Low-Level Program Verification using Matching Logic Reachability

Dwight Guth, Andrei Ştefănescu, and Grigore Roşu
University of Illinois at Urbana-Champaign

MODULE WCET-SYNTAX

SYNTAX *BInst* ::= *BOpCode* *Register*, *Exp*, *Exp* [klabel('binst), strict(3, 4)]

SYNTAX *BOpCode* ::= `add`
| `sub`
| `mul`
| `div`
| `or`
| `and`

SYNTAX *Exp* ::= *Register*
| # *Int*

SYNTAX *Register* ::= `r` *Int*

These productions define the basic pieces of B-Type instructions in the language. The language is a 3-register language, so each instruction has two sources and one target.

The language has two indexing modes: an integer may signify the contents of a numbered register, or it may be an immediate value. We use strictness to evaluate the arguments of an instruction to their final value.

SYNTAX *UInst* ::= *UOpCode* *Register*, *Exp* [strict(3), klabel('uinst)]

SYNTAX *UOpCode* ::= `not`

Here we define the first U-Type instruction: not. Instead of taking two sources and one target, it takes one source and one target.

SYNTAX *MInst* ::= *MOpCode* *Exp*, *Exp* [klabel('minst), strict(2, 3)]

SYNTAX *UOpCode* ::= `load`

SYNTAX *MOpCode* ::= `store`

In addition to basic arithmetic and bitwise-logical operations, our language supports load and store operations to index memory dynamically. Load evaluates a memory location into a register; store evaluates a memory location and a value and puts the value into memory.

SYNTAX *JInst* ::= *JOpCode* *Id* [klabel('jinst)]

SYNTAX *JOpCode* ::= `jmp`

The `jmp` instruction takes a labelled code block and unconditionally jumps to that location.

SYNTAX *BrInst* ::= *BrOpCode* *Id*, *Exp*, *Exp* [klabel('brinst), strict(3, 4)]

SYNTAX *BrOpCode* ::= `beq`
| `bne`
| `blt`
| `ble`

The language also supports conditional jump instructions. Each of these instructions evaluates two integers and either jumps or does not jump based on the result of an arithmetic comparison.

SYNTAX *NInst* ::= *NOpCode*

SYNTAX *NOpCode* ::= `halt`

The halt instruction terminates code execution immediately, no matter what instructions follow it.

SYNTAX *SInst* ::= *SOpCode* *Exp* [klabel('sinst), strict(2)]

SYNTAX *SOpCode* ::= `sleep`

The sleep instruction takes an argument specifying a length of time in cycles, and sleeps the processor until the time has elapsed.

SYNTAX *RInst* ::= *ROpCode* *Register*, *Id* [klabel('rinst)]

SYNTAX *ROpCode* ::= `read`

SYNTAX *WInst* ::= *WOpCode* *Id*, *Exp* [strict(3), klabel('winst)]

SYNTAX *WOpCode* ::= `write`

SYNTAX *RWInst* ::= *RWOpCode* *Register*, *Id*, *Exp* [strict(4), klabel('rwinst)]

SYNTAX *RWOpCode* ::= `rw`

These three instructions perform I/O operations on an asynchronous I/O buffer. This named buffer can be written to at any time by the environment the program is running in (simulated as a configuration parameter). The read instruction reads the value of the buffer, the write instruction writes the value of the buffer, and the rw instruction atomically first reads, then writes to the buffer. The purpose of the rw instruction is to prevent a race that can occur if the program reads a value, then resets the value of the buffer, but another value is written in between.

SYNTAX *BrOpCode* ::= `int`

SYNTAX *NOpCode* ::= `rfi`

Finally, we support periodic timer interrupts. The `int` instruction schedules an interrupt, which fires periodically until the program is terminated. The `rfi` instruction returns from an interrupt.

SYNTAX *UOpCode* ::= `li`

SYNTAX *BrOpCode* ::= `bgt`
| `bge`

RULE $\dfrac{\texttt{li}\ \ R{:}Register, E{:}Exp}{\texttt{or}\ \ R, E, \texttt{\#}\,0}$ [macro]

RULE $\dfrac{\texttt{bgt}\ \ X{:}Id, E{:}Exp, E2{:}Exp}{\texttt{blt}\ \ X, E2, E}$ [macro]

RULE $\dfrac{\texttt{bge}\ \ X{:}Id, E{:}Exp, E2{:}Exp}{\texttt{ble}\ \ X, E2, E}$ [macro]

We also define a couple convenience instructions as macros. The li instruction loads either an immediate or the contents of a register into anothe register. The bgt and bge instructions complete our set of conditional jump instructions.

