



AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

KATEDRA INFORMATYKI STOSOWANEJ

Praca dyplomowa magisterska

*Realizacja podstawowej funkcjonalności maszyny wirtualnej Erlanga
dla systemu FreeRTOS*

*Implementation of basic features of Erlang Virtual Machine for
FreeRTOS*

Autor: *Rafał Studnicki*
Kierunek studiów: *Informatyka*
Opiekun pracy: *dr inż. Piotr Matyasik*

Kraków, 2014

Oświadczam, świadomy(-a) odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Spis treści

1. Zaimplementowane elementy maszyny wirtualnej	7
1.1. Moduł ładujący kod (<i>loader</i>)	7
1.2. Tablice	8
1.2.1. Tablica atomów	10
1.2.2. Tablica eksportowanych funkcji	10
1.3. Typy danych	11
1.3.1. Wartości bezpośrednie a pośrednie	11
1.3.2. Wartości bezpośrednie	13
1.3.3. Listy	13
1.3.4. Krotki	14
1.3.5. Duże liczby	15
1.3.6. Niezaimplementowane typy danych	16
1.4. Interpreter kodu maszynowego	17
1.4.1. Maszyna stosowa a rejestrowa	17
1.4.2. Model interpretera	18
1.5. Procesy	19
1.6. Planista (<i>scheduler</i>)	19
1.7. Funkcje wbudowane (<i>Built-In Functions</i>)	19
1.7.1. Funkcje ogólne (moduł <code>erlang</code>)	19
1.7.2. Listy (moduł <code>lists</code>)	19
1.7.3. GPIO i obsługa przerwań (moduł <code>lpc_gpio</code>)	19
1.7.4. SPI (moduł <code>lpc_spi</code>)	19
1.8. <i>Garbage collector</i>	19
1.9. Mechanizmy zarządzania czasem	19
1.10. Podsumowanie różnic z maszyną BEAM	19
A. Lista instrukcji maszyny wirtualnej BEAM	23
A.1. Typy argumentów	23

A.2. Lista instrukcji	25
Bibliografia	41

1. Zaimplementowane elementy maszyny wirtualnej

Maszyna wirtualna jest warstwą abstrakcji uruchamianą pod kontrolą pewnego systemu operacyjnego. Powinna ona emulować fizyczny procesor w taki sam sposób niezależnie od systemu operacyjnego czy fizycznej architektury, na jakiej została uruchomiona. Dzięki temu możliwe jest uruchomienie tego samego kodu, nazywanego kodem pośrednim, przystosowanego do architektury maszyny wirtualnej, pod kontrolą różnych systemów operacyjnych i na różnych architekturach sprzętowych.

Wśród funkcjonalności wirtualnego procesa, które powinna implementować maszyna wirtualna dowolnego języka można wymienić:

- struktury danych, przy pomocy których opisane są instrukcje i ich argumenty;
- stos, wykorzystywany do wywołań funkcji;
- wskaźnik kolejnej instrukcji do wykonania;
- interpreter kodu pośredniego, pobierający kolejną instrukcję do wykonania, dekodujący jej argumenty i wykonujący ją.

W rozdziale wymieniono elementy maszyny wirtualnej Erlanga, które zaimplementowane ramach w pracy, tak aby powyższy zbiór funkcjonalności zapewniał możliwość wykonywania kodu modułów skompilowanych przy użyciu kompilatora Erlanga w wersji R16. Opis poszczególnych elementów zawiera wyjaśnienie ich sposobu działania, ich roli w maszynie wirtualnej, a także porównania funkcjonalności do maszyny wirtualnej BEAM.

Wszystkie opisywane funkcjonalności zostały zaimplementowane w języku C, podobnie jak jest to w przypadku maszyny wirtualnej BEAM.

1.1. Moduł ładujący kod (*loader*)

Moduł opisany w niniejszym podrozdziale został zaimplementowany w pliku źródłowym `beam_load.c`.

Podstawowym zadaniem *loadera* jest wykonanie zestawu operacji, po których będzie możliwe wykonanie kodu programu, zawartego w pliku będącym efektem kompilacji modułu w języku Erlang, z poziomu interpretera kodu pośredniego w maszynie wirtualnej. W maszynie BEAM źródłem plików binarnych może być system plików systemu operacyjnego lub inny węzeł Erlanga znajdujący się w tym

samym klastrze. Aby pliki te mogły być wykonywane na maszynie zaimplementowanej w ramach pracy, która nie obsługuje ani systemu plików ani protokołu klastrowania, muszą one ulec przetworzeniu i zostać wkompiłowane w kod maszyny wirtualnej. Dokonywane jest to za pomocą narzędzia opisanego w dodatku ??.

Zawartość pliku, który poddawany jest przetwarzaniu w module została szczegółowo opisana w dodatku ??, który ze względu na brak oficjalnej dokumentacji powstał na potrzeby niniejszej pracy.

Aby możliwe było wykonywanie kodu pośredniego przez maszynę wirtualną, konieczne jest wykonanie następujących kroków:

- załadowanie lokalnej tablicy atomów (fragment `Atom`) do globalnej tablicy atomów (por. 1.2.1);
- załadowanie lokalnej tablicy eksportowanych funkcji (fragment `ExpT`) do globalnej tablicy (por. 1.2.2);
- sparsowanie wyrażeń w postaci *External Term Format*, umieszczonych we fragmencie `LitT` i umieszczenie ich w pamięci o globalnym dostępie (globalnej stercie);
- wyszukanie w globalnej tablicy eksportów funkcji zewnętrznych używanych przez moduł (fragment `ImpT`);
- podstawienie wyrażeń rozpoznawanych globalnie (por. 1.3) za wyrażenia lokalne, opisane w podrozdziale A.1, we fragmencie `Code`. Do rozważanych wyrażeń należą: atomy, etykiety lokalnych funkcji, odnośniki do funkcji znajdujących się w innych modułach oraz wyrażenia umieszczone na globalnej stercie;
- podstawienie za numery operacji z sekcji `Code`, opisane w podrozdziale A.2), wskaźników do odpowiedniej sekcji interpretera kodu maszynowego wykonującemu daną instrukcję (por. 1.4).

Loader maszyny wirtualnej BEAM wykonuje jeszcze jeden bardzo istotny krok, którego nie wykonuje maszyna zaimplementowana w pracy. Jest nim zastosowanie gramatyki modyfikującej w bardzo istotny sposób kod maszynowy zawarty w pliku binarnym. Gramatykę tę można znaleźć w pliku `ops.tab` w kodzie źródłowym maszyny BEAM. Wynikowy zestaw instrukcji, odpowiadający instrukcjom faktycznie interpretowanym przez BEAM, jest dużo bardziej obszerny od zestawu który może zostać wygenerowany przez kompilator i został opisany w dodatku A. Motywacją do zmiany kodu maszynowego w ten sposób jest m.in. optymalizacja czasu wykonania często występujących po sobie operacji. W przeciwieństwie do oryginalnej maszyny wirtualnej, interpreter zawarty w niniejszej maszynie wirtualnej dokonuje bezpośredniej interpretacji opkodów, które znajdują się w pliku binarnym z kodem pośrednim.

1.2. Tablice

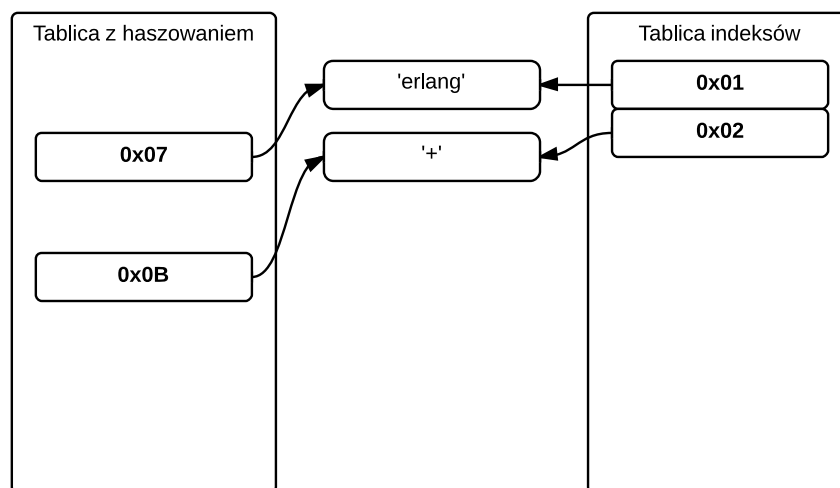
Tablice opisane w tym podrozdziale zostały zaimplementowane w plikach `hash.c`, `index.c`, `atom.c` oraz `export.c`.

Biorąc pod uwagę modułowy charakter aplikacji napisanych w języku Erlang, maszyna wirtualna musi posiadać pewien mechanizm pozwalający na utrzymywanie globalnego stanu systemu w zależności od aktualnie załadowanych modułów.

Strukturą danych przeznaczoną do tego celu jest tablica z haszowaniem wspomagana przez tablicę indeksów. Połączenie tych dwóch struktur umożliwia wstawienie nowego elementu oraz sprawdzenie jego indeksu w czasie stałym (konieczne jest wyliczenie skrótu). W takim samym czasie (nie jest do tego jednak konieczne wyliczanie skrótu) możliwe jest znalezienie obiektu znając jego indeks. Co więcej, w reprezentacji kodu maszynowego posługiwanie się indeksami obiektów trywializuje ich porównywanie czy pobieranie ich wartości w trakcie jego wykonywania.

Wybór takiej struktury danych ma więc charakter optymalizacyjny. W maszynie wirtualnej tablicowanymi obiektami są: **atomy** i **eksportowane funkcje**. Maszyna wirtualna BEAM dodatkowo tablicuje również **moduły**, gdzie przechowywane są informacje o wskaźnikach do początku kodu modułu w dwóch wersjach: nowej i starej, które mogą działać w maszynie wirtualnej niezależnie od siebie. Ponieważ jednak maszyna rozważana w pracy w obecnej fazie rozwoju nie zapewnia możliwości dynamicznego ładowania modułów, implementacja tej struktury nie było konieczne.

