note that an unbind instruction (**user::unbind-n** or **user::%restore-binding-stack**) will clear the Control register cleanup bit if it removes an entry from the binding stack with the chain bit 0.

*

****

Notes: Deleted -- 0 <39> Binding-stack-closure-bit 1 if this binding was created by dynamic closure or instance (for the debugger). The binding-stack-closure-bit is just for the debugger. The hardware never sets this bit and never looks at it.

Note the change in the paragraph following the table.

What about a base register?

>>> Where is unbinding described - instruction chapter?

****

Data Stack

The purpose of the data stack is to provide an allocation area for temporary data whose lifetime is associated with a function's lifetime. This allows less expensive allocation/deallocation than the general mechanism.

This is implemented in software in the same manner as on the 3600.

Registers Important to Function Calling and Returning

The following processor registers are relevant to function calling and returning:

Program Counter (PC)
: Address of the current instruction.
user::dtp-even-pc or **user::dtp-odd-pc**

Frame Pointer (FP)
: Address of the current stack frame.
user::dtp-locative

Local Pointer (LP)
: Address of the local-variable part of the current stack frame.
user::dtp-locative

Stack Pointer (SP)
: Address of the highest in-use word in the stack.
user::dtp-locative

Continuation register (CONT)
: Address of the first instruction to be executed after the next function call or return.
user::dtp-even-pc or **user::dtp-odd-pc**

Control register (CR)
: A bunch of bits and fields to be described below.
user::dtp-fixnum

The program counter contains the address of the current instruction.

The frame pointer points to the first word of the control stack frame header. This register is used to locate the function's arguments, which start at a fixed offset (2) past FP. It can also be used to locate the function's locals if the function does not accept a **&rest** argument. When a function returns, the SP is set to FP-1 to remove the function's frame.

After a finish-call instruciton, the local pointer points to the word after the spread arguments. Thus it points to the rest argument if there is one; otherwise it points to the first local variable. When there are optional arguments and no rest argument, LP points at the first optional argument not supplied by the caller, if there is one.

LP is used to locate local variables to the function. FP cannot always be used for this since in general the number of arguments the function accepts is variable. LP may be adjusted by the **user::entry** and **user::locate-locals** instructions.

The stack pointer points to the highest word in the control stack. SP is incremented or decremented as execution proceeds and pushes or pops the stack.

The Continuation register contains the address of the instruction to be executed after the next finish-call or return instruction. Whether this is the return address in the caller, or the first instruction in a function about to be called, depends on context. It is the address of the function to call between the start-call and finish-call instructions, and the return address in the caller between the finish-call and return instructions.

The Control register contains a fixnum with several packed fields:

Position	*Size*	*Name*
<7:0>	8 bits	Argument-size
<17>	1 bit	Apply
<19:18>	2 bits	Value-disposition
<26:24>	3 bits	Cleanup-bits
<26>		cleanup-catch
<25>		cleanup-bindings
<24>		trap-on-exit
<21:20>	2 bits	Instruction-state

```
<31:30>      1 bit   Trap-mode
 <8>         1 bit   Extra-argument
<16:9>       8 bits  Frame-size-of-caller
 <22>        1 bit   Call-started
 <23>        1 bit   Cleanup-in-progress
 <29>        1 bit   Instruction-trace
 <28>        1 bit   Call-trace
 <27>        1 bit   Trace-pending
```

Argument-size is the offset of LP from FP in the frame. It is used to restore the LP when the function resumes execution after calling another function. It is also used by the **user::entry** instruction to determine how many explicit arguments were supplied with the call. This field is set by the finish-call instructions (for the new frame). It is also adjusted by the **user::locate-locals** instruction.

Apply, if 1, indicates that a rest argument list has been supplied following the spread arguments and is stored in LP|0. This bit is set by the finish-call instructions, and is used to implement the Common Lisp **apply** function. This can be reset by the **user::entry** instruction doing a pull-apply-args operation.

Value-disposition specifies what the caller wants done with the result(s) produced by the function. It is set by the finish-call instructions. All three bits are cleared by a finish-call instruction. The interpretation of value-disposition is:

0	Effect	The function has been called for effect. Discard any values the function may produce.
1	Value	Only a single value is desired by the caller. Push this on the control stack, discarding any extra values.
2	Return	The value(s) returned by the function are also the value(s) returned by the caller. Pass the value(s) along to this frame's caller.
3	Multiple	The caller wants multiple values returned. Push any number of values on the stack, followed by a fixnum specifying the number of values.

The requested disposition is performed by a return instruction. Returned results are pushed onto the stack after the function's frame has been removed from the stack. If a function terminates abnormally, it does not return a value so Value-disposition is ignored.

Cleanup-bits specifies what actions need to be performed prior to removing the function's frame from the control stack. The actions are normally performed by a return instruction. In the case of abnormal termination, these actions are performed by the **throw** function (which uses a return instruction internally). The bits are:

Catch	This bit indicates there are catch/unwind-protect blocks in the frame. The catch cleanup bit is set whenever a catch or unwind-protect block is created. The bit is cleared when the outermost catch/unwind-protect block in a frame is destroyed. See the section "Catch Instructions".
Bindings	This bit indicates there is a non-empty binding-stack frame associated with this control-stack frame, in other words that this function has bound some special variables. This bit is set by the binding instructions (**user::bind-locative**) and can be

cleared by the unbinding instructions (**user::unbind-n**). See the section "Binding Instructions".

Trap-on-Exit — This bit causes a trap to software when the frame is exited. Used for bottom frame in stack, debugger c-X command, phantom stacks, metering, and so forth. The software can use the cdr-code bits of the two header words in the frame, which are initially set to 11 by the hardware to distinguish these cases. The trap-on-exit bit is set and cleared only by software, and only in copies of the Control register saved in memory, not in the live register.

For details: See the section "Frame Cleanup".

Instruction-state is set to zero upon successful completion of any instruction. If these bits are nonzero, the next instruction may behave differently than normal. If an instruction is interrupted in the middle these bits are sometimes set nonzero to allow the instruction to be retried correctly if it has not completely restored its initial state.

Trap-mode controls the handling of exception traps. The four modes, explained elsewhere (See the section "Trap Modes".), are:

0 Emulator

1 Extra Stack

2 High-Speed I/O

3 FEP

The trap-mode field is adjusted when a trap is taken. It is set to (max 1 current-trap-mode) by the **user::%allocate-list-block** or **user::%allocate-structure-block** instruction.

Extra-argument is set to 1 to indicate an extra argument has been supplied to the function by a start-call instruction. This happens when calling a lexical closure, a generic function, an instance, or any interpreted function or illegal data type. See the section "Starting a Function Call". This bit is just used to transmit information from a start-call instruction to the corresponding finish-call instruction and then is no longer needed. It is cleared by a finish-call instruction.

Frame-size-of-caller contains the size of the caller's stack frame (callee's FP minus caller's FP). It is used by return instructions to locate the start of the caller's frame when the function returns. This field is set by the finish-call instructions.

Call-started is set by start-call instructions and cleared by the finish-call instructions.

Cleanup-in-progress

Trace-pending when 1, causes a trap to occur before the next instruction executes. Note that a sequence break can intervene before the trap actually goes off. There is only one trap vector location for trace-pending, regardless of the semantic significance of the trap to the software. If a return instruction restores a control register value with the trace-pending bit set, the trap occurs after completion of the return instruction and before execution of the instruction returned to. The interaction of trace-pending with the repeated returns caused by Value-disposition Return is not defined.