SYNTAX *Id* ::= `main` [token]

"main" is a special program label denoting the entry point of the program.

SYNTAX *Insts* ::= *Inst*
| *Inst* *Insts*

SYNTAX *Block* ::= *Id* `:` *Insts*

SYNTAX *Blocks* ::= *Block*
| *Block* *Blocks*

A program consists of a list of labelled code blocks each containing a sequence of instructions.

SYNTAX *Inst* ::= *BInst*
| *UInst*
| *MInst*
| *JInst*
| *BrInst*
| *NInst*
| *SInst*
| *RInst*
| *WInst*
| *RWInst*

SYNTAX *OpCode* ::= *BOpCode*
| *UOpCode*
| *MOpCode*
| *JOpCode*
| *BrOpCode*
| *NOpCode*
| *SOpCode*
| *ROpCode*
| *WOpCode*
| *RWOpCode*

END MODULE

MODULE WCET

CONFIGURATION:

$$\Big\langle \langle \texttt{load}\,(\$PGM{:}K) \curvearrowright \texttt{jumpTo}\,(\texttt{main})\rangle_{\mathsf{k}}\ \langle \bullet_{Map}\rangle_{\mathsf{pgm}}\ \langle \bullet_{Map}\rangle_{\mathsf{mem}}\ \langle \bullet_{Map}\rangle_{\mathsf{reg}}\ \langle \$TIMING{:}Map\rangle_{\mathsf{timing}}\ \langle 0\rangle_{\mathsf{wcet}}\ \langle \$INPUT{:}List\rangle_{\mathsf{input}}\ \langle \$INITIAL{:}Map\rangle_{\mathsf{status}}\ \langle \bullet_{List}\rangle_{\mathsf{timers}}\ \langle 0\rangle_{\mathsf{priority}}\ \langle \bullet_{List}\rangle_{\mathsf{stack}}\ \langle \bullet_{Set}\rangle_{\mathsf{interrupts}} \Big\rangle_{\mathsf{T}}$$

This configuration declaration specifies the initial configuration of the program by means of several variables that are initialized by the tool after being specified by the user on the command line. This initial configuration performs preprocessing on the program to load it into the `pgm` cell and then jumps to the entry point, labelled `main`.

SYNTAX *KResult* ::= *Int*

SYNTAX *Exp* ::= *Int*

The values of the language are integers once an immediate or register index is evaluated.

SYNTAX *KItem* ::= `load` (*K*) [klabel('load)]
| `jumpTo` (*Id*) [klabel('jumpTo)]

Here we define additional auxiliary operators. load takes a program and puts it in the `pgm` cell for use by jump instructions later on. jumpTo takes an identifier and jumps immediately to it. We define this instruction because we want to use it in places where using the `jmp` instruction directly in our semantics is unsuitable because the `jmp` instruction increments the `wcet` cell.

RULE $\left\langle \dfrac{I{:}Inst\ \ Is{:}Insts}{I \curvearrowright Is} \cdots \right\rangle_{\mathsf{k}}$

Here we define sequential composition. A block of instructions is executed one at a time.

RULE $\dfrac{\texttt{load}\,(B{:}Block\ \ Bs{:}Blocks)}{\texttt{load}\,(B) \curvearrowright \texttt{load}\,(Bs)}$

RULE $\left\langle \dfrac{\texttt{load}\,(X{:}Id\ \texttt{:}\ Is{:}Insts)}{\bullet_{K}} \cdots \right\rangle_{\mathsf{k}}\ \left\langle \cdots \dfrac{\bullet_{Map}}{X \mapsto (Is)} \cdots \right\rangle_{\mathsf{pgm}}$

Note here that the `pgm` cell is only loaded with the contents of the current block. We will see the significance of this momentarily.

RULE $\left\langle \dfrac{\texttt{jumpTo}\,(X) \curvearrowright \text{—}}{K} \right\rangle_{\mathsf{k}}\ \langle \cdots X{:}Id \mapsto K{:}K \cdots \rangle_{\mathsf{pgm}}$

Here we take the contents of the `pgm` cell and replace the entire `k` cell with it. Because the pgm cell does not include all subseqent blocks, a program does not fall through from one block to the next. In order to continue executing, the block must end with a jump instruction to specify the next block.