Na rysunku 1.1 przedstawiono sposób w jaki wewnątrz maszyny wirtualnej przechowywane są stabilicowane dane. Przykład ten dotyczy dwuelementowej tablicy atomów, które były do niej wstawiane w kolejności: `erlang`, `+`. Strzałki na diagramie reprezentują przechowywanie wskaźników na struktury atomów przez tablice: z haszowaniem i indeksów.



Rysunek 1.1: Przykład przechowywania danych stabilicowanych danych wewnątrz maszyny wirtualnej.

W maszynie BEAM, w przypadku uruchomionych dużych systemów, powyższe struktury danych mogą zawierać bardzo dużą liczbę elementów, nawet rzędu kilkudziesięciu tysięcy. Zupełnie inaczej sytuacja wygląda w niniejszej maszynie wirtualnej ze względu na jej przeznaczenie, którym są systemy wbudowane i wynikające z tego restrykcyjne limity dostępnej pamięci RAM. Rozmiary tablic nie po-

winny zatem przekraczać liczby elementów wyrażonej w setkach atomów czy funkcji eksportowanych. Niemniej jednak, ze względu na możliwość uruchomienia systemu FreeRTOS na mikrokontrolerach o różnych parametrach, pozostawiono możliwość zdefiniowania maksymalnej liczby elementów, jakie mogą zostać wstawione do tablic. Zaimplementowano również mechanizmy automatycznego rozszerzania tablic wraz ze wzrostem liczby elementów, w celu optymalizacji pamięci zajmowanej przez tablice.

Ważną cechą charakterystyczną tablic w maszynie wirtualnej jest również fakt, że raz wstawionego do nich obiektu nie można z niej usunąć. W kontekście maszyny wirtualnej rozważanej w pracy ta cecha nie ma większego znaczenia, gdyż obecnie nie umożliwia ona dynamicznego ładowania modułów po jej uruchomieniu. Jednak w przypadku maszyny BEAM nie należy zapominać o tej cesze np. w sytuacji, gdy program dynamicznie generuje atomy. Tablica atomów w BEAM może przechowywać aż 1048576 atomów, należy jednak mieć na uwadze to, że próba dodania atomu do pełnej już tablicy zakończy się zakończeniem procesu maszyny wirtualnej.

1.2.1. Tablica atomów

Funkcja skrótu dla atomów (ich reprezentacji w postaci napisu) używana w maszynie wirtualnej to *hashpjw* [4]. Jest to funkcja o bardzo dobrym rozkładzie wartości skrótu dla napisów, jednak wartość zwracana przez oryginalną funkcję jest 32-bitowa.

W celu ograniczenia pamięci zajmowanej przez tablice w maszynie wirtualnej rozważanej w pracy, funkcja haszująca została zmodyfikowana tak, aby zwracała wynik 8-bitowy. Ze względu na duże różnice w rozmiarach tablic pomiędzy rozważaną maszyną a BEAM zmniejszenie długości skrótu zwracanego z funkcji haszującej nie będzie miało wpływu na liczbę kolizji w tablicy z haszowaniem.

Źródłem atomów w tablicy są atomy zdefiniowane w samej maszynie wirtualnej oraz atomy pochodzące z ładowanych modułów. Atomy zdefiniowane ładowane są do tablicy w trakcie uruchamiania maszyny wirtualnej w określonej kolejności, co za tym idzie atomy te mają z góry ustalony i znany indeks, co jest wykorzystywane np. przy definicji funkcji wbudowanych w maszynę wirtualną (por. 1.7). Z kolei indeksy atomów, które pochodzą z ładowanych modułów, a nie zostały wcześniej zdefiniowane, przydzielane są w kolejności ładowania modułów i występowania atomów w tablicach atomów modułów. Równość dwóch atomów oznacza zawsze równość ich indeksów w globalnej tablicy atomów i na odwrót, niezależnie od źródła ich pochodzenia ani momentu załadowania modułu do maszyny wirtualnej.

1.2.2. Tablica eksportowanych funkcji

Funkcja skrótu dla eksportowanych funkcji ma wartość $M \cdot F + A$, gdzie M to indeks w tablicy atomów dla nazwy modułu z którego eksportowana jest funkcja, F to indeks w tablicy atomów dla nazwy eksportowanej funkcji, a A to arność tej funkcji.

Wpisy w tablicy eksportowanych funkcji pochodzą z modułów załadowanych do maszyny wirtualnej, dla funkcji które zostały zdefiniowane w lokalnej tablicy eksportów dla danego modułu. W takiej sytuacji w tablicy eksportów przechowywany jest wskaźnik na miejsce w pamięci, w którym znajduje się

pierwsza instrukcja funkcji. Interpreter, wykonując kod używa indeksu do odczytania adresu tej instrukcji, a następnie wykonuje skok do tego miejsca pamięci i kontynuuje wykonywanie kodu, po uprzednim zapisaniu adresu powrotu.

Elementy tablicy mogą pochodzić również z wbudowanych funkcji (por. 1.7). W tym przypadku, tablica eksportów zawiera wskaźnik do funkcji w języku C, zaimplementowanej jako część maszyny wirtualnej, która zostanie wykonana przez interpreter.

Ponieważ równość indeksów w tablicy eksportów jest równoważna z równością trójek {moduł, funkcja, arność}, w sytuacji dynamicznej podmiany kodu nie jest konieczna zmiana indeksu w załadowanym do pamięci kodzie maszynowym, a tylko odpowiednia zmiana struktury znajdującej się pod tym indeksem.

1.3. Typy danych

Erlang jest językiem programowania o dynamicznym, lecz silnym typowaniu. Oznacza to, że każda zmienna, po przypisaniu do niej wartości ma ustalony konkretny typ danych, którego nie można zmienić. Niemożliwe jest również rzutowanie zmiennej na innych typ danych - konwersja do innego typu musi zostać wykonana jawnie a nowa zmienna zajmuje w takiej sytuacji inne miejsce w pamięci programu.

Wszystkie wyrażenia rozpoznawane przez interpreter kodu maszynowego Erlanga zapisane są w postaci wyrażenia takiego samego typu, z punktu widzenia języka C, o długości równej słowu maszynowemu dla danej architektury. W celu rozróżnienia typów zmiennych w pamięci programu, w maszynie wirtualnej wprowadzony został mechanizm **tagowania**, czyli oznaczania w różny sposób zmiennych w pamięci, w zależności od ich typu. Mechanizm ten został zaprojektowany w taki sposób, aby dodatkowy rozmiar w pamięci przeznaczony dla typu był jak najmniejszy. Sposób jego działania został przedstawiony w niniejszej sekcji.

1.3.1. Wartości bezpośrednie a pośrednie

Podstawowy podział typów danych wewnątrz maszyny wirtualnej Erlanga wynika ze względu na sposób dostępu do danych.

Jeżeli dana może zostać przechowana na odpowiednio małym obszarze pamięci, czyli w jednym słowie maszynowym (w przypadku niniejszej maszyny są to 32 bity) z uwzględnieniem tagu oznaczającego typ to dane tego typu nazywane są wartościami bezpośrednimi (**IMMED**). Aby dokonać tagowania lub odczytania wartości ze zmiennej tego typu wystarczy wykonać jedną operację przesunięcia bitowego.

Przeciwnieństwem danych bezpośrednich są dane pośrednie, które mogą przybrać postać listy (**CONS**) lub typu opakowanego (**BOXED**). Wyrażenia oznaczone tagiem dla jednego z tych typów przechowują fizyczny wskaźnik na miejsce w pamięci, gdzie znajdują się dla nich właściwe dane. Przy tagowaniu wskaźników wykorzystany został fakt, że bloki pamięci alokowane przez maszynę wirtualną są zawsze wielokrotnością całego słowa maszynowego. Co za tym idzie dwa najmniej znaczące bity wskaźnika, na maszynie uruchomionej w architekturze 32-bitowej lub wyższej będą zawsze zerami co

można wykorzystać do przechowania dodatkowej, dwubitowej, informacji. W tym wypadku jest to tag rozróżniający wskaźniki na listy od wskaźników na typy opakowane oraz od pozostałych wyrażeń.

Tabela 1.1 prezentuje sposób tagowania ww. typów danych. Tagi dla poszczególnych typów danych zostały w zapisie słowa maszynowego pogrubione.

Na przykład do przechowania wskaźnika do pierwszego elementu listy, znajdującego się pod adresem 128, w pamięci zapisane zostanie wyrażenie:

$$10000000 \vee \mathbf{01} = 00000000 \ 00000000 \ 00000000 \ 100000\mathbf{01}$$

Do odczytania wartości wskaźnika wystarczy więc wyzerowanie dwóch najmniej znaczących bitów wyrażenia:

$$00000000 \ 00000000 \ 00000000 \ 100000\mathbf{01} \wedge \neg(11) = 10000000$$

Typ danych	Słowo maszynowe (binarnie)	Opis
IMMED	IIIIIIII IIIIIIII IIIIIIII IIBBTT 11	<ul style="list-style-type: none"> – bity T budują konkretny tag (por. 1.3.2) typu bezpośredniego; – bity I oznaczają wartość przechowywaną przez wyrażenie, która w zależności od rozmiaru tagu może mieć 26 lub 28 bitów; – bity B mogą być dwoma najbardziej znaczącymi bitami tagu lub dwoma najmniej znaczącymi bitami przechowywanej wartości, w zależności od typu.
CONS	PPPPPPPP PPPPPPPP PPPPPPPP PPPPPP 01	<ul style="list-style-type: none"> – bity P są 30 najbardziej znaczącymi bitami wskaźnika do wyrażenia stanowiącego pierwszy element listy, dwa najmniej znaczące bity zawsze będą zerami dlatego mogą zostać nadpisane przez tag.
BOXED	PPPPPPPP PPPPPPPP PPPPPPPP PPPPPP 10	<ul style="list-style-type: none"> – bity P są 30 najbardziej znaczącymi bitami wskaźnika do nagłówka identyfikujące typ i rozmiar opakowanych danych, w tym przypadku również dwa najmniej znaczące bity zawsze będą zerami.