Instruction-trace when 1 at the beginning of an instruction, causes completion of the instruction to set trace-pending and causes a trap before the next instruction executes. If a post-trap occurs when instruction-trace is 1, trace-pending is set in the control register saved as part of taking the trap. This is not true of a pre-trap. If a return instruction restores a control register value with the instruction-trace bit set, the instruction returned to is executed before the trap occurs.

Call-trace when 1, causes the finish-call instructions to set trace-pending and causes a trap before the first instruction of the called function executes. If stack overflow occurs simultaneously, trace-pending is set in the saved control register in the frame header of the stack overflow trap handler's frame. When the stack overflow handler returns, the trace trap occurs. Call-trace does not affect the implicit finish-call performed when a trap occurs, because call-trace gets cleared first.

Spare bit not allocated yet.

*

****

Notes:

The exact value stored in the argument-size field may be adjusted to simplify the hardware and/or microcode. For example, it might store LP-FP-2 instead of LP-FP.

About the instruction-state field -- [Details elsewhere.] **This is probably going away.**

The phantom-stack scheme, if we decide to implement it, may require a bit that is 1 if any objects in this frame have been evacuated from the stack. The bit changes the way a few instructions execute.

DCP would like to have the **user::abort-call** instruction back.

****

Function Calling

A function call requires three different actions: specifying the function to call, pushing the arguments to the function, and finishing the call by building the new stack frame and entering the target function. The instructions that accomplish these actions are described below.

Starting a Function Call

A function call is begun by executing one of the start-call types of instructions, whose single argument is the function to be called. These instructions create the header of the callee's stack frame, possibly push an extra argument onto the stack, and set the continuation according to the type of function being called.

The most general start-call instruction, **user::start-call** itself, takes its argument from the top of stack or from a local variable. Several full-word instructions are also supplied; these contain an address that specifies the function and possibly its data type. In summary:

user::start-call Takes a general stack operand.

user::dtp-call-compiled-even and **user::dtp-call-compiled-odd**
Address a compiled-function directly, specifying whether to

start with the even or odd halfword instruction in the addressed location.

user::dtp-call-indirect
Addresses a function cell and fetches its contents.

user::dtp-call-generic
Addresses a generic function directly.

Each full-word start-call type of instruction comes in prefetching and nonprefetching versions. Semantically these are identical, but the prefetching version is a hint to the hardware that a finish-call instruction appears soon enough after the start-call instruction that it would be worthwhile to prefetch the first few instructions of the called function rather than continuing to fetch ahead instructions from the calling function. The decision of when to use the prefetching version is up to the compiler; it is probably appropriate when there are no nested function calls in the arguments and the number of instructions in the arguments is less than a certain constant (around half a dozen). Prefetching makes the ensuing finish-call operation run faster. The hardware does not necessarily actually prefetch when the prefetching version is executed; it depends on the particular instruction, on the data type of the function, and on how complex the hardware turns out to be. The prefetching versions of the indirect and generic calls are almost certainly not treated any differently from the normal versions by the hardware: they exist entirely for software reasons.

The start-call instructions push the Continuation and Control registers (in that order) onto the control stack with their cdr codes both set to 3; they will become the header of the callee's control stack frame. After the Control register is pushed, the CR.call-started bit is set to 1.

Depending on the data type of the function being called, a start-call instruction may push a third word which is called the "extra argument". Its cdr code is set to 0. All data types other than **user::dtp-compiled-function** receive an extra argument. In the case of instance or generic function, the word pushed on the stack is just a placeholder for the real extra argument the function will be called with, since this cannot be computed until the first argument is known. (This extra argument mechanism is necessary because in general the data type of the function being called is not known until run time. Note that if a method is called directly, as from a combined method, or if a lexically internal function is called directly, as from its parent, the extra argument is passed instead as a normal argument. Any given function always receives its arguments in the same format, and does not need to know whether the first argument was supplied normally by the caller or was an "extra" argument.)

If the start-call instructions push an extra argument, they set the extra-argument bit in the Control register to 1; otherwise they clear the bit to 0. This information is saved for the finish-call instruction. The setting or clearing of this bit takes place after the Control register is saved on the stack.

After Continuation and Control registers are saved, the Continuation register is set to a PC value pointing at the beginning of the function to be called (the argument of the start-call). Depending on the data type of the function, this continuation can be computed from the function itself or can be fetched from one of 64 trap-vector locations, indexed by the data-type of the function. The effect of the function's data type on a start-call is as follows:

compiled-function There is no extra argument. The continuation is set to

	user::dtp-even-pc with the address of the function.
symbol	Fetch the contents of the symbol's function cell and try again. Trap if the function cell contains **user::dtp-null.**
instance	Push the instance as the extra argument. The continuation comes from the trap vector.
generic	Push the generic function as the extra argument. The continuation comes from the trap vector.
lexical closure	Fetch the enclosed function and the environment from memory. If the enclosed function is compiled, push the environment as the extra argument and set the continuation to dtp-even-pc and the function's address, producing a call to the enclosed function with the environment as its extra argument. If the enclosed function is not compiled, push the lexical closure as the extra argument and take the continuation from the trap vector.
anything else	Push the original function as the extra argument. Use the data type of the function as an index into the trap vector to fetch the appropriate interpreter function and set the continuation to that.

After a start-call instruction the continuation is guaranteed to be a PC pointing into a compiled function, assuming the trap-vector has been initialized correctly.

For the instance and generic function cases, the real function (the method) and the real extra argument (the mapping table) cannot be computed until the value of the first argument is known, so these have to be deferred until a finish-call instruction is executed.

Note that after doing a start-call a program does not know the exact depth of the stack, because it does not know whether an extra argument was pushed. The compiler avoids using SP-relative addressing to access variables deeper in the stack than the incipient frame header.

>>> Picture of stack at end of start-call

Pushing the Arguments

After starting a function call, the caller computes the arguments and pushes them onto the stack, in order. Results of instructions normally are **user::cdr-next,** to facilitate the linking of the arguments into a list to be passed to an **&rest** argument. The resetting of the final cdr code is performed by the **user::entry** instruction.

>>>Picture of stack at end of pushing args

Finishing the Call

After starting a function call and pushing the arguments, the caller executes a finish-call instruction. This instruction builds the new stack frame, checks for control stack overflow, and enters the callee at the appropriate starting instruction.

Instructions at the beginning of the callee are in charge of checking the number of arguments and rearranging them to suit its needs, or signalling an error if the

wrong number of arguments were supplied. Every compiled function should contain code to do this, but the linker (which places **user::dtp-call-compiled-even** or **user::dtp-call-compiled-odd** instructions into compiled callers) can optimize calls by bypassing those instructions and arranging for the called function to be entered directly at the right place.

There are two finish-call instructions, **user::finish-call-n**, and **user::finish-call-tos** which differ only in how they obtain their argument. **user::finish-call-n** takes its argument as an 8-bit field of a 10-bit immediate, and **user::finish-call-tos** pops its argument from the top of stack.

The operand, called *N-Args*, indicates the number of arguments explicitly supplied with the call. Thus it does not include the apply argument, if any, or the extra argument, if any.

Three additional bits supplied with the instruction, I<9:8> of the 10-bit immediate field, are used as follows.

Value-disposition A 2-bit field taken from the operand field that specifies what to do with the result(s) produced by the function being called:

0	Effect	The function is being called for effect. Discard any values it may produce.
1	Value	Only a single return value is desired. Discard any additional values the function may produce.
2	Return	The value(s) returned by the function being called are also the value(s) returned by this function. Pass the value(s) along to this frame's caller. This is illegal in nested calls.
3	Multiple	Multiple values are desired. These should be returned along with a fixnum specifying the number of values returned.