RULE $\left\langle \dfrac{\texttt{r}\ I{:}Int}{I2{:}Int} \cdots \right\rangle_{\mathsf{k}}\ \langle \cdots I \mapsto I2 \cdots \rangle_{\mathsf{reg}}$

Register lookup looks up the numbered register in the `reg` cell.

RULE $\dfrac{\texttt{\#}\ I{:}Int}{I}$

And an immediate evaluates to its own declared value.

RULE $\left\langle \dfrac{\texttt{add}\ \ \texttt{r}\ I{:}Int, I2{:}Int, I3{:}Int}{\texttt{time}\,(\,\texttt{add})} \cdots \right\rangle_{\mathsf{k}}\ \left\langle \dfrac{R{:}Map}{R[I2 +_{Int} I3 \texttt{ <- } I]} \right\rangle_{\mathsf{reg}}$

RULE $\left\langle \dfrac{\texttt{sub}\ \ \texttt{r}\ I{:}Int, I2{:}Int, I3{:}Int}{\texttt{time}\,(\,\texttt{sub})} \cdots \right\rangle_{\mathsf{k}}\ \left\langle \dfrac{R{:}Map}{R[I2 -_{Int} I3 \texttt{ <- } I]} \right\rangle_{\mathsf{reg}}$

RULE $\left\langle \dfrac{\texttt{mul}\ \ \texttt{r}\ I{:}Int, I2{:}Int, I3{:}Int}{\texttt{time}\,(\,\texttt{mul})} \cdots \right\rangle_{\mathsf{k}}\ \left\langle \dfrac{R{:}Map}{R[I2 *_{Int} I3 \texttt{ <- } I]} \right\rangle_{\mathsf{reg}}$

RULE $\left\langle \dfrac{\texttt{div}\ \ \texttt{r}\ I{:}Int, I2{:}Int, I3{:}Int}{\texttt{time}\,(\,\texttt{div})} \cdots \right\rangle_{\mathsf{k}}\ \left\langle \dfrac{R{:}Map}{R[I2 \div_{Int} I3 \texttt{ <- } I]} \right\rangle_{\mathsf{reg}}$

RULE $\left\langle \dfrac{\texttt{or}\ \ \texttt{r}\ I{:}Int, I2{:}Int, I3{:}Int}{\texttt{time}\,(\,\texttt{or})} \cdots \right\rangle_{\mathsf{k}}\ \left\langle \dfrac{R{:}Map}{R[I2 \mid_{Int} I3 \texttt{ <- } I]} \right\rangle_{\mathsf{reg}}$

RULE $\left\langle \dfrac{\texttt{and}\ \ \texttt{r}\ I{:}Int, I2{:}Int, I3{:}Int}{\texttt{time}\,(\,\texttt{and})} \cdots \right\rangle_{\mathsf{k}}\ \left\langle \dfrac{R{:}Map}{R[I2\ \&_{Int}\ I3 \texttt{ <- } I]} \right\rangle_{\mathsf{reg}}$

RULE $\left\langle \dfrac{\texttt{not}\ \ \texttt{r}\ I{:}Int, I2{:}Int}{\texttt{time}\,(\,\texttt{not})} \cdots \right\rangle_{\mathsf{k}}\ \left\langle \dfrac{R{:}Map}{R[\sim_{Int} I2 \texttt{ <- } I]} \right\rangle_{\mathsf{reg}}$

The first instructions of the language are defined above. Once the arguments are evaluated, the arithmetic operation is performed and the result is stored in the target register. Then the time of the instruction is made to elapse. The auxiliary operation `time` elapses time according to the specified length of the instruction in question.

RULE $\left\langle \dfrac{\texttt{load}\ \ \texttt{r}\ I{:}Int, I2{:}Int}{\texttt{time}\,(\,\texttt{load})} \cdots \right\rangle_{\mathsf{k}}\ \langle \cdots I2 \mapsto I3{:}Int \cdots \rangle_{\mathsf{mem}}\ \left\langle \dfrac{R{:}Map}{R[I3 \texttt{ <- } I]} \right\rangle_{\mathsf{reg}}$

The `mem` cell contains memory. An integer location in memory is evaluated, indexed in memory, and the value found there is put in a register.