Tablica 1.1: Rozróżnienie tagów ze względu na sposób dostępu do danych

1.3.2. Wartości bezpośrednie

Wartości bezpośrednie (**IMMED**) mogą być przechowywane przez różne typy danych, dla których przewidziano dodatkowe 2 lub 4 bity na tag. Tabela 1.2 podsumowuje wszystkie zaimplementowane w maszynie wirtualnej opisywanej w pracy typy zawierające wartości bezpośrednie.

Typ danych	Słowo maszynowe (binarnie)	Opis
PID	IIIIIIII IIIIIIII IIIIIIII IIII 0011	Wartością przechowywaną przez wyrażenie tego typu jest indeks procesu w tablicy procesów w maszynie wirtualnej (por. 1.5).
SMALL_INT	IIIIIIII IIIIIIII IIIIIIII IIII 1111	Przechowywaną wartością jest liczba całkowita (ze znakiem), którą można zapisać na maksymalnie 28 bitach w pamięci.
ATOM	IIIIIIII IIIIIIII IIIIIIII II 001011	Wartością przechowywaną jest indeks atomu w tablicy atomów (por. 1.2.1). Dzięki takiemu zapisowi porównanie dwóch dowolnych atomów sprowadza się do porównania dwóch 32-bitowych liczb.
NIL	00000000 00000000 00000000 00 111011	Przechowywaną wartością jest zawsze zero. Wyrażenie to służy do oznaczania końca listy.

Tablica 1.2: Rozróżnienie tagów dla danych bezpośrednich

Na przykład atom mający indeks $2_{10} = 10_2$ w tablicy atomów w pamięci będzie zapisany w postaci:

$$(10 \ll 6) \vee \mathbf{001011} = 00000000 \ 00000000 \ 00000000 \ 10\mathbf{001011}$$

Do odczytania indeksu atomu wystarczy zatem wykonać operację przesunięcia bitowego w prawo:

$$00000000 \ 00000000 \ 00000000 \ 10\mathbf{001011} \gg 6 = 10$$

1.3.3. Listy

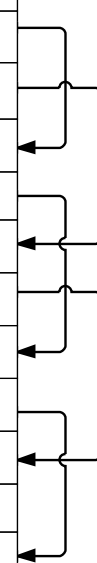
Listy są jednym ze złożonych typów danych obsługiwanych przez język Erlang. W maszynie wirtualnej zaimplementowane zostały przy użyciu listy jednokierunkowej. Wyrażenie, które służy np. do przechowywania listy na stosie procesu lub przekazywania jej jako argument do funkcji otagowane jest jako typ **CONS** (por. 1.3.1) i zawiera wskaźnik do pierwszego elementu listy. Element listy jest zwykłym wyrażeniem erlangowym, a więc zajmuje jedno słowo maszynowe. Słowem maszynowym następującym po elemencie jest kolejne wyrażenie typu **CONS**, które zawiera wskaźnik do kolejnego elementu listy. Wyrażenie to może być również typu **NIL** (por. 1.3.2), co oznacza że dany element był ostatnim elementem listy.

W Erlangu nie ma osobnego typu do przechowywania ciągu znaków. Udostępniony lukier składniowy pozwala jednak na posługiwanie się napisami, np. w postaci: "hello". Wyrażenie tego typu

zostanie jednak zinterpretowane jak lista liczb całkowitych, odpowiadającymi kodom ASCII kolejnych liter w napisie. W tym przypadku będzie to następująca lista: `[104, 101, 108, 108, 111]`.

Na rysunku 1.2 zaprezentowano przykładową stertę procesu Erlanga zawierającą powyższą listę, wraz z wyjaśnieniem typu i wartości zawartych w poszczególnych słowach maszynowych. Wyrażenie będące początkiem listy znajduje się w pierwszym wierszu sterty. Strzałki na diagramie reprezentują zawieranie wskaźnika do innego miejsca w pamięci przez wyrażenie z którego wychodzą.

Adres	Słowo maszynowe	Opis wyrażenia
	00000000 00000000 00000000 1000000 1	128 CONS
132	00000000 00000000 00000000 0111100 1	120 CONS
128	00000000 00000000 00000110 1000 1111	104 SMALL_INT
124	00000000 00000000 00000000 0111000 1	112 CONS
120	00000000 00000000 00000110 0101 1111	101 SMALL_INT
116	00000000 00000000 00000000 0110100 1	104 CONS
112	00000000 00000000 00000110 1100 1111	108 SMALL_INT
108	00000000 00000000 00000000 0110000 1	096 CONS
104	00000000 00000000 00000110 1100 1111	108 SMALL_INT
100	00000000 00000000 00000000 00 111011	NIL
96	00000000 00000000 00000110 1111 1111	111 SMALL_INT



Rysunek 1.2: Przykład przechowywania listy na sterze procesu

Powyższy przykład dobrze ilustruje narzut pamięciowy jaki wprowadza sposób zapisu napisu przy użyciu listy. Informacja, która przy użyciu innych języków programowania może być zapisana przy użyciu 5 bajtów w języku Erlang potrzebuje aż 10 słów maszynowych (40 bajtów na architekturze 32-bitowej). Receptą na tego typu problem, wprowadzoną w maszynie wirtualnej BEAM, jest binarny typ danych. Napis "hello" przy jego użyciu zajmowałby w pamięci 3 słowa maszynowe (nagłówek i 2 słowa przeznaczone na dane). Typ ten jednak nie został zaimplementowany w obecnej wersji maszyny na system FreeRTOS.

Jak można zauważyć zarówno złożoność obliczeniowa (dostęp do danych na liście trwa czas liniowy) jak i pamięciowa przy wykorzystaniu tego typu danych jest dość znacząca.

1.3.4. Krotki

Kolejnym złożonym typem danym, z którego można korzystać w języku Erlang jest krotka, zajmująca spójny obszar pamięci. Z implementacyjnego punktu widzenia można porównać ją do tablicy

zawierającej wyrażenia erlangowe.

Krotka jest jednym z typów opakowanych (**BOXED**), zatem referencja do niej z poziomu stosu procesu zawiera wskaźnik do nagłówka krotki. Nagłówek, podobnie jak pozostałe wyrażenia zajmuje jedno słowo maszynowe i przechowuje rozmiar krotki w postaci:

AAAAAAAA AAAAAAAAA AAAAAAAAA AA000000

gdzie bity A oznaczają rozmiar (arność) krotki. Wyrażenia wchodzące w skład krotki zajmują kolejne, następujące po nagłówku, słowa maszynowe.

Na rysunku 1.3 zaprezentowany został przykład przechowywania danych wewnątrz krotki na stercie procesu, jak w przykładzie na rys. 1.2, czyli krotki {104, 101, 108, 108, 111}.

Adres	Słowo maszynowe	Opis wyrażenia
	00000000 00000000 00000000 10000110	132 BOXED
132	00000000 00000000 00000001 01000000	005 ARITYVAL
128	00000000 00000000 00000110 10001111	104 SMALL_INT
124	00000000 00000000 00000110 01011111	101 SMALL_INT
120	00000000 00000000 00000110 11001111	108 SMALL_INT
116	00000000 00000000 00000110 11001111	108 SMALL_INT
112	00000000 00000000 00000110 11111111	111 SMALL_INT

Rysunek 1.3: Przykład przechowywania krotki na stercie procesu

Ze względu na to, że krotka przechowuje dane w spójnym obszarze pamięci i dostęp do nich odbywa się w czasie stałym, użycie tego typu danych jest dobrym pomysłem w przypadku przechowywania danych tablicowych.

1.3.5. Duże liczby

Drugim zaimplementowanym, opakowanym typem danych są duże liczby całkowite. Wszystkie liczby całkowite, które nie mieszczą się w zakresie typu **SMALL_INT**, czyli potrzebują do ich zapisania przynajmniej 29 bitów zapisywane są w typie danych **BIGNUM**. Maszyna wirtualna zaimplementowana w ramach pracy, podobnie jak maszyna BEAM, implementuje własną arytmetykę implementującą operacje na tego typu liczbach.


Nagłówek typu **BOXED** ma w tym przypadku postać:

AAAAAAAA AAAAAAAAA AAAAAAAAA AA001S00

gdzie bity A oznaczają liczbę słów maszynowych, które składają się na całą liczbę bez znaku. Słowa te, w kolejności od najmniej do najbardziej znaczącego, zajmują w pamięci kolejne słowa maszynowe po nagłówku. Bit S przechowuje znak liczby: 1 dla liczb ujemnych, 0 dla dodatnich.

Na rysunku 1.4 zaprezentowany został przykład przechowania liczby 10^{28} na stercie procesu.

Adres	Słowo maszynowe	Opis wyrażenia
	00000000 00000000 00000000 10000110	132 BOXED
132	00000000 00000000 00000000 11001000	003 BIGNUM (+)
128	00010000 00000000 00000000 00000000	0x10000000
124	00111110 00100101 00000010 01100001	0x3e250261
120	00100000 01001111 11001110 01011110	0x204fce5e



Rysunek 1.4: Przykład przechowywania dużej liczby na stercie procesu

1.3.6. Niezaimplementowane typy danych

Typy danych dostępne w maszynie BEAM, które nie zostały zaimplementowane w pracy to:

- **lambdy** (tag **FUN**) - ten typ danych używany jest w komunikacji między węzłami wewnątrz klastra Erlanga, co nie jest aktualnie obsługiwane przez maszynę wirtualną, lambdy pojawiające się w kodzie źródłowym modułów reprezentowane są za pomocą funkcji lokalnych i nie potrzebują osobnego typu danych do poprawnego działania;
- **referencje** (tag **REF**) - referencje z założenia używane są do oznaczania wiadomości wysyłanych do innych węzłów z uruchomioną maszyną wirtualną a klastrowanie nie jest wspierane przez implementowaną maszynę wirtualną;
- **porty** (tag **PORT**) - porty używane są do identyfikacji procesów uruchomionych w systemie operacyjnym poza maszyną wirtualną Erlanga, a do których delegowane są pewne operacje, wykonywane na poziomie systemu operacyjnego. Wykonanie tych operacji (jak np. operacje na plikach) z założenia może zająć pewien dłuższy okres czasu, co mogłoby zakłócić harmonogramowanie procesów. Typ danych nie został zaimplementowany, gdyż w maszynie wirtualnej wszystkie operacje wykonywane na poziomie mikrojądra zostały zaimplementowane przy pomocy funkcji wbudowanych (por. 1.7);
- **liczby zmiennoprzecinkowe** (tag **FLONUM**) - ten typ danych nie został zaimplementowany w wersji maszyny opisywanej w pracy. Liczby zmiennoprzecinkowe rzadko wykorzystywane są

w programowaniu urządzeń wbudowanych ze względu na bardzo duży narzut czasowych w przypadku mikrokontrolerów nie mających koprocatora (np. procesor ARM Cortex-M3 nie posiada FPU). M.in. z tego powodu do maszyny BEAM arytmetyka zmiennoprzecinkowa została dodana dopiero w wersji R8;

- **binaria** (tagi *_**BINARY**) - binarny typ danych wyłącznie ze wszystkimi operacjami dotyczącymi dopasowywania do nich zmiennych również nie został zaimplementowany w maszynie. Jest to funkcjonalność warta rozważenia w przypadku dalszej pracy nad maszyną ze względu na binarny charakter szeregowych interfejsów wejścia-wyjścia obsługiwanych przez mikrokontrolery. Podstawowe operacje na binariach do maszyny BEAM zostały dodane w wersji R7, bardziej zaawansowane były sukcesywnie dodawane od wersji R10.