Apply A 1-bit field taken from the opcode, which is a 1 if the top word in the stack is a list of arguments. The list may be spread or packed by the entry instruction. This implements the Common Lisp **apply** function.

There are a number of applications for calling a function with the number of arguments not known at compile time, where the arguments do not come from a list, including the **user::%finish-function-call** and **multiple-value-call** special forms and things built on them. These are handled by using the **user::finish-call-tos** instruction.

The operations of finish-call are described sequentially below, although in the actual hardware many of them happen in parallel.

The first thing finish-call does is to check for Apply = 1 but the top word on the stack is **nil** (an empty list). In this case it pops the stack and clears its copy of the Apply bit, turning into a normal call. This canonicalization simplifies the argument match-up procedure described later.

The finish-call instruction next builds the new stack frame with the following procedure:

```
FP              <=  SP - Apply - N-Args - cr.extra_argument - 1

LP              <=  SP + 1 - Apply              ;this could be past SP

Continuation    <=  the address of the next instruction after the
finish-call.

SP, Binding-stack-pointer, and Data-stack-pointer are unchanged.

Save the old contents of Continuation temporarily (see below).
```

The control register is adjusted as follows:

```
Argument-Size           <=  N-Args + cr.extra_argument + 2
                            (that is, new LP - new FP)

Apply                   <=  Apply bit in the instruction

Value-Disposition       <=  Value Disposition bits in the instruction

Cleanup-Bits            <=  0

Instruction-State       <=  0

Trap-Mode               <=  unchanged

Extra-Argument          <=  0  ;actually this doesn't matter

Frame-size-of-caller    <=  new FP minus old FP

Call-started            <=  0
```

After building the new frame, finish-call checks for a control stack overflow. If the stack overflows in normal mode, it switches to extra stack mode and traps to the stack-overflow handler. Extra-stack mode turns on function trace. If the stack overflows in extra stack mode, the machine halts with a fatal error.

The finish-call instruction copies cr.call-trace to cr.trace-pending, forcing a trace pre-trap upon execution of the next instruction if call-trace was 1.

Finally execution proceeds with the instruction at the halfword address specified in the Continuation register before it was set to the return address.

Trapping Out of Finish-call and Restarting

Traps in the finish-call instructions always occur after building the new frame and setting the Control register, the Continuation register, and the Program Counter to their new values. Thus any trap occurring in a finish-call instruction looks like a pre-trap in the first instruction of the called function. No special action is required to restart after such a trap.

Aborting Calls

It is sometimes necessary to abort a call that has been started, instead of finishing it with a finish-call instruction. Aborting a call consists of popping the stack back to the level before the call was started and restoring some of the Continuation and

Control register values saved by the start-call instruction. This is performed by Lisp code.

*

****

Notes: About canonicalization: [This is what it does on the 3600; it's actually a matter for detailed design of the I Machine to determine whether this canonicalization is worth anything to it.]

NOTE: the exact value stored in the N-Args operand of the FINISH-CALL instruction will be adjusted to suit the convenience of the hardware. For example, the computation of the new FP might be eased by storing N-Args + Apply + 1 in this field.

Trap handling in finish-call is no different from any other trap. After the FINISH-CALL completes, and before the instruction it jumped to is executed, the trap is taken, building another new frame. When the trap handler returns, the first instruction in the called function is executed.

****

Function Entry

A compiled function starts with a sequence of instructions that are involved in receiving the arguments. The first instruction is known as the entry instruction. It is followed by a possibly-empty sequence of instructions known as the entry vector. The function can be entered at the entry instruction, which will check the number of arguments and select the first instruction to be executed, either an element of the entry vector or the first instruction after the entry vector. Alternatively, this selection can be made by the linker when the number of arguments is known statically, and the function can be entered directly at an element of the entry vector or at the first instruction after the entry vector. In either case, execution proceeds from the selected instruction according to normal instruction sequencing, possibly executing additional instructions from the entry vector. After completing the entry vector, some additional argument-taking instructions may be executed, depending on the particular function. Thus a compiled function consists of:

Object header (2 words)
Entry instruction
Entry-vector instructions
Other argument-taking instructions
Body instructions

See the section "Representation of Compiled Functions".

Each entry-vector element is two half-word instructions long. For each **&optional** or **&rest** argument, there is an element of the entry vector. (This includes an automatically-generated **&rest** argument in a function with **&key** arguments.) The element of the entry vector corresponding to an argument contains instructions that are executed if that argument is not supplied by the caller. These instructions compute the default value (**nil** for a **&rest** argument) and push it on the stack. If this computation will not fit in an entry-vector element, the compiler inserts a branch to the rest of the code, which ends in a branch back. If the computation is smaller than the size of an entry-vector element, it ends with a branch to the next element, some kind of no-op padding, or cdr-code sequencing that skips an instruction, whichever is fastest.

The entry instruction contains the following information:

Number of required arguments
Number of optional arguments
Number of rest arguments (zero or one)

An entry instruction performs an argument match-up process that either traps (for wrong number of arguments) or adjusts the stack and then branches to the appropriate instruction of the entry vector, or to the instruction after the entry vector. The first entry-vector element follows immediately after the entry instruction. Adjusting the stack is done by performing one of two operations described later: pull-apply-args or push-apply-args.

The following conditions are computed by an entry instruction:

- Too few spread arguments (*N-Args+2* < *min-args+2*) *[min-args+2 = req]*
- Too many spread arguments (*N-Args+2* > *max-args+2*) *[max-args+2 = req+opt]*
- Maximum spread arguments (*N-Args+2* = *max-args+2*)
- Rest argument wanted (rest-arg = 1)
- Rest argument supplied (cr.apply = 1)

Note that the argument comparisons are all biased by plus 2. cr.arg-size in the control register is two greater than the actual number of arguments in the frame because it includes the two frame header words (this makes **return** faster). To simplify these entry comparisons, the arguments *min-args* and *max-args* in the entry instructions are correspondingly biased by two.

If "rest argument wanted" and "rest argument supplied" are both false, this is the simple case. If there are too few or too many arguments, take a wrong number of arguments trap. Otherwise, enter the function at entry-vector element (N-Args - min-args); this skips over the default-initialization instructions for those optional arguments that had values supplied.

If "rest argument wanted" is false and "rest argument supplied" is true, then if there are too few spread arguments do a pull-apply-args operation. Otherwise, take a wrong number of arguments trap because there are too many arguments.

If "rest argument wanted" is true and "rest argument supplied" is false, then if there are too many spread arguments do a push-apply-args operation. Otherwise, not enough arguments were supplied to create a rest argument. If there were too few spread arguments, take a wrong number of arguments trap. Otherwise, enter the function at entry-vector element (N-Args - min-args); this skips over the default-initialization instructions for those optional arguments that had values supplied. The last element of the entry vector will be executed; it pushes **nil** to default the rest argument.

If "rest argument wanted" and "rest argument supplied" are both true, a push-apply-args operation is required if there are too many spread arguments; otherwise a pull-apply-args operation is required unless there are the maximum number of spread arguments. If neither operation is required, set the cdr code of the top word in the stack to **user::cdr-nil** and enter the function at entry-vector element (max-args - min-args + 1). This skips over the default-initialization instructions for the optional arguments and for the rest argument.

Push-apply-args

The push-apply-args operation is invoked when there are too many spread arguments and a rest argument is wanted. It pushes some spread arguments back into the rest argument, after which the function is started at its all-arguments-supplied entry point. This operation does not involve any memory references nor any possibility of trapping.