RULE $\left\langle \dfrac{\texttt{store}\ \ I{:}Int, I2{:}Int}{\texttt{time}\,(\,\texttt{store})} \cdots \right\rangle_{\mathsf{k}}\ \left\langle \dfrac{M{:}Map}{M[I2 \texttt{ <- } I]} \right\rangle_{\mathsf{mem}}$

`store` evaluates two arguments. The first expresses the location in memory to write to, and the second expresses the value to write there. Once evaluated, we update the `mem` cell with the value in memory.

RULE $\dfrac{\texttt{jmp}\ \ X{:}Id}{\texttt{time}\,(\,\texttt{jmp}) \curvearrowright \texttt{jumpTo}\,(X)}$

The main semantics of `jmp` are the jumpTo opereration. The instruction takes time, however.

RULE $\left\langle \dfrac{\texttt{beq}\ \ X{:}Id, I{:}Int, I2{:}Int}{\texttt{time}\,(\,\texttt{beq}) \curvearrowright \texttt{branch}\,(I ==_{Int} I2, X)} \cdots \right\rangle_{\mathsf{k}}$

RULE $\left\langle \dfrac{\texttt{bne}\ \ X{:}Id, I{:}Int, I2{:}Int}{\texttt{time}\,(\,\texttt{bne}) \curvearrowright \texttt{branch}\,(I =/=_{Int} I2, X)} \cdots \right\rangle_{\mathsf{k}}$

RULE $\left\langle \dfrac{\texttt{blt}\ \ X{:}Id, I{:}Int, I2{:}Int}{\texttt{time}\,(\,\texttt{blt}) \curvearrowright \texttt{branch}\,(I <_{Int} I2, X)} \cdots \right\rangle_{\mathsf{k}}$

RULE $\left\langle \dfrac{\texttt{ble}\ \ X{:}Id, I{:}Int, I2{:}Int}{\texttt{time}\,(\,\texttt{ble}) \curvearrowright \texttt{branch}\,(I \le_{Int} I2, X)} \cdots \right\rangle_{\mathsf{k}}$

SYNTAX *KItem* ::= `branch` (*Bool*, *Id*) [klabel('branch)]

RULE $\dfrac{\texttt{branch}\,(\mathsf{true}, X{:}Id)}{\texttt{jumpTo}\,(X)}$

RULE $\dfrac{\texttt{branch}\,(\mathsf{false}, \text{—})}{\bullet_{K}}$

Conditional jump evaluates a boolean condition. If the condition is true, we jump to the target. If it is false, we fall through to the next instruction.

RULE $\dfrac{\texttt{halt} \curvearrowright \text{—}}{\texttt{time}\,(\,\texttt{halt})}$

`halt` removes the entire `k` cell, effectively ending execution (once it is finished executing).

RULE $\dfrac{\texttt{sleep}\ \ I{:}Int}{\texttt{waitFor}\,(I)}$

The auxiliary operation `waitFor` elapses a specified length of time.

RULE $\left\langle \dfrac{\texttt{read}\ \ \texttt{r}\ I{:}Int, X{:}Id}{\texttt{time}\,(\,\texttt{read})} \cdots \right\rangle_{\mathsf{k}}\ \langle \cdots X \mapsto I2{:}Int \cdots \rangle_{\mathsf{status}}\ \left\langle \dfrac{Reg{:}Map}{Reg[I2 \texttt{ <- } I]} \right\rangle_{\mathsf{reg}}$

An I/O read looks up the current value of the data in the `status` cell, then writes it to a register.

RULE $\left\langle \dfrac{\texttt{write}\ \ X{:}Id, I{:}Int}{\texttt{time}\,(\,\texttt{write})} \cdots \right\rangle_{\mathsf{k}}\ \left\langle \dfrac{Status{:}Map}{Status[I \texttt{ <- } X]} \right\rangle_{\mathsf{status}}$

`write` evaluates a value, then writes it to the `status` cell.

RULE $\left\langle \dfrac{\texttt{rw}\ \ \texttt{r}\ I{:}Int, X{:}Id, I3{:}Int}{\texttt{time}\,(\,\texttt{rw})} \cdots \right\rangle_{\mathsf{k}}\ \left\langle \cdots X \mapsto \dfrac{I2{:}Int}{I3} \cdots \right\rangle_{\mathsf{status}}\ \left\langle \dfrac{Reg{:}Map}{Reg[I2 \texttt{ <- } I]} \right\rangle_{\mathsf{reg}}$

`rw` simply combines these two instructions together.