1.4. Interpreter kodu maszynowego

Moduł opisany w niniejszym podrozdziale został zaimplementowany w pliku źródłowym `beam_emu.c`.

1.4.1. Maszyna stosowa a rejestrowa

Spośród sposobów implementacji maszyny wirtualnej można wymienić dwa: maszynę stosową i rejestrową. Różnicę między tymi dwoma podejściami stanowi sposób przechowywania argumentów wywoływanej funkcji, miejsca zapisu wyniku jej wykonania oraz zmiennych tymczasowych przez nią używanych.

W przypadku maszyny stosowej dane te przechowywane są na stosie. Kolejne argumenty operacji umieszczane są na stosie za pomocą operacji **PUSH**, natomiast przed wykonaniem operacji zdejmowane są przez instrukcję **POP**. Instrukcje wykonywane przez maszynę wirtualną nie potrzebują zatem żadnych dodatkowych argumentów, gdyż te powinny być przed jej wywołaniem umieszczone na szczycie stosu, podobnie jak wynik zwrócony przez wykonaną instrukcję.

Z kolei w przypadku maszyny rejestrowej, powyższe informacje zapisywane są w zbiorze rejestrów. Konsekwencją tego jest brak instrukcji manipulujących stosem w wykonywanym kodzie, co wpływa pozytywnie na szybkość działania interpretera kodu pośredniego. Właściwe instrukcje wymagają jednak przekazania dodatkowych argumentów adresujących rejestry, w których znajdują się argumenty operacji i rejestr docelowy dla wyniku jej wykonania. Efektem tego jest dłuższy zapis instrukcji w kodzie pośrednim niż w przypadku maszyny stosowej. Dodatkowym atutem użycia rejestrów jest możliwość optymalizacji kodu polegającego na wyliczeniu i zapisaniu do rejestru pewnego pośredniego wyniku. Wynik ten może następnie zostać wykorzystany przez kilka różnych instrukcji, zamiast wyliczania go od nowa przez każdą z nich.

Przykładami stosowych maszyn są maszyny języków takich jak Java czy .NET. Z kolei przykładami maszyn rejestrowych są maszyny wirtualne języka Lua czy maszyna wirtualna Javy na system Android - Dalvik.

1.4.2. Model interpretera

Maszyna wirtualna języka Erlang (BEAM oraz maszyna zaimplementowana w pracy) jest również przykładem rejestrowej maszyny wirtualnej.

Interpreter korzysta z następujących rejestrów:

- rejestry X_0 - X_{255} służące do przechowywania kolejnych argumentów z jakimi wywoływana jest funkcja, dodatkowo w rejestrze X_0 zapisywana jest wartość zwracana przez funkcję;
- rejestry Y znajdujące się na stosie aktualnie uruchomionego procesu, służące do przechowywania zmiennych lokalnych;
- rejestr IP przechowujący wskaźnik do aktualnie wykonywanej przez interpreter instrukcji;
- rejestr CP przechowujący adres powrotu - wskaźnik do instrukcji, którą interpreter powinien wykonać gdy nastąpi powrót z aktualnie wykonywanej funkcji.

Na listingu 1.1 zaprezentowany został przykładowy kod pośredni funkcji o arności 3, wyliczającej maksimum ze wszystkich jej argumentów, wykorzystujący do tego celu funkcję wbudowaną `erlang:max/2`. Wyjaśnienie działania poszczególnych operacji zostało zawarte w dodatku A.

```

1 | {allocate, 1, 3}.
2 | {move, {x, 2}, {y, 0}}.
3 | {call_ext, 2, {extfunc, erlang, max, 2}}.
4 | {move, {y, 0}, {x, 1}}.
5 | {call_ext_last, 2, {extfunc, erlang, max, 2}, 1}.

```

Listing 1.1: Fragment kodu pośredniego liczącego maksimum z trzech wyrażeń modułu

Na rysunkach od 1.5 do 1.12 zaprezentowano zawartość rejestrów X (interpretera) i Y (na stosie procesu) po wykonaniu poszczególnych operacji kodu powyższego dla argumentów: 5, 7, 9. Zapis $\{1, 1\}$ oznacza, że wskaźnik do instrukcji wskazuje na linię 1 z przykładu na listingu 1.1, z kolei `erlang:max/2` oznacza, że wskaźnik do instrukcji wskazuje na początek tej funkcji wbudowanej. Przez zapis CP rozumiana jest wartość wskaźnika CP w chwili wywołania funkcji.

Jak można zauważyć na rys. 1.5 wszystkie trzy argumenty zostały umieszczone w rejestrach X : 0, 1 i 2. Instrukcja `{allocate, 1, 3}`, znajdująca się w pierwszej linii powoduje rozszerzenie stosu procesu o 2 wyrażenia, z czego na szczycie stosu umieszczany jest adres powrotu, z jakim została wywołana funkcja, zapisany pod wskaźnikiem CP .

W linii 2 (rys. 1.7) dokonane zostaje przeniesienie wyrażenia z rejestru X_2 do rejestru Y_0 , który jest pierwszym wyrażeniem na stosie poniżej jego szczytu. Należy zauważyć, że pomimo wykorzystania

stosu do zapisu zmiennych lokalnych, które nie zostaną zmodyfikowane pomiędzy wywołania funkcji, do operacji na nim nie wykorzystuje się operacji stosowych. Nie należy zatem wiązać wykorzystania stosu procesu z faktem, że maszyna wirtualna jest maszyną stosową.

Przed wywołaniem funkcji `erlang:max/2` w linii 3, jako adres powrotu zostaje zapisana kolejna instrukcja w module, znajdująca się w linii 4. Argumentami wywoływanej funkcji są wartości znajdujące się w rejestrach X_0 i X_1 . Wywołana funkcja w momencie powrotu przepisuje wskaźnik powrotu (**CP**) na kolejną instrukcję do wykonania (**IP**). Wartość zwrócona z funkcji znajduje się w rejestrze X_0 .

W linii 5 (rys. 1.10) wartość przechowywana na stosie zostaje przepisana do rejestru X_1 . Drugie wywołanie funkcji `erlang:max/2` jest wywołaniem ogonowo-rekurencyjnym. Dlatego też przykładowa funkcja odpowiedzialna jest za przywrócenie wskaźnika **CP** ze stosu i zwolnienie go jeszcze przed wywołaniem funkcji zewnętrznej. Wartość zwrócona z trójargumentowej funkcji znajduje się w rejestrze X_0 , a kolejna wykonana instrukcja będzie następująca po instrukcji, która ją wywołała.

1.5. Procesy

1.6. Planista (*scheduler*)

preemptive/cooperative, redukcje, wywłaszczenie, priorytety

1.7. Funkcje wbudowane (*Built-In Functions*)

1.7.1. Funkcje ogólne (moduł `erlang`)

1.7.2. Listy (moduł `lists`)

1.7.3. GPIO i obsługa przerwań (moduł `lpc_gpio`)

1.7.4. SPI (moduł `lpc_spi`)

1.8. *Garbage collector*

Cheney, długość sterty, rootset, gc

1.9. Mechanizmy zarządzania czasem

1.10. Podsumowanie różnic z maszyną BEAM

Na elementy, z których składa się maszyna wirtualna Erlanga można popatrzeć z punktu widzenia poszczególnych obszarów pamięci przez nią zajmowanych.