In detail, push-apply-args does the following:

- Set the cdr code of the last word in the stack to **user::cdr-nil.**
- If a rest argument was supplied, set the cdr code of the second to last word in the stack (the last spread argument) to **user::cdr-normal.**

  Since arguments are pushed with **user::cdr-next,** the stack now contains a list of all of the arguments.
- Make a rest argument out of the arguments after the max number of spread arguments wanted by the function by creating a **user::dtp-list** pointer to (frame-pointer + max-args + 2). Push this rest argument onto the stack.
- If apply=0, leave cr.arg_size and cr.apply alone. They describe the arguments preceding the rest argument that was just pushed, which is regarded as a local variable of the callee rather than an argument supplied by the caller.
- If apply=1, increment LP and cr.arg_size, and leave cr.apply alone. LP now points at the revised rest argument that was just pushed, instead of the original rest argument, which has been turned into the cdr word of a two-word cons.

The function is entered at entry vector element (max-args - min-args + 1) [past the **&rest** argument default].

Pull-apply-args

The pull-apply-args operation is invoked when there are fewer than the maximum number of spread arguments and a **&rest** argument was supplied. It pulls some additional spread arguments out of the **&rest** argument.

In detail, pull-apply-args pops the list of arguments off the stack, extracts some arguments from the list, pushes them into the stack, pushes the tail of the list into the stack, adjusts cr.arg_size, and retries the argument match-up process. If the **&rest** argument is too short, the cr.apply bit is turned off; the retry may then signal too few arguments or may simply default some optional arguments. The pull-apply-args operation occurs even if the callee did not want a **&rest** argument; if the desired number of arguments are pulled out of the **&rest** argument and more arguments remain, a wrong number of arguments trap will occur when the argument match-up process is retried.

Following the entry vector, the following instructions may appear.

In a function with both **&optional** and **&rest** arguments, it is necessary to adjust the LP register to make sure that the **&rest** argument is in LP|0. (If there is a **&rest** argument but not **&optional** arguments, LP will already contain the correct value.) Any function that takes a **&rest** argument may be called with an arbitrary number of spread arguments; push-apply-args will generate the correct **&rest**

argument, but there remains an arbitrary distance between FP and SP at the time the function is entered and starts creating its local variables. This is the reason why the local pointer exists; it permits such functions to address their local variables. Functions without **&rest** arguments do not normally use the local pointer. The first instruction after the entry vector, when there are both **&optional** and **&rest** arguments, is a **user::locate-locals** instruction, which does the following:

- Push (cr.arg_size - 2) onto the stack, as a fixnum. This is the number of spread arguments that were supplied, which is less than the number of spread arguments now in the stack if some **&optional** arguments were defaulted.
- Set LP to (new-SP - 1). Thus LP|0 is the **&rest** argument and LP|1 is the argument count. *new-SP* here refers to the SP after the incrementation caused by the *locate-locals* instruction.
- Set cr.arg_size to (LP - FP) as always.

The next step is to create the auxiliary **user::supplied-p** variables for optional arguments. Each of these variables is stored as a local variable (after all the arguments) whose initial value is created by arithmetic comparison between the number of arguments supplied and an appropriate constant. The number of arguments supplied is cr.arg_size - 2 except in functions with both **&optional** and **&rest** arguments, where it is LP|1. The computation can be performed with a sequence of existing instructions. The initialization of **user::supplied-p** variables recomputes information that waas available while exectuting the entry vector, but there was no space in the stack to store that information then.

The next step takes care of any arguments that were declared special by binding the special variables to the values using the normal instructions for that purpose. If there are any non-special arguments after the special arguments, orphan words will be left in the stack since the values of the special arguments cannot be popped off.

If there are problematic dependencies among optional-argument default-value computations, special care is required. A problematic dependency occurs if the default value for an optional argument depends on a **user::supplied-p** variable of a previous optional argument or can be affected by a previous argument that is declared SPECIAL. The 3600 handles this with an alternate function entry sequence that the compiler generates if necessary. The I Machine will handle it by using **nil** as the default value in the entry vector and then generating code after the entry vector that tests whether the argument was supplied (just as if initializing a **user::supplied-p** variable) and if not computes the default value and pops it into the argument's slot in the stack. This code is interleaved with the binding of special variables so that everything happens in the right order.

Note that if a **user::supplied-p** variable is used in a read-only way, the value can simply be computed where it is needed, rather than waiting until a stack slot is allocated for the variable, and the problematic case need not occur.

The next step is to compute the values of **&key** arguments and push them on the stack as local variables. This is done with code that looks at the rest argument, just as on the 3600.

This completes the function entry sequence. If the body of the function creates local variables (or **&aux** variables) pushing the initial value of the variable on the

stack allocates a stack slot, just as on the 3600. These stack slots can be addressed from the top of the stack frame (relative to SP) or can be addressed from the bottom of the stack frame (relative to FP if the function does not take a **&rest** argument or relative to LP if it does).

Trapping Out of Entry and Restarting

Traps can occur in an entry instruction. Error traps such as wrong number of arguments are handled in a totally ordinary way.

The pull-apply-args operation references memory, so it is possible for it to trap. Usually, however, the **&rest** argument will be a cdr-coded list in the stack and no trap will occur; these cases are handled quickly by microcode. It is implementation-dependent whether the pull-apply-args microcode handles the full generality of **car** and **cdr,** including non-cdr-coded lists and invisible pointers. Cases it does not handle make the stack frame self-consistent and then call a special trap handler that performs the rest of the pull-apply-args operation and then returns to the **user::entry** instruction, which will not need a pull-apply-args this time. (The trap handler needs to use the **user::return-kludge** instruction.) Ordinary traps, such as page faults, are handled by making the state of the stack frame self-consistent and then calling the ordinary trap handler. After the reason for the trap has been rectified, the trap handler returns to the **user::entry** instruction, which will go back into pull-apply-args and should make further progress this time.

*

****

Notes: In Release 6.0, the function with the largest number of required+optional arguments is TV:DRAW-TRIANGLE-SETUP, which takes 15 arguments.

****

Function Returning

Function Return Instructions

A function returns to its caller by executing one of the return instructions. These instructions specify the value(s) to be returned, remove the returning function's frames from the various stacks, restore the state of the caller, and resume execution of the caller with the returned values on the stack in the form desired by the caller.

The value(s) to be returned can be constant or can be some number of words at the top of the stack; the number of words can be either fixed or variable.

The form of values desired by the caller can be to throw all the values away, to push the first value on the stack, or to push on the stack all the values and a fixnum which is the number of values excluding itself. The caller uses the value-disposition field of the Control register to specify the desired form of values. Note that any form of values supplied to the return instruction can be converted to any form of values desired by the caller. In addition to this format conversion, the return instruction must move the values from one place in the stack to another, from the callee's frame to the caller's frame.

The return instructions are:

user::return-single
Return a single value.

user::return-multiple
Return multiple values (zero or more).

user::return-kludge
Return multiple values in a non-standard form.

user::return-single has an immediate operand that addresses an internal register that supplies the value to be returned. The values that can be returned include **nil**, **t**, and the top-of-stack. **user::return-single** does not do anything that cannot be done with **user::return-multiple** (accompanied by a **push** in some cases), but it is likely that **user::return-single** can be implemented to be much faster than the corresponding **user::return-multiple,** which will speed up important special cases.

user::return-multiple has a standard operand that specifies the number of values to be returned. The values themselves are on the top of the stack. The operand must be a non-negative fixnum. If there is an implementation dependent upper limit on the number of values, it must be at least 16. Although **user::return-multiple** takes a standard operand, only immediate and sp-pop operands are legal. (The reason for this is discussed below.)

user::return-kludge takes the same argument as **user::return-multiple,** but it returns the values in a different way. **user::return-kludge** ignores the value disposition and simply places the values at the top of the caller's stack, without pushing the number of values. **user::return-kludge** also ignores the Cleanup Bits in the Control register. **user::return-kludge** is used for certain internal stack-manipulating subroutines. Note that because **user::return-kludge** does not return values according to the standard calling sequence, it can only be used in subroutines that are specially known by the compiler, and in certain trap handlers.