SYNTAX *KItem* ::= (*Id*, *Int*, *Int*)

RULE $\left\langle \dfrac{\texttt{int}\ \ X{:}Id, I{:}Int, I2{:}Int}{\texttt{time}\,(\,\texttt{int})} \cdots \right\rangle_{\mathsf{k}}\ \left\langle \cdots \dfrac{\bullet_{List}}{\texttt{ListItem}\,((X, I +_{Int} Time, I2))} \right\rangle_{\mathsf{timers}}\ \langle Time{:}Int \rangle_{\mathsf{wcet}}$

`int` schedules an interrupt I cycles after executing, to execute periodically every $I2$ cycles thereafter. The `timers` cell stores the currently activated interrupts in a list of tuples.

SYNTAX *KItem* ::= (*K*, *Int*)

RULE $\left\langle \dfrac{\texttt{rfi} \curvearrowright \text{—}}{\texttt{time}\,(\,\texttt{rfi}) \curvearrowright K} \right\rangle_{\mathsf{k}}\ \left\langle \dfrac{\texttt{ListItem}\,((K{:}K, Priority{:}Int))}{\bullet_{List}} \cdots \right\rangle_{\mathsf{stack}}\ \left\langle \dfrac{\text{—}}{Priority} \right\rangle_{\mathsf{priority}}$

To return from an interrupt, we restore the previously executing code from the `stack` cell, which also contains the previously-executing priority to restore to the `priority` cell. Interrupts are assigned numeric priority in the order they are scheduled by the program, and can interrupt only code running at a lower priority. Recall the program begins executing at priority 0.

SYNTAX *KItem* ::= `time` (*OpCode*) [klabel('time)]

RULE $\left\langle \dfrac{\texttt{time}\,(O{:}OpCode)}{\texttt{waitFor}\,(I)} \cdots \right\rangle_{\mathsf{k}}\ \langle \cdots O \mapsto I{:}Int \cdots \rangle_{\mathsf{timing}}$

For modularity, we allow the time each instruction executes for to be specified dynamically in the `timing` cell.

SYNTAX *KItem* ::= `waitFor` (*Int*) [klabel('waitFor)]

RULE $\left\langle \dfrac{\texttt{waitFor}\,(I{:}Int)}{\texttt{updateStatus}\,(I2) \curvearrowright \texttt{updateTimers}\,(L) \curvearrowright \texttt{interrupt}\,(L,\ \texttt{size}\,(L))} \cdots \right\rangle_{\mathsf{k}}\ \left\langle \dfrac{I2{:}Int}{I2 +_{Int} I} \right\rangle_{\mathsf{wcet}}\ \left\langle \dfrac{L{:}List}{\bullet_{List}} \right\rangle_{\mathsf{timers}}$

The `wcet` cell stores the length of time the program has already been running for, in total. When time passes, two types of asynchronous events can occur. The first is a write on an I/O line, provided in advance in the `input` cell. The second is the firing of a timer interrupt.

SYNTAX *KItem* ::= `updateStatus` (*Int*) [klabel('updateStatus)]

SYNTAX *KItem* ::= (*Int*, *Map*) [klabel('status)]

RULE $\langle \texttt{updateStatus}\,(Start{:}Int) \cdots \rangle_{\mathsf{k}}\ \left\langle \dfrac{\texttt{ListItem}\,((I2{:}Int, M{:}Map))}{\bullet_{List}} \cdots \right\rangle_{\mathsf{input}}\ \langle I{:}Int \rangle_{\mathsf{wcet}}\ \left\langle \dfrac{Status{:}Map}{\texttt{updateMap}\,(Status, M)} \right\rangle_{\mathsf{status}}$ requires $I \geq_{Int} I2 \wedge_{Bool} I2 \geq_{Int} Start$

RULE $\left\langle \dfrac{\texttt{updateStatus}\,(Start{:}Int)}{\bullet_{K}} \cdots \right\rangle_{\mathsf{k}}\ \langle \texttt{ListItem}\,((I2{:}Int, \text{—}{:}\mathrm{Map})) \cdots \rangle_{\mathsf{input}}\ \langle I{:}Int \rangle_{\mathsf{wcet}}$ requires $\neg_{Bool}(I \geq_{Int} I2 \wedge_{Bool} I \geq_{Int} Start)$

RULE $\left\langle \dfrac{\texttt{updateStatus}\,(\text{—})}{\bullet_{K}} \cdots \right\rangle_{\mathsf{k}}\ \langle \bullet_{List} \rangle_{\mathsf{input}}$

Essentially, updateStatus processes every input event between time $Start$ and time I. Each event contains a map of associated I/O writes, which is merged with the status cell. Events ocurring before or after the time window of the instruction are ignored.