Do elementów tych należą:

Rejestry (X)	Stos (Y)	Instrukcje
{x, 0}	5	IP {1, 1}
{x, 1}	7	CP CP
{x, 2}	9	

Rysunek 1.5: Rejestry przed wykonaniem instrukcji w linii 1

Rejstry (X)		Stos (Y)		Instrukcje	
{x, 0}	7	{y, -1}	CP	IP	{1, 4}
{x, 1}	7	{y, 0}	9	CP	{1, 4}
{x, 2}	9				

Rysunek 1.9: Rejestry przed wykonaniem instrukcji w linii 4

Rejestry (X)		Stos (Y)		Instrukcje	
{x, 0}	5	{y, -1}	CP	IP	{1, 2}
{x, 1}	7	{y, 0}	?	CP	CP
{x, 2}	9				

Rysunek 1.6: Rejestry przed wykonaniem instrukcji w linii 2

Rejestry (X)		Stos (Y)		Instrukcje	
{x, 0}	7	{y, -1}	CP	IP	{1, 5}
{x, 1}	9	{y, 0}	9	CP	{1, 4}
{x, 2}	9				

Rysunek 1.10: Rejestry przed wykonaniem instrukcji w linii 5

Rejestry (X)		Stos (Y)		Instrukcje	
{x, 0}	5	{y, -1}	CP	IP	{1, 3}
{x, 1}	7	{y, 0}	9	CP	CP
{x, 2}	9				

Rysunek 1.7: Rejestry przed wykonaniem instrukcji w linii 3

Rejstry (X)	Stos (Y)	Instrukcje
{x, 0}	7	IP erlang: max/2
{x, 1}	9	CP CP
{x, 2}	9	

Rysunek 1.11: Rejestry przed wywołaniem funkcji w linii 5

Rejestry (X)		Stos (Y)		Instrukcje	
{X, 0}	5	{y, -1}	CP	IP	erlang: max/2
{X, 1}	7	{y, 0}	9	CP	{1, 4}
{X, 2}	9				

Rysunek 1.8: Rejestry przed wywołaniem funkcji w linii 3

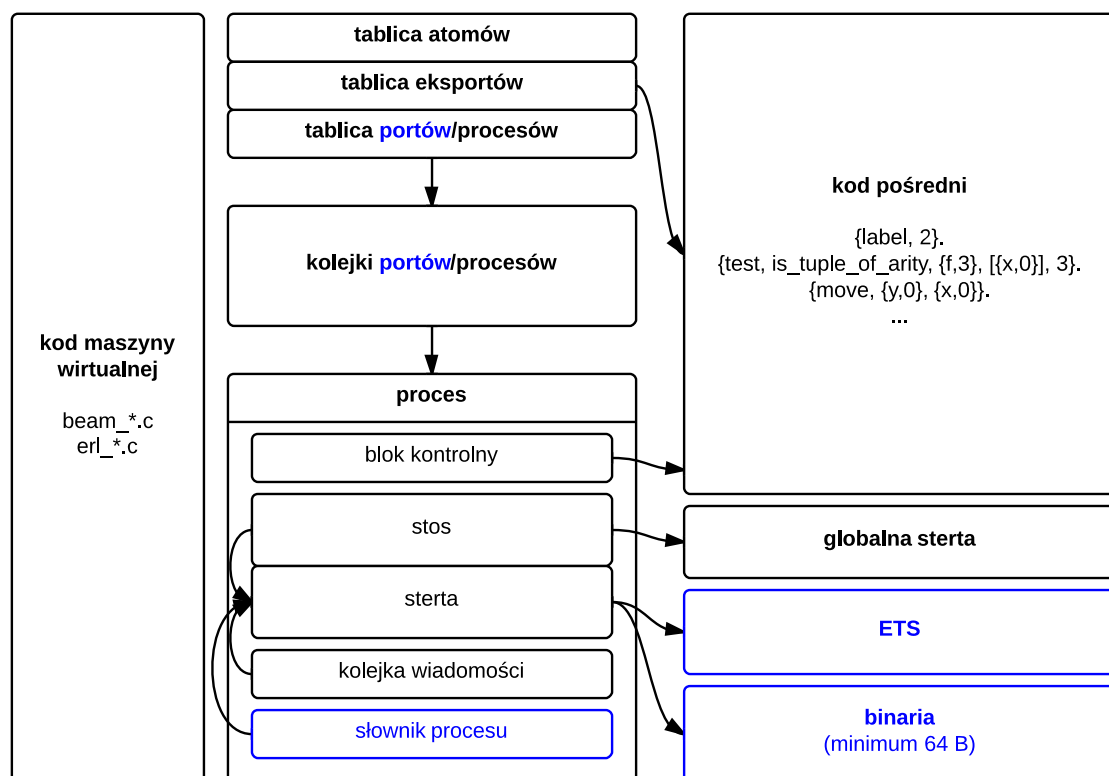
Rejstry (X)	Stos (Y)	Instrukcje	
{x, 0}	9	IP	CP
{x, 1}	9	CP	CP
{x, 2}	9		

Rysunek 1.12: Rejestry po wykonaniu instrukcji w linii 5, a zarazem i całej funkcji

- skompilowany kod wykonywalny modułów opisanych w niniejszym rozdziale;
- tablice: atomów, eksportowanych funkcji, portów i procesów;
- kolejki procesów i portów, na podstawie których *scheduler* decyduje o wyborze kolejnego procesu do wykonania;
- uruchomione procesy z informacjami kontrolnymi, kolejkami wiadomości i pamięcią do nich należąca;
- załadowany kod pośredni modułów, który wykonywany jest w kontekście procesów;
- globalna sarta, na której umieszczane są stałe wyrażenia pochodzące z plikami z modułów;
- tablice ETS (*Erlang Term Storage*) stanowiące w maszynie wirtualnej mutowalny obszar pamięci, w postaci bazy danych klucz-wartość;
- sarta, na której przechowywane są dane typu binarnego, których długość wynosi przynajmniej 64 bajtów. Dane te mogą być współdzielone przez procesy, których liczba przechowywana jest przez licznik referencji, na którego podstawie *garbage collector* zwalnia nieużywane już obszary pamięci.

Na rysunku 1.13 zaprezentowany został powyższy podział, z uwzględnieniem elementów które posiada zarówno maszyna BEAM jak i maszyna zaimplementowana w pracy, które zostały oznaczone kolorem czarnym. Kolorem niebieskim z kolei oznaczone zostały elementy, które nie zostały zaimplementowane w maszynie na system FreeRTOS. Strzałki na diagramie przedstawiają fakt przechowywania wskaźników na elementy w jednym obszarze pamięci przed drugi.

Elementami, które nie zostały zaimplementowane w pracy są: tablice ETS oraz słownik procesu, których użycie nie jest konieczne do implementacji w pełni funkcjonalnych modułów w języku Erlang. Nie pozostały również zaimplementowane sarta binariów oraz tablice i kolejki portów, ze względu na niezaimplementowanie tych typów danych w maszynie.



Rysunek 1.13: Elementy maszyny wirtualnej Erlanga jako obszary pamięci.

A. Lista instrukcji maszyny wirtualnej BEAM

Dodatek zawiera listę instrukcji maszyny wirtualnej BEAM, jakie może zawierać skompilowany kod pośredni przez nią wykonywany oraz sposób zapisu argumentów dla instrukcji.

Kod danej operacji zajmuje zawsze 1 bajt w pliku ze skompilowanym kodem pośrednim modułu. Argumenty mogą zajmować więcej przestrzeni, zgodnie z opisem w sekcji A.1.

Kolejność bajtów w zapisie kodu pośredniego to zawsze *big endian*.

A.1. Typy argumentów

Argumentem jest zawsze liczba całkowita, reprezentująca wartość liczbową bądź indeks w odpowiedniej tablicy z wartościami (pierwszym indeksem takiej tablicy jest 0). W związku z tym argumenty mogą być różnego typu. Aby rozróżnić argument jednego typu od drugiego poddaje się je odpowiedniemu tagowaniu. Operację wykonuje się bezpośrednio na argumentie, jeśli jest dostatecznie mały, lub na odpowiednim nagłówku poprzedzającym argument. Rozróżnienie to jest spowodowane oszczędnością rozmiaru kodu pośredniego, który musi być przechowywany w pamięci.

Każdy z tagów, które zostały wymienione w tabeli A.1, jest możliwy do zapisania przy użyciu 3 bitów, które zajmują najmniej znaczące bity argumentu. Jednak w kodowaniu binarnym do zapisu typu używane są dodatkowo 1 lub 2 bity. Dzięki nim możliwe jest rozróżnienie pomiędzy argumentami zapisanymi przy użyciu różnej liczby bajtów.

Tagowanie odbywa się za pomocą następującej operacji:

$$(0000XXXX \ll N)_{(2)} \vee 000\mathbf{S}\mathbf{T}\mathbf{T}\mathbf{T}_{(2)},$$

gdzie $XXXX_{(2)}$ jest tagowaną liczbą, $N = 4$ lub 5 , $\mathbf{SS}_{(2)}$ są dodatkowymi bitami znakującymi rozmiar argumentu, a $\mathbf{TTT}_{(2)}$ jest danym tagiem.

Tag		Typ
binarnie	dziesiętnie	
000	0	uniwersalny indeks, np. do tablicy stałych
001	1	liczba całkowita
010	2	indeks do tablicy atomów

011	3	numer rejestru X maszyny wirtualnej
100	4	numer rejestru Y maszyny wirtualnej
101	5	etykieta, używana w funkcjach skoku
111	7	złożone wyrażenie (np. lista, liczba zmiennoprzecinkowa)

Tablica A.1: Tagi typów danych w pliku ze skompilowanym modulem

Jeżeli tagowana liczba jest nieujemna, mniejsza od 16 (możliwe jest zapisanie jej przy użyciu 4 bitów) to argument jest zapisany przy użyciu jednego bajtu a jego postać binarna to:

$$X_1X_2X_3X_4\mathbf{0TTT}_{(2)},$$

gdzie $X_1X_2X_3X_4$ to tagowana liczba, X_1 jest jej najbardziej znaczącym bitem, a $TTT_{(2)}$ to tag danego typu argumentu.

Na przykład, atom, który w tablicy atomów modułu ma indeks $2_{10} = 10_2$, po zakodowaniu będzie miał postać:

$$00100010_2 = 22_{16} = 34_{10}.$$

W przypadku, gdy liczba jest nieujemna, mniejsza lub równa 16, a mniejsza od 2048 (możliwe jest jej zapisanie przy użyciu 11 bitów), argument jest zapisany przy użyciu dwóch bajtów, których postać binarna to:

$$X_1X_2X_3\mathbf{01TTT} X_4X_5X_6X_7X_8X_9X_{10}X_{11(2)},$$

gdzie $X_1...X_{11(2)}$ to tagowana liczba, X_1 jest jej najbardziej znaczącym bitem, a $TTT_{(2)}$ to tag danego typu argumentu.

Na przykład, liczba całkowita $565_{10} = 010\ 00110101_2$ po zakodowaniu będzie miała postać:

$$01001001\ 00110101_2 = 4935_{16} = 18741_{10}.$$

Jeżeli argument jest liczbą ujemną lub dodatnią wymagającą w zapisie dwójkowym więcej niż 11 bitów to liczba taka zapisywana jest binarnie w kodzie uzupełnień do dwóch (U2) poprzedzona odpowiednim nagłówkiem.