Note that the description of return values in the instructions above is from the callee's perspective. In other words, this represents what the function would normally return upon completion. The value-disposition field in the Control register, set by the caller, specifies what should actually be done with the return value(s) (that is, they could be discarded).

Before return can remove the frame from the stack, it may have to perform other cleanup actions. These are specified by the Cleanup Bits in the Control register being nonzero. The actions include popping the binding stack, popping the catch stack (a list threaded through the control stack), executing unwind-protect instructions (which may pop the data stack), and escaping to arbitrary software. See the section "Frame Cleanup".

Once these cleanups have been taken care of, the return instruction restores the state of the caller using the information saved in the frame header of the frame being abandoned, according to this procedure:

```
PC                    <=  Continuation register (unless user::vd is return)
Continuation register <=  FP|0
temp                  <=  FP|1
SP                    <=  FP - 1
FP                    <=  FP - cr.frame-size-of-the-caller
Control Register      <=  temp
LP                    <=  FP + cr.argument-size
```

At this point the function's frame has been removed from the control stack. The

stack cache now is either empty or contains part or all of the caller's frame. Since the frame that was just removed from the stack was entirely in the stack cache, the lowest word in the stack cache is less than or equal to SP+1; if equal, the stack cache is empty. The return instruction does not worry about refilling the stack cache at this stage.

The return instruction now places the values being returned at the top of the control stack, according to the value disposition field in the old Control Register and the particular type of return instruction being executed. The **user::return-single** instruction can simply push its argument, but the **user::return-multiple** and **user::return-kludge** instructions may have to transfer a block of values. The source and destination locations of this block can overlap, both in virtual memory and in stack-cache memory, so care must be taken when copying the block of values to its new location.

The specific handling of the value disposition is as follows:

Effect	Leave the stack alone. This leaves the TOS register invalid.
Value	Push the first value being returned onto the stack. If no values were being returned, use **nil** as the first value.
Multiple	Copy the values down from the old top of the stack to the new top of the stack, and form them into a multiple group by appending a count.
Return	Copy the arguments to the Return instruction down to the new top of the stack and then re-execute the instruction. If the instruction was **user::return-multiple** and its operand was sp-pop, the count of values must be pushed back on the stack.

The final thing the return instruction does is to make sure that the frame being returned to is contained in the stack cache. If necessary, words in the frame are fetched from main memory. If a trap or interrupt occurs during this process, PC points at the instruction in the caller being returned to, not at the return instruction, so that the return instruction is not retried (which would return from an extra level of call). When the trap/interrupt handler returns, its return instruction will continue loading the frame into the stack cache. Note that the stack cache must be refilled in *decreasing* order of addresses, so that if a trap occurs the range of addresses validly contained in the stack cache will be contiguous.

When the value disposition is Return, the stack cache is refilled if necessary and then the return instruction is re-executed, causing the value(s) to be returned from the caller. This process can be repeated any number of times.

If the callee returns more values than will fit in the caller's frame, the hardware takes an error trap out of the callee's **return** instruction, before the stack becomes illegal. The checking for oversize stack frame caused by returning too many values can be done at compile-time if the compiler imposes a limit on the maximum number of values returned by any function and the limit is much smaller than the maximum size of a stack frame. Sixteen would be a reasonable limit. The maximum frame size is limited to 256 by the 8-bit field in the Control register [and may be limited to a smaller value by the design of the stack cache].

In order to allow smooth trapping out of the middle of a **return,** it is required that all **return** instructions keep their state, if any, at the top of the stack. This means that we cannot have a **user::return-local** instruction that returns the value

of a local variable; you have to first push the value on the stack and then return it from there with **user::return-single.** Similarly, the number-of-values operand of a **user::return-multiple** instruction cannot be addressed with FP-relative addressing; only immediate and sp-pop operands are allowed. This restriction eliminates any need to play around with special macro-PCs; any trap out of a **return** leaves the PC pointing at the original **return** instruction and the stack set up so that the instruction can be retried.

Returning from a call that had Value-disposition equal to Effect does not restore the TOS register from the top of the stack. This is because there is no time to do it: three reads from the stack cache would be required in this case, whereas when the Value-disposition equals Value two reads from the stack cache plus one write are required and the **user::return-single** instruction executes in only two cycles. This is normally not a problem, since the compiler can compensate, just as it does on the 3600 for other instructions that leave TOS invalid. The compiler simply knows that a finish-call instruction with a value disposition of Effect has the smashes-stack attribute. However, a trap handler for a pre-trap must return with TOS valid, since the trapped instruction might depend on it. It is the responsibility of the person writing such a trap handler to end it by returning the value of the macro **user::trap-handler-restore-tos** which expands into (**user::%p-contents-offset** (**user::%stack-frame-pointer**) -1) or something equivalent. The effect of this is to fetch the proper value from the top of the caller's frame and return it; it will be discarded by the value disposition of Effect, but will cause the proper value to be left in TOS.

****

Notes: Because the handling of Multiple and Return value dispositions is similar, the **user::return-single** and **user::return-multiple** instructions can be implemented by starting with a four-way dispatch to these cases:

1. Cleanup Bits non-zero — Perform the cleanup and then retry the instruction.
2. Value Disposition = Effect — Just return without worrying about the values.
3. Value Disposition = Value — Just return the first value.
4. Value Disposition = Multiple or Return — Take complex actions.

****

Frame Cleanup

The Cleanup Bits in the Control register specify actions necessary before the frame can be exited. Traps, such as page faults, can occur while cleaning up. After handling the trap, the return instruction is retried. The state of the stack while cleaning up is always self-consistent.

The bits and the cleanup actions they cause are as follows, listed in the order that they are processed:

Catch — This bit indicates there are catch/unwind-protect blocks to be unthreaded. Unthreading a block examines the words in the stack addressed by the %catch-block-list register. If the cleanup-handler-executing bit is set, there is an unwind-protect cleanup handler that must be invoked, and the following

actions are performed:

- Restore stack-pointer to its original value, if it was popped by an sp-pop operand.
- If the catch-block-binding-stack-pointer is less than the binding-stack-pointer, unbind special variables until the two pointers are equal. Note that this can clear the Bindings cleanup bit.
- Push the current PC onto the stack.
- Set the PC to the catch-block-PC, which is the address of the cleanup handler.
- Set the cleanup-handler-executing bit in the Control register.
- Set control-register.cleanup-bits.catch in accordance with the cdr code.
- Set the %catch-block-list register to the catch-block-previous, which is the address of the previous catch block or **nil** if there is none.
- Transfer control to the first instruction of the cleanup handler. When the cleanup handler exits the return instruction will be retried.

If the cleanup-handler-executing bit is not set, there is only a catch block to be removed. The following actions are performed:

- Set control-register.cleanup-bits.catch in accordance with the cdr code.
- Set the %catch-block-list register to the catch-block-previous, which is the address of the previous catch block or **nil** if there is none.
- Check the cleanup bits again.

Bindings — This bit indicates there is a non-empty binding-stack frame associated with this control-stack frame, in other words that this function has bound some special variables. Pop the binding stack and undo bindings until a binding stack entry whose binding-stack-chain-bit is zero is encountered. Then clear control-register.cleanup-bits.bindings and check the cleanup bits again.

Trap-on-Exit — Take a trap. If the trap handler clears the Trap-on-Exit bit and returns, the return instruction can proceed.

