デュアルコア・プロセッサのためのDSPスクリプト言語の設計と実装

高橋清隆†

携帯電話に代表される携帯端末にMPUコアとDSPコアを１つのチップに集積したデュアルコア･プロセッサが利用され始めている．デュアルコア･プロセッサの採用は高性能化や低消費電力化に効果がある一方でMPU側とDSP側で別々のソフトウェアの開発環境が必要になるなど開発工程を煩雑にする．我々はデュアルコア・プロセッサにおけるDSPソフトウェアの開発環境を効率化するため，MPU側で動作するオペレーティングシステム上から直接DSPのプログラミングを可能にするDSPスクリプト言語とその処理系を提案する．本稿では，我々の提案するシステムのプロトタイプの設計，実装と評価について述べる．

Design and Implementation of a DSP scripting language for dual-core processor

Kiyotaka Takahashi†

Mobile terminals such as mobile phones are adopting a dual core processor having MPU core and DSP core inside. Dual core processor significantly benefits mobile terminals in terms of performance and power consumption. However it makes the software development more complicated because each MPU and DSP requires its own development process. In order to improve the efficiency of software development with dual core processor, we propose a DSP scripting language and its execution environment that enables direct DSP programming from MPU side. This paper describes the design, implementation, and evaluation of the prototype system.

1．はじめに

近年，携帯電話，PDAや家電などの組込みソフトウェアの規模が大きくなり，その複雑さが急速に増している．組込みソフトウェアの開発の効率化を図るため，複数の企業でLinuxを組込み機器向けのオペレーティングシステムとして採用する動きが出てきた．また，CE Linux Forumと言う複数の大手電気機器メーカーによる組込み用途でのLinuxの普及を目指した業界団体も設立されている．その一方で，携帯端末のプロセッサに対しては，高性能化・省電力化が求められている．特に，携帯端末上で動画などのマルチメディア処理のサポートが求められている．現在，これらの要求を実現するために，既存のMPU(Micro Processing Unit)に対してマルチメディア処理を可能にする命令セットを追加する手法や、MPUとマルチメディア処理の為の信号処理用にDSP(Digital Signal Processor)を１つのチップに集積しデュアルコア･プロセッサとする手法などがとられている．

デュアルコア･プロセッサの利用は，マルチメディア処理に適した高性能なDSPをMPU側から利用できると言う大きな利点がある．しかし，そのソフトウェア開発はMPUとDSPのそれぞれに開発環境を用意する必要があり，個々のソフトウェア・モジュールの開発から，それらモジュールのインテグレーションの段階まで，それぞれの開発環境が必要になる．MPUとDSPソフトウェアのインテグレーション作業中に障害が発生した場合でも，個々の開発環境に戻り該当個所を修正する必要があるなど開発工程が煩雑になりがちである．

また，一般にDSPのソフトウェアは，高速な処理を要求されるディジタル信号処理のアルゴリズムなど，その中核となる部分は現在もアセンブリ言語により記述されている事が多く，それらを繋ぎ合わせるグルー部分にC言語などの高級言語が利用されているのが現状である．加えて，デュアルコア･プロセッサ環境においては，DSP側のソフトウェアにディジタル信号処理のアルゴリズムと，それを利用するMPU側のソフトウェアとの通信を行うためのインターフェース部分を組込む必要がある．

本研究は，携帯端末向けデュアルコア･プロセッサにおいてMPU側から実行できるDSPスクリプト言語とその処理系を提案する．このスクリプト言語は，これまでのDSPソフトウェア開発においてC言語などの高級言語で記述していた部分を置き換え，MPU側ソフトウェアへの簡易的なインターフェースをサポートする事により，デュアルコア･プロセッサにおけるDSPソフトウェアの開発工程を効率化する事を目標とする．本稿では，現在開発を進めているDSP

† ノキア・ジャパン株式会社　ノキア・リサーチセンター
Nokia Research Center, Nokia Japan Co., Ltd.

スクリプト言語のプロトタイプの設計と実装について述べ，その評価結果について報告する．

2．既存のソフトウェア開発環境

本章では，デュアルコア・プロセッサにおける既存のソフトウェア開発環境とその問題点を述べる．

2．1　MPU と DSP の開発環境の分離

デュアルコア・プロセッサは１つのチップに MPU と DSP を集積しているので，2 つのソフトウェア開発環境が必要になる．さらに，組込み機器向けのソフトウェア開発ではターゲットとなる MPU/DSP とは別の PC やワークステーション上にクロス開発環境を用意する必要がある．

2．2　ソフトウェア開発の工程が煩雑

一般にデュアルコア・プロセッサ上のソフトウェアは，DSP 側にはマルチメディア処理などのディジタル信号処理を行うための機能が実装され，MPU 側にはそれらの機能を利用するソフトウェアが実装される．現在，ソフトウェア開発の全ての工程において，それぞれの開発環境が必要になる．特に，MPU と DSP ソフトウェアの協調を実現するためのインターフェース部分の実装，また，それらのインテグレーション作業は，障害が発見される毎にそれぞれの開発環境を利用する必要があるなど，作業が煩雑になりがちである．

2．3　DSP ソフトウェア開発環境は高価

DSP のソフトウェアはリアルタイム処理を実現する必要があるために，様々な解析機能などがサポートされている専用の開発環境が用意されている．しかし，この様な開発環境は一般に高価であり，DSP ソフトウェアの開発を専業にしない者にとっては，手軽に購入し試用できる価格ではない事が多い．