Jeżeli zakodowaną liczbę można zapisać na nie więcej niż 8 bajtach, to nagłówek ma następującą postać:

$$N_1N_2N_3\mathbf{11TTT}_{(2)},$$

gdzie $N_1N_2N_3$ to rozmiar argumentu w bajtach pomniejszony o 2 (jeżeli argument jest liczbą ujemną zajmującą 1 bajt to powinien on zostać dopełniony do 2 bajtów), N_1 jest jego najbardziej znaczącym bitem, a $TTT_{(2)}$ to tag danego typu argumentu.

Na przykład, aby zapisać na dwóch bajtach liczbę $-21_{10} = 11111111\ 11101011_{U2}$, jej postać binarną należy poprzedzić nagłówkiem:

$$00011001_2 = 19_{16} = 25_{10}.$$

Jeżeli do zapisania liczby w kodzie uzupełnień do dwóch potrzeba przynajmniej 9 bajtów, wtedy nagłówek jest dwubajtowy i ma postać:

$$11111\mathbf{TTT} \ N_1N_2N_3N_4\mathbf{0000}_{(2)},$$

gdzie $N_1N_2N_3N_4_{(2)}$ to rozmiar argumentu w bajtach pomniejszony o 9, N_1 jest jego najbardziej znaczącym bitem, a $\mathbf{TTT}_{(2)}$ to tag danego typu argumentu.

Na przykład, w celu zapisania liczby $2^{(15 \times 8) - 1} - 1$ na 15 bajtach, należy zapis tej liczby w kodzie U2 poprzedzić następującym nagłówkiem:

$$11111\mathbf{001} \ 0110\mathbf{0000}_2 = F960_{16} = 63840_{10}.$$

A.2. Lista instrukcji

W tabeli A.2 zawarto listę instrukcji rozumianych przez maszynę wirtualną BEAM wraz z jednobajtowym kodem operacji, listą jej argumentów i krótkim opisem działania.

Na liście oznaczono operacje, które zostały zaimplementowane w maszynie wirtualnej opisywanej w pracy. Brak implementacji poszczególnych instrukcji podyktowanym jest brakiem wsparcia pewnych funkcjonalności lub typów danych w maszynie.

Instrukcje nieużywane przez kompilator Erlanga w wersji R16 zostały pominięte.

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
01	1	label Lbl	Wprowadza lokalną dla danego modułu etykietę identyfikującą aktualne miejsce w kodzie.	✓
02	2	func_info M F A	Definiuje funkcję F, w module M o arności A.	✓
03	3	int_code_end	Oznacza koniec kodu.	✓
04	4	call Arity Lbl	Wywołuje funkcję o arności Arity znajdującą się pod etykietą Lbl. Zapisuje następną instrukcję jako adres powrotu (wskaźnik CP).	✓

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
05	5	<code>call_last Arity Lbl Dest</code>	Wywołuje rekurencyjną ogonowo funkcję o arności <code>Arity</code> znajdującą się pod etykietą <code>Lbl</code> . Nie zapisuje adresu powrotu. Przed wywołaniem zwalnia <code>Dest</code> słów pamięci na stosie.	✗
06	6	<code>call_only Arity Lbl</code>	Wywołuje rekurencyjną ogonowo funkcję o arności <code>Arity</code> znajdującą się pod etykietą <code>Lbl</code> . Nie zapisuje adresu powrotu.	✓
07	7	<code>call_ext Arity Dest</code>	Wywołuje zewnętrzną funkcję o arności <code>Arity</code> mającą indeks <code>Dest</code> w tablicy funkcji zewnętrznych. Zapisuje następną instrukcję jako adres powrotu (wskaźnik CP).	✗
08	8	<code>call_ext_last Arity Des Dea</code>	Wywołuje rekurencyjną ogonowo zewnętrzną funkcję o arności <code>Arity</code> mającą indeks <code>Des</code> w tablicy funkcji zewnętrznych. Nie zapisuje adresu powrotu. Przed wywołaniem zwalnia <code>Dea+1</code> słów pamięci na stosie. Przywraca wskaźnik CP ze stosu.	✗
09	9	<code>bif0 Bif Reg</code>	Wywołuje wbudowaną funkcję <code>Bif/0</code> . Wynik zapisywany jest w rejestrze <code>Reg</code> .	✗
0A	10	<code>bif1 Bif Arg Reg</code>	Wywołuje wbudowaną funkcję <code>Bif/1</code> z argumentem <code>Arg</code> . Wynik zapisywany jest w rejestrze <code>Reg</code> .	✗
0B	11	<code>bif2 Bif Arg1 Arg2 Reg</code>	Wywołuje wbudowaną funkcję <code>Bif/2</code> z argumentami <code>Arg1</code> , <code>Arg2</code> . Wynik zapisywany jest w rejestrze <code>Reg</code> .	✗
0C	12	<code>allocate StackN Live</code>	Alokuje miejsce dla <code>StackN</code> słów na stosie. Używanych jest <code>Live</code> rejestrów X , gdyby w trakcie alokacji konieczne było uruchomienie <i>garbage collector</i> . Zapisuje CP na stosie.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
0D	13	allocate_heap StackN HeapN Live	Alokuje miejsce dla StackN słów na stosie. Upewnia się że na sterpie jest HeapN wolnych słów. Używanych jest Live rejestrów X , gdyby w trakcie alokacji konieczne było uruchomienie <i>garbage collector</i> . Zapisuje CP na stosie.	X
0E	14	allocate_zero StackN Live	Tak jak allocate/2, ale zaalokowana pamięć jest wyzerowana.	✓
0F	15	allocate_heap_zero SN HN L	Tak jak allocate_heap/3, ale zaalokowana pamięć jest wyzerowana.	X
10	16	test_heap HN L	Upewnia się że na sterpie jest HN wolnych słów. Używanych jest L rejestrów X , gdyby w trakcie konieczne było uruchomienie <i>garbage collector</i> .	X
11	17	init N	Zeruje N-te słowo na stosie.	X
12	18	deallocate N	Przywraca CP ze stosu i dealokuje N+1 słów ze stosu.	✓
13	19	return	Wraca do adresu zapisanego we wskaźniku CP .	✓
14	20	send	Wysyła wiadomość z rejestru X1 do procesu w rejestrze X0 .	X
15	21	remove_message	Usuwa aktualną wiadomość z kolejki wiadomości. Zapisuje wskaźnik do niej w rejestrze X0 . Usuwa aktywne przeterminowanie (<i>timeout</i>).	X
16	22	timeout	Resetuje wskaźnik SAVE . Czyści flagę przeterminowania.	X
17	23	loop_rec Lbl Src	Zapisuje kolejną wiadomość w kolejce wiadomości Src w rejestrze R0 . Jeśli jest pusta wykonuje skok do etykiety Lbl.	X
18	24	loop_rec_end Lbl	Ustawia wskaźnik SAVE na kolejną wiadomość w kolejce wiadomości i wykonuje skok do etykiety Lbl.	X