SYNTAX *KItem* ::= `updateTimers` (*List*) [klabel('updateTimers)]

RULE $\left\langle \texttt{updateTimers}\,\Big(\dfrac{\texttt{ListItem}\,((X{:}Id, Expires{:}Int, Interval{:}Int))}{\bullet_{List}}\ \text{—}\Big) \cdots \right\rangle_{\mathsf{k}}\ \langle I{:}Int \rangle_{\mathsf{wcet}}\ \left\langle \cdots \dfrac{\bullet_{List}}{\texttt{ListItem}\,((X, Expires, Interval))} \right\rangle_{\mathsf{timers}}$ requires $I <_{Int} Expires \vee_{Bool} Interval ==_{Int} 0$

RULE $\left\langle \texttt{updateTimers}\,\Big(\texttt{ListItem}\,\Big(\Big(X{:}Id, \dfrac{Expires{:}Int}{Expires +_{Int} Interval}, Interval{:}Int\Big)\Big)\ \text{—}\Big) \cdots \right\rangle_{\mathsf{k}}\ \langle I{:}Int \rangle_{\mathsf{wcet}}\ \left\langle \cdots \dfrac{\bullet_{Set}}{\texttt{SetItem}\,(X)} \right\rangle_{\mathsf{interrupts}}$ requires $I \geq_{Int} Expires \wedge_{Bool} Interval =/=_{Int} 0$

RULE $\dfrac{\texttt{updateTimers}\,(\bullet_{List})}{\bullet_{K}}$

Essentially, updateTimers updates the `interrupts` cell with all the interrupts that have been triggered by the previous instruction. When an interrupt is triggered, the tuple containing info on the interrupt is updated with a new expiration time.

SYNTAX *KItem* ::= `interrupt` (*List*, *Int*) [klabel('interrupt)]

RULE $\left\langle \texttt{interrupt}\,\Big(\dfrac{\texttt{ListItem}\,((X{:}Id, \text{—}, \text{—}))}{\bullet_{List}}\ \text{—}, \dfrac{N{:}Int}{N -_{Int} 1}\Big) \cdots \right\rangle_{\mathsf{k}}\ \langle S{:}Set \rangle_{\mathsf{interrupts}}\ \langle Priority{:}Int \rangle_{\mathsf{priority}}$ requires $\neg_{Bool}(X\ \texttt{in}\ S) \vee_{Bool} N \leq_{Int} Priority$

RULE $\left\langle \dfrac{\texttt{interrupt}\,(\texttt{ListItem}\,((X{:}Id, \text{—}, \text{—}))\ \text{—}, N{:}Int) \curvearrowright K{:}K}{\texttt{jumpTo}\,(X)} \right\rangle_{\mathsf{k}}\ \left\langle \dfrac{Priority{:}Int}{N} \right\rangle_{\mathsf{priority}}\ \left\langle \dfrac{S{:}Set}{S -_{Set} \texttt{SetItem}\,(X)} \right\rangle_{\mathsf{interrupts}}\ \left\langle \dfrac{\bullet_{List}}{\texttt{ListItem}\,((K, Priority))} \cdots \right\rangle_{\mathsf{stack}}$ requires $N >_{Int} Priority \wedge_{Bool} X\ \texttt{in}\ S$

RULE $\left\langle \dfrac{\texttt{interrupt}\,(\bullet_{List}, \text{—})}{\bullet_{K}} \cdots \right\rangle_{\mathsf{k}}$

Essentially, interrupt fires exactly one pending interrupt, removing it from the `interrupts` cell. This only occurs if there is a pending interrupt to fire and its priority is higher than the current priority of the process, however. When an interrupt is fired, we jump to the label specified by the `int` instruction and begin executing with the new priority. Note that the priority of an interrupt is simply the number of interrupts that were scheduled after the interrupt in question. Thus if we schedule 5 interrupts, the first has priority 5, the second has priority 4, etc. User code has priority 0. Once an interrupt reaches an `rfi` instruction, this process will repeat again and another interrupt will fire. Only once all interrupts have been fired will normal code resume. Thus, like a real assembly language, we can starve low-priority code by firing too many high-priority interrupts.

SYNTAX *Id* ::=

END MODULE