Value Matchup

When Value-disposition is Multiple, the instruction after a finish-call instruction

will usually be a **user::take-values** instruction. As on the 3600, this converts the multiple group left on the stack by **return** into the desired number of values, popping extra values or pushing **nil** as a default for missing values.

Catch, Throw and Unwind-Protect

A catch block is a sequence of words in the control stack that describes an active catch or unwind-protect operation. All catch blocks in any given stack are linked together, each block containing the address of the next outer block. They are linked in decreasing order of addresses. An internal register named catch-block-pointer contains the address of the innermost catch block, as a dtp-locative, or contains nil if there are no active catch blocks. The address of a catch block is the address of its catch-block-pc word.

The format of a catch block for a catch operation is as follows:

Word Name	*Bit 39*	*Bit 38*	*Contents*
catch-block-tag	0	invalid flag	any object reference
catch-block-pc	0	0	catch exit address
catch-block-binding-stack-pointer	0	0	binding stack level
catch-block-previous	extra-argument	cleanup-catch	previous catch block
catch-block-continuation	value disposition		continuation

The format of a catch block for an unwind-protect operation is as follows:

Word Name	**Bit 39**	**Bit 38**	**Contents**
catch-block-pc	0	0	cleanup handler
catch-block-binding-stack-pointer	0	1	binding stack level
catch-block-previous	extra-argument	cleanup-catch	previous catch block

The *catch-block-tag* word refers to an object that identifies the particular catch operation. The catch-block-invalid-flag bit in this word is initialized to 0, and is set to 1 by the **throw** function when it is no longer valid to throw to this catch block; this addresses a problem with aborting out of the middle of a throw and throwing again. This word is not used by an unwind-protect operation and is only known about by the **throw** function, not by hardware.

The *catch-block-pc* word has data type **user::dtp-even-pc** or **user::dtp-odd-pc.** For a catch operation, it contains the address to which **throw** should transfer control. For an unwind-protect operation, it contains the address of the first instruction of the cleanup handler. The cdr code of this word is set to zero (**user::cdr-next**) and not used. For a catch operation with a value disposition of Return, the catch-block-pc word contains **user::nil.**

The *catch-block-binding-stack-pointer* word contains the value of the binding-stack-pointer hardware register at the time the catch or unwind-protect was established. When undoing the catch or unwind-protect, special-variable bindings are undone until the binding-stack-pointer again has this value. The cdr-code field of this word uses bit 38 to distinguish between catch and unwind-protect; bit 39 is set to zero and not used.

The *catch-block-previous* word contains a dtp-locative pointer to the catch-block-pc word of the previous catch block, or else contains **user::nil.** The cdr-code field of this word saves two bits of the control register that need to be restored.

The *catch-block-continuation* word saves the Continuation hardware register so that

throw can restore it. The cdr-code field of this word saves the value disposition of a catch; this tells the **throw** function where to put the values thrown. This word is not used by unwind-protect.

An unwind-protect cleanup handler terminates with a **user::%jump** instruction. This instruction checks that the data type of the top word on the stack is **user::dtp-even-pc** or **user::dtp-odd-pc,** jumps to that address, and pops the stack. In addition, if the bit 39 of the top word on the stack is 1, it stores bit 38 of that word into control-register.cleanup-in-progress. If bit 39 is 0, it leaves the control register alone.

The compilation of the **catch** special form is approximately as follows:

```
Code to push the catch tag on the stack.
Push a constant PC, the address of the first instruction after the catch.
A user::catch-open instruction.
The body of the catch.
A user::catch-close instruction.
Code to move the values of the body to where they are wanted; this
usually includes removing the 5 words of the catch block from the stack.
```

The compilation of the **unwind-protect** special form is approximately as follows:

```
Push a constant PC, the address of the cleanup handler.
A user::catch-open instruction.
The body of the unwind-protect.
A user::catch-close instruction.
Code to move the values of the body to where they are wanted; this
usually includes removing the 3 words of the catch block from the stack.
```

Somewhere later in the compiled function:

```
The body of the cleanup handler.
A user::%jump instruction.
```

*

****

Notes:

Each active catch or unwind-protect operation has an associated catch-block stored in the control stack and linked onto a list whose root is a processor register, named **user::%catch-block-list,** that is saved in the stack group by context switch.

>>> How is the data stack unwound? But, the idea is that all the frames between the current and the destination of the throw are "unwound" individually, and the data stack is taken care of by this. Each frame that uses the data-stack has an unwind-protect to clean it up. The binding stack is also taken care of by this; the only reason for the binding SP in the catch block is because bindings can happen at any point in the function, and only those that happened after the throw should be undone (the binding stack itself only says with which frame the bindings are associated, not where in the frame).

return from a frame with catch blocks in it automatically removes the catch blocks, via the Cleanup Bits mechanism. What **return** does is similar to the **user::catch-close** instruction.

The implementation of **throw** is somewhat similar to the way it is done on the 3600, but simpler and with less special kludgery. A **throw** special form

```
(throw <tag> <values>)
```

is compiled as

```
(multiple-value-call #'%THROW (VALUES <tag>) <values>)
```

which calls **user::%throw** with the value of <tag> as its first argument and the values of <values> as its remaining arguments. **user::%throw** starts by searching the list of catch blocks for one with the correct tag. If it doesn't find one, or if the catch-block-invalid bit is set in the block it finds, it signals an error. Having located the destination catch block, **user::%throw** prepares to discard all intervening stack frames and catch blocks; this requires invoking any unwind-protect cleanup handlers that are present, each in its proper stack frame and special-variable binding environment. **user::%throw** changes the value disposition of each intervening stack frame to Return, and sets the catch-block-invalid bit in each intervening catch block. Next, **user::%throw** examines the restart PC and value disposition of the destination catch block, and modifies the return PC and value disposition of the next frame in the stack, the one that was called by the frame containing the catch block. There are two cases:

If the catch value disposition is Return, **user::%throw** sets the frame value disposition to Return and returns the values to be thrown. These values are passed back through all the intervening frames, since their value dispositions are Return, and eventually arrive at the desired destination.

Otherwise, **user::%throw** sets the frame value disposition to Multiple, sets the frame return PC to the address of a hand-crafted helping routine, pushes the following values on the stack, and executes a **user::return-multiple** instruction that returns these values through all of the intervening frames. The values pushed are:

- the words to be left in the stack when control reaches the catch's restart PC. This depends on the catch's value disposition and could be nothing, one word, or a multiple group. These are derived from the values to be thrown passed to %THROW as its arguments.
- The catch's restart PC.
- The number of catch blocks to be closed in the destination frame. This is at least 1, and will be more if there are other catches inside the destination catch in the same frame.
- The number of special variable bindings to be undone. This is always zero in this context, but the same helping routine is used for other purposes.
- A count of the total number of values, to make this a valid multiple group.

The hand-crafted helping routine proceeds as follows:

- Loop executing **user::catch-close** instructions the specified number of times.
- Loop executing **user::unbind** instructions the specified number of times.
- Pop the top three words off the stack.
- Do a POPJ instruction, which jumps to the catch's restart PC and leaves the values thrown in the stack.

>>>Note: Think about frame-too-large error in the middle of %THROW.

Note that the return PC and value disposition that need to be modified are actually stored in the frame header of the frame two frames up in the stack from the frame containing the destination catch block. The frame containing the destination catch block could be the same one that called **user::%throw.** In order to avoid having to modify the internal processor registers (Return PC and Control register), **user::%throw** calls itself recursively in this case.