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
19	25	wait Lbl	Zawiesza proces aż do otrzymania wiadomości, który zostanie wznowiony na początku bloku receive w etykiecie Lbl.	✗
1A	26	wait_timeout Lbl T	Zawiesza proces jak wait. Ustawia przeterminowanie T i zapisuje następną instrukcję, która zostanie wykonana jeśli przeterminowanie się zrealizuje.	✗
27	39	is_lt Lbl Arg1 Arg2	Porównuje Arg1 z Arg2 i wykonuje skok do Lbl jeśli Arg1 jest większe lub równe od Arg2.	✗
28	40	is_ge Lbl Arg1 Arg2	Porównuje Arg1 z Arg2 i wykonuje skok do Lbl jeśli Arg1 jest mniejsze Arg2.	✗
29	41	is_eq Lbl Arg1 Arg2	Porównuje Arg1 z Arg2 i wykonuje skok do Lbl jeśli Arg1 jest arytmetycznie różne od Arg2.	✗
2A	42	is_ne Lbl Arg1 Arg2	Porównuje Arg1 z Arg2 i wykonuje skok do Lbl jeśli Arg1 jest arytmetycznie równe Arg2.	✗
2B	43	is_eq_exact Lbl Arg1 Arg2	Porównuje Arg1 z Arg2 i wykonuje skok do Lbl jeśli Arg1 jest różne Arg2.	✓
2C	44	is_ne_exact Lbl Arg1 Arg2	Porównuje Arg1 z Arg2 i wykonuje skok do Lbl jeśli Arg1 jest równe Arg2.	✗
2D	45	is_integer Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on liczbą całkowitą.	✗
2E	46	is_float Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on liczbą rzeczywistą.	✗
2F	47	is_number Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on liczbą.	✗
30	48	is_atom Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on atomem.	✗
31	49	is_pid Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on identyfikatorem procesu.	✗
32	50	is_reference Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on referencją.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
33	51	is_port Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on portem.	✗
34	52	is_nil Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on zerem (nil).	✗
35	53	is_binary Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on zmienną binarną.	✗
37	55	is_list Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on ani listą ani zerem.	✗
38	56	is_nonempty_list Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on niepustą listą.	✗
39	57	is_tuple Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on krotką.	✗
3A	58	test_arity Lbl Arg1 Arity	Sprawdza arność krotki Arg1 i skacze do Lbl jeśli nie jest ona równa Arity.	✗
3B	59	select_val Arg Lbl Dest	Skacze do etykiety Dest [Arg]. Jeśli nie istnieje skacze do Lbl.	✗
3C	60	select_tuple_arity Tuple Lbl Dest	Sprawdza arność krotki Tuple i skacze do etykiety Dest [Arity]. Jeśli etykieta nie istnieje skacze do Lbl.	✗
3D	61	jump Lbl	Skacze do etykiety Lbl.	✓
3E	62	catch Dest Lbl	Tworzy nowy blok catch. Zapisuje etykietę Lbl na Dest miejscu na stosie.	✗
3F	63	catch_end Dest	Kończy blok catch. Wymazuje etykietę na miejscu Dest na stosie.	✗
40	64	move Src Dest	Przenosi wartość z Src do rejestru Dest.	✓
41	65	get_list Src Hd Tail	Umieszcza głowę listy Src w rejestrze Hd i jej ogon w rejestrze Tail.	✗
42	66	get_tuple_element Src Elem Dest	Umieszcza element Elem krotki Src w rejestrze Dest.	✗
43	67	get_tuple_element Elem Tuple Pos	Umieszcza element Elem w krotce Tuple na pozycji Pos.	✗
45	69	put_list Hd Tail Dest	Tworzy komórkę listy [Hd Tail] na szczycie sterty i umieszcza ją w rejestrze Dest.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
46	70	put_tuple Dest Arity	Tworzy krotkę o arności Arity na szczycie sterty i umieszcza ją w rejestrze Dest.	✗
47	71	put Arg	Umieszcza Arg na szczycie stosu.	✗
48	72	badmatch Arg	Rzuca wyjątek badmatch z argumentem Arg.	✗
49	73	if_end	Rzuca wyjątek if_clause.	✗
4A	74	case_end Arg	Rzuca wyjątek case_clause z argumentem Arg.	✗
4B	75	call_fun Arity	Woła obiekt funkcyjny o arności Arity. Zakłada, że argumenty znajdują się w rejestrach X0...X(Arity-1) , a lambda w rejestrze X(Arity) . Zapisuje następną instrukcję we wskaźniku CP .	✗
4D	77	is_function Lbl Arg1	Sprawdza typ argumentu Arg1 i skacze do Lbl jeśli nie jest on funkcją.	✗
4E	78	call_ext_only Arity Lbl	Wywołuje rekurencyjną ogonowo zewnętrzną funkcję o arności Arity mającą indeks Lbl w tablicy funkcji zewnętrznych. Nie zapisuje adresu powrotu.	✓
59	89	bs_put_integer Fail Size Unit Flags Src	Umieszcza liczbę całkowitą w utworzonym wcześniej kontekście zmiennej binarnej, zajmując w nim rozmiar Size w jednostkach Unit. Źródłowa liczba znajduje się w rejestrze Src. Flagi dotyczące sposobu umieszczenia zmiennej w kontekście znajdują się w liczbie całkowitej Flags. Adres skoku Fail jest nieużywany.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
5A	90	bs_put_binary Fail Size Unit Flags Src	Umieszcza zmienną binarną w utworzonym wcześniej kontekście zmiennej binarnej, zajmując w nim rozmiar Size w jednostkach Unit. Źródłowa zmienna znajduje się w rejestrze Src. Flagi dotyczące sposobu umieszczenia zmiennej w kontekście znajdują się w liczbie całkowitej Flags. Adres skoku Fail jest nieużywany.	✗
5B	91	bs_put_float Fail Size Unit Flags Src	Umieszcza liczbę zmiennoprzecinkową w utworzonym wcześniej kontekście zmiennej binarnej, zajmując w nim rozmiar Size w jednostkach Unit. Źródłowa liczba znajduje się w rejestrze Src. Flagi dotyczące sposobu umieszczenia liczby zmiennoprzecinkowej w kontekście znajdują się w liczbie całkowitej Flags. Adres skoku Fail jest nieużywany.	✗
5C	92	bs_put_string Size Bytes	Bezpośrednio umieszcza bajty w utworzonym wcześniej kontekście zmiennej binarnej. Źródłowe bajty wskazywane są przez wskaźnik Bytes a ich liczba do przekopowania to Size.	✗
5E	94	fclearerror	Czyści flagę błędu zmiennoprzecinkowego, jeśli jest ustawiona.	✗
5F	95	fcheckerror Arg0	Sprawdza czy Arg0 zawiera wartość NaN lub nieskończoność. Jeśli tak, rzuca wyjątek badarith.	✗
60	96	fmove Arg0 Arg1	Kopiuje wartość Arg0 do Arg1.	✗
61	97	fconv Arg0 Arg1	Konwertuje wartość spod Arg0 na liczbę zmiennoprzecinkową i umieszcza ją w rejestrze Arg1.	✗
62	98	fadd Arg0 Arg1 Arg2 Arg3	Zapisuje w rejestrze Arg3 wynik dodawania Arg1 do Arg2. Argument Arg0 jest nieużywany.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
63	99	fsub Arg0 Arg1 Arg2 Arg3	Zapisuje w rejestrze Arg3 wynik odejmowania Arg2 od Arg1. Argument Arg0 jest nieużywany.	✗
64	100	fmul Arg0 Arg1 Arg2 Arg3	Zapisuje w rejestrze Arg3 wynik mnożenia Arg1 przez Arg2. Argument Arg0 jest nieużywany.	✗
65	101	fdiv Arg0 Arg1 Arg2 Arg3	Zapisuje w rejestrze Arg3 wynik dzielenia Arg1 przez Arg2. Argument Arg0 jest nieużywany.	✗
66	102	fnegate Arg0 Arg1 Arg2	Zapisuje ujemną wartość z rejestru Arg1 w rejestrze Arg2. Argument Arg0 jest nieużywany.	✗
67	103	make_fun2 N	Odczytuje wpis o indeksie N w tablicy lambd modułu i umieszcza go w rejestrze X0.	✗
68	104	try Dest Label	Tak jak instrukcja catch/2.	✗
69	105	try_end Dest	Kończy blok catch. Wymazuje etykietę na miejscu Dest na stosie. Jeśli nie ma zapisanej wartości w rejestrze R0 oznacza to że w bloku catch złapano wyjątek. W takim przypadku dokonywane jest przepisanie wartości rejestrów: R0 = X1, X1 = X2 i X2 = X3.	✗
6A	106	try_case Dest	Jak instrukcja try_end/1.	✗
6B	107	try_case_end Reason	Rzuca wyjątek try_clause z argumentem Reason.	✗
6C	108	raise Stacktrace Reason	Rzuca wyjątek Reason ze stosem wywołań Stacktrace.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
6D	109	bs_init2 Fail Sz Words Regs Flags Dst	Inicjalizuje miejsce dla zmiennej binarnej o rozmiarze Sz. Zapewnia, że oprócz tego rozmiaru po inicjalizacji dostępne będzie dodatkowo Words słów maszynowych. Regs oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie inicjalizacji konieczne było uruchomienie <i>garbage collector</i> a. Adres skoku Fail oraz opcje Flags są aktualnie nieużywane.	✗
6F	111	bs_add Fail S1 S2 Unit Dst	Oblicza sumę bitów w S1 i liczby jednostek Unit w S2. Wynik przechowuje w Dst. W przypadku niepowodzenia wykonuje skok do etykiety Fail.	✗
70	112	apply N	Znajduje adres początku funkcji zapisanej w rejestrze X(N+1) , w module zapisanym w rejestrze X(N) o arności N i skacze do tego adresu. Zapisuje następną instrukcję we wskaźniku CP .	✗
71	113	apply_last N Dea	Skacze do zewnętrznej funkcji tak jak instrukcja apply/1. Ściąga wartość wskaźnika CP ze stosu. Zwalnia Dea miejsc na szczycie stosu.	✗
72	114	is_boolean/2	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on ani atomem true ani false.	✗
73	115	is_function2 Lbl Arg1 Arity	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on funkcją o arności Arity.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
74	116	bs_start_match2 Fail Bin X Y Dst	Sprawdza czy Bin jest kontekstem zmiennej binarnej z odpowiednią liczbą miejsc do zapisania tymczasowych przesunięć bitowych (<i>offsetów</i>) określonych przez liczbę Y. Jeśli aktualna liczba miejsc jest za mała tworzony jest nowy kontekst porównywania zmiennej binarnej z odpowiednią liczbą miejsc, który zostanie zapisany w miejscu Dst. W przypadku niepowodzenia dokonany zostanie skok do etykiety Fail. X oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie operacji konieczne było uruchomienie <i>garbage collector</i> a.	✗
75	117	bs_get_integer2 Fail Ms Live Sz Unit Flags Dst	Pobiera liczbę całkowitą z kontekstu zmiennej binarnej Ms o rozmiarze Sz w jednostkach Unit. Wynik zapisywany jest do Dst. Opcje operacji zapisywane są w liczbie całkowitej Flags. W przypadku niepowodzenia wykonywany jest skok do etykiety Fail. Live oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie operacji konieczne było uruchomienie <i>garbage collector</i> a.	✗
76	118	bs_get_float2 Fail Ms Live Sz Unit Flags Dst	Pobiera liczbę zmiennoprzecinkową z kontekstu zmiennej binarnej Ms o rozmiarze Sz w jednostkach Unit. Wynik zapisywany jest do Dst. Opcje operacji zapisywane są w liczbie całkowitej Flags. W przypadku niepowodzenia wykonywany jest skok do etykiety Fail. Live oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie operacji konieczne było uruchomienie <i>garbage collector</i> a.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
77	119	bs_get_binary2 Fail Ms Live Sz Unit Flags Dst	Pobiera zmienną binarną z kontekstu zmiennej binarnej Ms o rozmiarze Sz w jednostkach Unit. Wynik zapisywany jest do Dst. Opcje operacji zapisywane są w liczbie całkowitej Flags. W przypadku niepowodzenia wykonywany jest skok do etykiety Fail. Live oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie operacji konieczne było uruchomienie <i>garbage collector</i> .	✗
78	120	bs_skip_bits2 Fail Ms Sz Unit Flags	Pomija Sz jednostek wyrażonych w Unit z kontekstu zmiennej binarnej Ms. Opcje operacji wyrażone są za pomocą liczby całkowitej Flags. W przypadku niepowodzenia wykonywany jest skok do etykiety Fail.	✗
79	121	bs_test_tail2 Fail Ms Bits	Sprawdza, czy kontekst zmiennej binarnej Ms ma jeszcze dokładnie Bits niedopasowanych bitów. Wykonuje skok do etykiety Fail jeżeli tak nie jest.	✗
7A	122	bs_save2 Reg Index	Zapisuje aktualne przesunięcie bitowe <i>offset</i> z kontekstu zmiennej binarnej zawartego w rejestrze Reg i zapisuje do jego tablicy <i>offsetów</i> .	✗
7B	123	bs_restore2 Reg Index	Odczytuje przesunięcie bitowe kontekstu zmiennej binarnej zapisanego w rejestrze Reg z indeksu Index. Zapisuje go jako aktualne dla tego kontekstu.	✗
7C	124	gc_bif1 Lbl Live Bif Arg1 Reg	Wywołuje funkcję wbudowaną Bif/1 z argumentem Arg1. Wynik zapisuje w rejestrze Reg. W przypadku niepowodzenia skacze do etykiety Lbl. Uruchamia <i>garbage collector</i> jeśli jest to konieczne, zachowując Live rejestrów X.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
7D	125	gc_bif2 Lbl Live Bif Arg1 Arg2 Reg	Wywołuje funkcję wbudowaną Bif/2 z argumentami Arg1, Arg2. Wynik zapisuje w rejestrze Reg. W przypadku niepowodzenia skacze do etykiety Lbl. Uruchamia <i>garbage collector</i> jeśli jest to konieczne, zachowując Live rejestrów X .	✓
81	129	is_bitstr Lbl Arg1	Sprawdza typ Arg1 i skacze do Lbl jeśli nie jest on ciągiem bitów.	✗
82	130	bs_context_to_binary Reg	Zamienia kontekst zmiennej binarnej znajdującej się w rejestrze Reg na właściwą zmienną binarną.	✗
83	131	bs_test_unit Fail Ms Unit	Sprawdza czy rozmiar niedopasowanego jeszcze fragmentu kontekstu zmiennej binarnej Ms jest podzielny przez Unit. Jeżeli nie, wykonywany jest skok do etykiety Fail.	✗
84	132	bs_match_string Fail Ms Bits Val	Dokonyuje porównania Bits bitów, począwszy od miejsca wskazywanego przez wskaźnik Val z kontekstem zmiennej binarnej Ms. Jeżeli porównywane wartości nie są równe dokonywany jest skok do etykiety Fail.	✗
85	133	bs_init_writable	Alokuje miejsce o rozmiarze R0 na stercie procesu. Tworzy w zaalokowanym miejscu strukturę zmiennej binarnej. Dodatkowo tworzy wskaźnik do utworzonej struktury, który umieszczony zostanie w rejestrze R0 .	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
86	134	bs_append Fail Size Extra Live Unit Bin Flags Dst	Dopisuje Size jednostek Unit do zmiennej binarnej Bin i zapisuje wynik do Dst. Jeśli nie ma wystarczająco dużej ilości miejsca, tworzona jest nowa struktura na sterckie z odpowiednią ilością miejsca, powiększona dodatkowo o Extra słów maszynowych. W przypadku niepowodzenia wykonywany jest skok do etykiety Fail. Opcje operacji zapisane są w liczbie całkowitej Flags. Live oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie alokacji konieczne było uruchomienie <i>garbage collector</i> .	✗
87	135	bs_private_append Fail Size Unit Bin Flags Dst	Operacja ma działanie podobne do operacji bs_append/8 jednak w przypadku zbyt małej ilości miejsca dokonuje realokacji aktualnej zmiennej.	✗
88	136	trim N Remaining	Redukuje stos o N słów, zachowując CP na jego szczycie.	✗
89	137	bs_init_bits Fail Sz Words Regs Flags Dst	Alokuje zmienną binarną na sterckie o rozmiarze Sz bitów. Jeżeli rozmiar nie jest podzielny przez 8, tworzony jest wskaźnik na strukturę z zapisaną ilością zajmowanych bitów. Wynik operacji zapisany jest do Dst. Upewnia się że na sterckie jest dodatkowo Words słów maszynowych możliwych do zaalokowania. Opcje operacji zapisane są w liczbie całkowitej Flags. W razie niepowodzenia wykonywany jest skok do etykiety Fail. Regs oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie alokacji konieczne było uruchomienie <i>garbage collector</i> .	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
8A	138	bs_get_utf8 Fail Ms Arg2 Arg3 Dst	Pobiera znak zapisany w UTF-8 z kontekstu zmiennej binarnej Ms i zapisuje go do Dst. W przypadku niepowodzenia wykonuje skok do etykiety Fail. Argumenty Arg2 oraz Arg3 są aktualnie nieużywane.	✗
8B	139	bs_skip_utf8 Fail Ms Arg2 Arg3	Pomija znak zakodowany w UTF-8 w kontekście zmiennej binarnej Ms. W przypadku niepowodzenia wykonuje skok do etykiety Fail. Argumenty Arg2 oraz Arg3 są aktualnie nieużywane.	✗
8C	140	bs_get_utf16 Fail Ms Arg2 Flags Dst	Pobiera znak zapisany w UTF-16 z kontekstu zmiennej binarnej Ms, używając opcji zapisanych w liczbie całkowitej Flags i zapisuje go do Dst. W przypadku niepowodzenia wykonuje skok do etykiety Fail. Argument Arg2 jest aktualnie nieużywany.	✗
8D	141	bs_skip_utf16 Fail Ms Arg2 Flags	Pomija znak zakodowany w UTF-16 w kontekście zmiennej binarnej Ms, używając opcji zapisanych w liczbie całkowitej Flags. W przypadku niepowodzenia wykonuje skok do etykiety Fail. Argument Arg2 jest aktualnie nieużywany.	✗
8E	142	bs_get_utf32 Fail Ms Live Flags Dst	Pobiera znak zapisany w UTF-32 z kontekstu zmiennej binarnej Ms, używając opcji zapisanych w liczbie całkowitej Flags i zapisuje go do Dst. Live oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie operacji konieczne było uruchomienie <i>garbage collector</i> . W przypadku niepowodzenia wykonuje skok do etykiety Fail.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
8F	143	bs_skip_utf32 Fail Ms Live Flags	Pomija znak zakodowany w UTF-32 w kontekście zmiennej binarnej Ms, używając opcji zapisanych w liczbie całkowitej Flags. Live oznacza liczbę aktywnych rejestrów, w razie gdyby w trakcie operacji konieczne było uruchomienie <i>garbage collector</i> . W przypadku niepowodzenia wykonuje skok do etykiety Fail.	✗
90	144	bs_utf8_size Fail Literal Dst	Oblicza liczbę bajtów koniecznych do zapisania Literal w UTF-8 i zapisuje wynik do Dst. Argument Fail jest aktualnie nieużywany.	✗
91	145	bs_put_utf8 Fail Flags Src	Umieszcza znak zakodowany w UTF-8 znajdujący się w Src w aktualnym kontekście zmiennej binarnej. Argumenty Fail i Flags są aktualnie nieużywane.	✗
92	146	bs_utf16_size Fail Literal Dst	Oblicza liczbę bajtów koniecznych do zapisania Literal w UTF-16 i zapisuje wynik do Dst. Argument Fail jest aktualnie nieużywany.	✗
93	147	bs_put_utf16 Fail Flags Literal	Umieszcza znak zakodowany w UTF-16 znajdujący się w Src w aktualnym kontekście zmiennej binarnej z użyciem opcji zapisanych w liczbie całkowitej Flags. Argument Fail jest aktualnie nieużywany.	✗
94	148	bs_put_utf32 Fail Flags Literal	Umieszcza znak zakodowany w UTF-32 znajdujący się w Src w aktualnym kontekście zmiennej binarnej z użyciem opcji zapisanych w liczbie całkowitej Flags. W przypadku niepowodzenia wykonuje skok do etykiety Fail.	✗
95	149	on_load	Oznacza kod wykonywany przy ładowaniu modułu.	✗