The purpose of the catch-block-invalid bit is to detect the case where a **throw** begins, is interrupted part way through, and the interrupt handler does another **throw** to a catch that is inside the original catch. This can also happen if an unwind-protect cleanup handler gets an error and a **throw** occurs from the Debugger. Since the stack has already been clobbered by changing the value disposition of the frame containing this new catch, the program would operate incorrectly if the second **throw** was permitted to occur. The 3600 deals with this differently; it doesn't modify the value disposition of each frame until it is just about to return from it. This still has a possibility of the same bug, since there could be a catch in the frame being returned from, but the timing window is open for a much smaller time. The 3600's method is more difficult to do on the IMach because of the Control register.

catch-block-invalid catches nonlocal, but lexical, **go**s and **return**s too, since they are compiled as **throw** to a special tag. It does not catch local GOs and **return**s out of unwind-protect cleanup handlers, but those are thoroughly illegal!
****

Generic Functions and Message Passing

The flavor system deals with flavors, instances, instance variables, generic functions, and message passing. A flavor describes the behavior of a family of similar instances. An instance is an object whose behavior is described by a flavor. An instance variable is a variable that has a separate value associated with each instance. A generic function is a function whose implementation dispatches on the flavor of its first argument and selects a method that gets called as the body of the generic function. In message passing, an instance is called as a function; its first argument, known as the message name, is a symbol that is dispatched upon to select a method that gets called. Message passing is the current excuse for generic functions; we plan to phase it out eventually (over a couple of years).

Flavor

A flavor is a structure that contains information shared by all its instances. The header of each instance points into the middle of the structure, at three words known by hardware. Other portions of the flavor are architecturally defined, but not known by hardware. Still other portions of the flavor are known only by the guts of the flavor system.

The data-representation chapter lists the architecturally defined fields of a flavor. See the section "Flavor Instances".

Handler Table

A handler table is a hash table that maps from a generic function or a message to

the method to be invoked and a parameter used by that method to access instance variables. The details concerning the contents of a handler table are presented elsewhere. See the section "Flavor Instances".

The hashing function used to search the handler table is designed to maximize speed and simplify hardware implementation, not to maximize density. It is optimized assuming that the search succeeds on the first or second probe of the hash table. It operates as follows:

- Run the generic function or message name through a hash box to improve its distribution. Bit-reversing and XORing would suffice. The 3600 uses the "identity" hash box that does nothing, and perhaps that is adequate. The hash box computation can be overlapped with memory access delay for the instance header and the two words fetched from the instance descriptor.
- **logand** the result with the hash mask from the flavor.
- Multiply the result by 3 (this is just a shift and an add).
- Add the product to the handler hash table address from the flavor and initiate a block read of sequential locations starting at that address.
- For each block of three words, if the first word does not match the generic function or message name, and is not **nil,** skip the next two words and go on to the next block.
- When a block is found whose key matches or is **nil,** accept the method and the parameter and terminate the search.

Note that when a mismatch occurs, the hash search proceeds through consecutive addresses; it does not rehash. It also does not wrap around when it gets to the end of the table. Consequently the software must allocate sufficient room at the end of the table, after the highest address defined by the hash mask, to accomodate overflow from the end of the table and a final entry with a key of **nil** that is guaranteed to terminate the search.

The hash mask is normally a power of 2 minus 1.

Methods are normally **user::dtp-even-pc** or **user::dtp-odd-pc**. An interpreted method traps to a special entry point to the Lisp interpreter; this is implemented by storing the interpreter (the PC that points to its first instruction) as the method and storing the actual method as the parameter.

*

****

As a special optimization to minimize I-Cache perturbation, a trivial method that simply sets or gets an instance variable is not implemented as a compiled function. Instead, a fixnum is stored in the handler table. The sign of the fixnum is 0 to get or 1 to set; the remaining bits are the offset in the instance of the slot to be accessed. Hardware must verify that the right number of arguments were supplied.
>>>Note: This feature is only an optimization and can be omitted if necessary.
****

Calling a Generic Function

A call to a generic function can be started by **user::dtp-call-generic,**

user::dtp-call-generic-prefetch, user::dtp-call-indirect, user::dtp-call-indirect-prefetch that finds a **user::dtp-generic-function,** or a **user::start-call** instruction whose operand is a **user::dtp-generic-function.** In any case, the generic function is pushed as the extra-argument to the call and the continuation is set to the trap-vector element for calling a **user::dtp-generic-function.** When the call is finished, control transfers to the continuation, which is always a function that consists of nothing but a **user::%generic-dispatch** instruction (there is no entry vector).

The **user::%generic-dispatch** instruction sees the following on the stack:

```
FP|0,1          the usual function-call save area
FP|2            the generic function
FP|3            the instance
FP|4,5,...      additional arguments, if any
```

user::%generic-dispatch operates as follows:

- Make sure that the number of "spread arguments" is at least 2. This ensures that FP|2 and FP|3 are valid. If necessary, perform a pull-lexpr-args operation. If that fails to produce two arguments, signal a "too few arguments" error.

- Get the address of the interesting part of the flavor, which specifies the size and address of the handler hash table. This is done by checking whether the data type of FP|3 is one of the instance data types. If it is, fetch its header following forwarding pointers (header-read). If it is not, use the data type to index a 64-element table in the trap vector that points to the hash-mask fields of the flavor descriptions.

- Fetch two words from the flavor, the hash mask and hash-table address, and perform the handler hash table search described above. If the parameter is not **nil,** store it into FP|2, otherwise leave the generic function in FP|2 (the default handler needs it). If the method is **user::dtp-even user::pc** or **user::dtp-odd-pc,** jump to its entry instruction. If the method is **user::dtp-fixnum,** check the number of arguments, set or get an instance variable, and return from the function call. If the method is anything else, trap (this is an error).

*
****

A reasonable optimization would be to avoid the memory references to fetch the trap-vector element and to fetch the **user::%generic-dispatch** instruction, since calling of generic functions is so common. (It would save 2 memory references out of 5, and avoid perturbing the I cache.) The **user::%generic-dispatch** instruction could be fed magically into the instruction pipeline, and the PC could be set to a constant value that is architecturally required to be the address of a memory location containing a **user::%generic-dispatch** instruction; this location will be referenced if the **user::%generic-dispatch** traps (for example, for a page fault) and has to be retried.

Future hardware might contain a special-purpose cache used th the generic-dispatch instruction to speed repeated lookups with the same generic function and instance.

****

Sending a Message

Sending a message occurs when **user::dtp-call-indirect, user::dtp-call-indirect-prefetch** or a **user::start-call** instruction finds an instance data type as the function. It pushes the instance as the extra-argument to the call and sets the continuation to the trap-vector element for calling that data type. When the call is finished, control transfers to the continuation, which is a function that dispatches to the appropriate method.

At this point, the stack contains the following:

```
FP|0,1          the usual function-call save area
FP|2            the instance
FP|3            the message
FP|4,5,...      additional arguments, if any
```

This is almost like the generic function case except that FP|2 and FP|3 have been exchanged. The distinction between a message and a generic function is unimportant at this level; they are both used only as keys for searching the handler hash table.

The way to perform the dispatch that puts the least burden on the hardware, but makes message passing slower than generic function calling, is for the dispatch function (found in the trap vector) to consist of the following sequence of instructions:

```
ENTRY
PUSH FP|2
PUSH FP|3
POP FP|2
POP FP|3
%GENERIC-DISPATCH
```

The **user::%message-dispatch** instruction, whose description is similar to that of **user::%generic-dispatch** except that the arguments are interchanged, performs the above sequence.

Accessing Instance Variables

Instructions exist to read, write, and locate instance variables.

- Read: fetch the value of the variable, trapping if it is **user::dtp-null,** and push the value on the stack.
- Write: pop a value off the stack and store it into the instance variable, preserving the cdr code of the location and checking for invisible pointers and **user::dtp-monitor-forward** (the same as when writing a special variable).
- Locate: compute the address of the instance variable's value cell and push it on the stack with **user::dtp-locative.** If the value cell contains an invisible pointer, **user::dtp-null,** or **user::dtp-monitor-forward,** that has no effect on the result of this instruction.

These instructions are parameterized by the instance in question and the offset within that instance of the instance-variable slot. There are three groups of instructions:

- Access an arbitrary instance, typified by **user::%instance-ref**: The instruction receives the instance and the offset as ordinary arguments.

- Access **self** unmapped, typified by **user::push-instance-variable-ordered**: The instruction finds the instance in FP|3 (the first argument in the current stack frame, after the extra-argument) and receives the offset as an immediate operand.

- Access **self** mapped, typified by **user::push-instance-variable**: The instruction finds the instance in FP|3 (the first argument in the current stack frame, after the extra-argument), receives an instance variable number as an immediate operand, and finds a mapping table in FP|2 (the extra-argument or "environment"). The mapping table is always a simple, short-prefix ART-Q array. The instance variable number is used as a subscript into the mapping table to get the offset. [Note to those who understand the format of mapping tables used in Release 6 on the 3600: some slots in mapping tables are used for instance variable offsets as described here; other slots are used for other purposes such as subsidiary mapping tables for combined methods. The slots are allocated dynamically by the flavor system as they are required and in general the two types of slots will be interspersed. This eliminates the complexity and slowness of using array-leaders and art-16b arrays.]

If an instance has been **structure-forward**ed to another instance, the value of **self** (FP|3) in a method is the original instance. This means that the instructions to access instance variables must check the header of the instance for a **user::dtp-header-forward,** just as the array referencing instructions do, before adding the offset to the address of the header to get the address of the instance variable.

Stack-Group Switching

The major steps of a stack-group switch are:

1. Inhibit preemption
2. Check the state of the new stack group for resumability
3. Set argument, resumer of new stack group
4. Save internal processor and coprocessor registers
5. Swap out special-variable bindings of the current stack group
6. Make sure the new stack group is prepared for execution
7. Dump the stack cache
8. Switch to the new stack and load the stack cache
9. Restore internal processor and coprocessor registers
10. Swap in special-variable bindings of the new stack group
11. Enable preemption and return

Saving internal processor and coprocessor registers is done by using **user::%read-internal-register** instructions to read the registers into local variables in the stack. When the switch to the new stack group is done, the new current stack frame will be one whose local variables contain the register values for the new stack group.

To restore internal processor and coprocessor registers, use **user::%write-internal-register** instructions to pop the local variables off the stack and put them back in the registers.

Swapping special-variable bindings in and out is the same except that swapping *in* traverses the binding stack in *ascending* address order and swapping *out* traverses it in *descending* address order. All memory reads are done with block-read instructions, since those contain magic bits to select special memory operand reference types.

The basic procedure to swap one binding, assuming that *P* points to a pair of words in the binding stack, is:

```
loc ← data_read(P)                    ;Get address of bound cell
old ← bind_read_no_monitor(P+1)       ;Get old contents of
                                      ;that cell
new ← bind_read_no_monitor(loc)       ;Get new contents of
                                      ;that cell
                                      ;If an invisible pointer is
                                      ;followed, update loc
mem(loc) ← merge_cdr(old,new)         ;Store back old contents
                                      ;preserve cdr
mem(P+1) ← new                        ;Store new contents into
                                      ;binding stack
```

P and *loc* are block-address registers (BARs), *old* and *new* are locations in the stack, *data_read* and *bind_read_no_monitor* are memory read operations described in section "Operand References".

In assembly language, the procedure is as follows. Assume *P* is BAR-1, *loc* is BAR-2, and these BARs can be used for both reading and writing (if only one BAR can write, the procedure includes instructions to copy addresses from one BAR to another). (The order of these instructions might be rearranged to cut down on memory interference and to put the two block-1-reads adjacent, but that is a secondary consideration.)

```
block-1-read                  ;data_read(P)
write-internal-register bar-2 ;loc ←
block-1-read last_word,       ;old ← bind_read_no_monitor(P+1)
  bind_read_no_monitor,no_increment
block-2-read last_word,       ;new ← bind_read_no_monitor(loc)
  bind_read_no_monitor,preserve_cdr,no_increment
merge-cdr-nopop sp|-1         ;cdr(old) ← cdr(new)
block-1-write sp-pop          ;mem(P+1) ← new
block-2-write sp-pop          ;mem(loc) ← old
```

To make sure the new stack group is prepared for execution, it is necessary to call a subroutine in the paging system to wire down appropriate pages of the stack, and to run the GC scavenger over those pages if necessary. This also determines the appropriate values for the stack limit registers. The paging system maintains enough state so that this operation is very fast if the stack group has been run

recently. Doing this before actually switching to that stack ensures that no traps (page faults or transport traps) can occur during the actual act of switching, when things are inconsistent, and ensures that the new stack group has enough space for the extra-stack.

To dump the stack cache, use a loop that does block-read and block-write at identical addresses. This is based on the assumption that writes to memory locations in the stack cache write through to main memory.

To switch to the new stack and load the stack cache, initialize the registers that control the stack cache to suitable values and then do block-reads to fill it. In detail:

1. Save the SP into the current stack group.

2. Get the SP value of the new stack group. The FP value is at a known offset from this. These bracket a stack frame which is in the same format as the current stack frame, but contains the register values of the other stack group.

3. Go into extra-stack mode so no traps/interrupts can occur.

4. Store the FP value into the hardware FP and into a BAR.

5. Set the stack cache lower bound register to the SP value +1, so that the following block-reads will neither read from the stack cache nor cause it to overflow.

6. Store the FP value minus 1 into the hardware SP. Do this last, since it renders the old stack frame inaccessible.

7. Execute a sequence of block reads that fetch the new stack frame into the stack cache and increment the SP to its appropriate value.

8. Set the stack cache lower bound register to FP. The stack cache is now consistent.

9. Set the stack limit registers to the values for the new stack group.

Restoring the internal processor registers will turn off extra-stack mode by restoring the control register. The order of restoring of these registers needs to be chosen with some care so that a sequence break immediately after extra-stack mode is turned off will work correctly.

*

****

Existing instructions have the following capabilities: - ability to do appropriate special memory references, using block-read/write - ability to do necessary cdr-code hacking - ability to dump the entire stack cache into memory - ability to load a new stack into an empty stack cache - ability to read and write all internal processor and coprocessor registers that are part of the stack group context - ability to inhibit all traps and interrupts while the stack cache control registers are in an inconsistent state - ability to inhibit process preemption during the whole operation this is done by setting a software flag respected by the preempt

Other instruction assumptions: bind_read_no_monitor bit in block-read instruction no_increment bit in block-read instruction prevents incrementing BAR preserve_cdr

bit in block-read instruction inhibits setting cdr of result to 0 (this is already in the rev -2 spec) when block-read follows an invisible pointer, it updates the BAR merge-cdr-nopop instruction: cdr(operand) ← cdr(top-of-stack), no change to SP this could be done with **user::%p-tag-ldb** and **user::%p-tag-dpb** but it would be much slower.

Note that it is possible for the cdr code of the bound location to change while it is bound, which is why the merge-cdr-nopop instruction is required instead of simply rewriting all 40 bits with the value saved in the binding stack.

Alternatively, to the assumption that memory locations in the stack write through to main memory, a specific instruction could be provided to dump the entire stack cache, since the processor already knows how to dump parts of the stack cache when it fills up.

****