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi	Jest?
hex	dec			
96	150	recv_mark Lbl	Zapamiętuje aktualną wiadomość z kolejki oraz etykietę Label do instrukcji loop_rec/2.	X
97	151	recv_set Lbl	Jeśli etykieta Lbl wskazuje na instrukcję loop_rec/2 to przepisuje wiadomość zachowaną przez instrukcję recv_mark do wskaźnika SAVE .	X
98	152	gc_bif3 Lbl Live Bif Arg1 Arg2 Arg3 Reg	Wywołuje funkcję wbudowaną Bif/3 z argumentami Arg1, Arg2, Arg3. Wynik zapisuje w rejestrze Reg. W przypadku niepowodzenia skacze do etykiety Lbl. Uruchamia <i>garbage collector</i> jeśli jest to konieczne, zachowując Live rejestrów X .	X
99	153	line N	Znakuje aktualne miejsce jako linia o indeksie N w tablicy linii.	X

Tablica A.2: Lista operacji maszyny wirtualnej BEAM

Bibliografia

- [1] Ericsson AB. Erlang Embedded Systems User's Guide, 1997.
- [2] Ericsson AB. Distributed Erlang. http://www.erlang.org/doc/reference_manual/distributed.html, 2014. [data dostępu: 21.03.2014].
- [3] Ericsson AB. Erlang External Term Format. http://erlang.org/doc/apps/erts/erl_ext_dist.html, 2014. [data dostępu: 21.03.2014].
- [4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
- [5] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, KTH, 2003.
- [6] Richard Barry. *Using the FreeRTOS Real Time Kernel*. Real Time Engineers Ltd., 2011.
- [7] HopeRF Electronic. *RFM73 Datasheet*. Hope Microelectronics Co. Ltd., 2006.
- [8] Jim Gray. *Why Do Computers Stop And What Can Be Done About It?*, 1985.
- [9] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI'73*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
- [10] Maxim Kharchenko. Erlang on Xen, A quest to lower startup latency. *Erlang Factory SF Bay Area 2012, San Francisco*, 2012.
- [11] Erlang Solutions Ltd. Erlang Embedded. <http://www.erlang-embedded.com>, 2013. [data dostępu: 17.03.2014].
- [12] J. Morrison. EA IFF 85: Standard for interchange format files. *Amiga ROM Kernel Reference Manual: Devices (3rd edition)*, Addison-Wesley, 1(99):1, 1985.
- [13] NXP Semiconductors. *LPC1769/68/67/66/65/64/63. Product data sheet*. NXP Semiconductors, N.V., 2014.

- [14] NXP Semiconductors. LPCXpresso IDE. <http://www.lpcware.com/lpcxpresso/home>, 2014. [data dostępu: 3.07.2014].
- [15] Peer Stritzinger. Full Metal Erlang. *Erlang User Conference 2013, Stockholm*, 2013.