3．DSP スクリプト言語の概要

本章では，我々が提案する DSP スクリプト言語のプロトタイプの言語仕様とその処理系について説明する．

3．1　言語仕様

言語仕様は，プログラマが短時間で学習できるように，既存の C，Pascal などを基に設計した．

定数

数字の列からなる 10 進数の整数である．16 進数・8 進数・2 進数の定数は扱わない．また，浮動小数点定数や文字定数なども扱わない．

識別子

英数文字列であり，英文字で始まらなければならない．英文字の大文字と小文字は区別される．変数を表す文字列の長さに制限はない．

予約語

下記のものが予約語として使われる．

```
do     else    endif   endwhile
if     main    read    then
var    while   write
```

型と演算子

以下に型と演算子をあげる．演算子の優先順位および結合規則は C 言語と同じである．

表 1：型と演算子

型	整数型 (16[bits])
	整数型の配列
型指定子	var
代入演算子	:=
算術演算子	+, -, *, /, %
関係演算子	==, !=, <, >, <=, >=
論理否定演算子	!
単項マイナス演算子	-
配列参照	[n] (n=0,…)

変数

変数は識別子により参照される．変数には 16[bits] 表現の整数値が格納される．全ての変数はスタック上に割り当てられる．

文

文は，識別子・演算子・定数から構成される式とセミコロンから構成される．セミコロンは文の終端を示す．

選択文

選択文として if 文をサポートする．if 文の構文は以下の通りである．

```
if 式 then 文 endif;
if 式 then 文 else 文 endif;
```

繰り返し文

繰り返し文として while 文をサポートする．while 文の構文は以下の通りである．

```
while 式 do 文 endwhile;
```

ジャンプ文

制御の流れを無条件に変えるジャンプ文はサポートしない．

3．2　言語処理系

提案する言語処理系はデュアルコア・プロセッサの特徴を生かし，スクリプト言語の中間言語への翻訳は汎用プロセッサである MPU 側で行い，DSP 側ではディジタル信号処理に関連した処理のみを行うように設計した．図 1 に示す様に各コンポーネントを配置して負荷の分散と処理の効率化を図る．

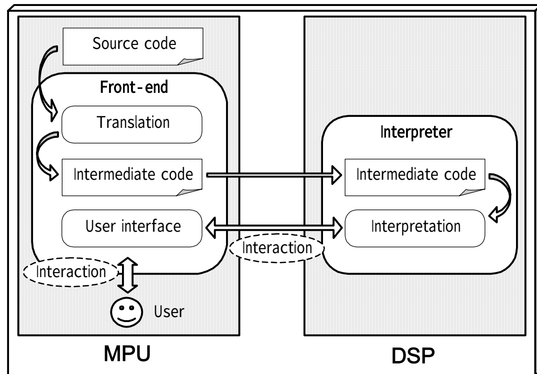

図 1：各コンポーネントの構成

以下に示す手順により処理が行われる.

1. ユーザは MPU 側で提供されるフロントエンドに対してソースプログラムを与え実行を開始する.
2. ソースプログラムは MPU において中間言語に翻訳され,DSP 側に転送される.
3. DSP 側でインタプリタが中間コードを実行する.
4. ユーザは必要に応じて,MPU 側のフロントエンドを介して DSP 側のインタプリタと通信を行う.

4．プロトタイプの実装

本研究で提案した DSP スクリプト言語の処理系を，TI 社製デュアルコア･プロセッサの OMAP1510 上に構築した. OMAP1510 は ARM925 コアと DSP の C5510 コアを持つ.

4．1　ソフトウェア環境

本プロトタイプの OMAP1510 プロセッサ上では表 2 に示したオペレーティングシステムが動作している.

表 2：オペレーティングシステム

プロセッサ	OS
ARM925	Linux
C5510	DSP/BIOS

DSP/BIOS は TI 社が提供する DSP 用のリアルタイム OS である．また，ARM と DSP 間の通信を実現するために，弊社で開発を進めている DSP Gateway を使用した. これは, ARM で動作する Linux 上のソフトウェアと，DSP で動作する DSP/BIOS 上のソフトウェアとの通信を可能にするソフトウェアである.

4．2　中間言語の仕様

中間言語の仕様は, ARM 側でスクリプト言語から中間言語へ翻訳し DSP 側へ転送され, DSP 側のインタプリタにより実行されるので，最適化のしやすさを考慮し C5510 で使われているアセンブリ言語をベースにした．ターゲットとなる DSP のアセンブリ言語に近づける事により，異なるターゲットへの移植性は低くなるが，今後 JIT コンパイラ(Just-In-Time Compiler)の開発を容易にするねらいがある．表 3 に，この中間言語でサポートするレジスタの種類を示す.

表 3：レジスタの種類

レジスタ名	種類
AR0/AR1	汎用レジスタ
TC0	条件レジスタ
SP	スタックポインタ
PC	プログラムカウンタ

表 4 に，この中間言語でサポートする各命令のそれらの書式と動作を示す.

表 4：各命令の構文

オペコード	オペランド	動作
MOV	MOV S, Rd	Rd = S
ADD	ADD S, Rd	Rd = Rd + S
SUB	SUB S, Rd	Rd = Rd - S
MPY	MPY S, Rd	Rd = Rd * S
DIV	DIV S. Rd	Rd = Rd / S
CMP	CMP S op D, Rd	Rd = S op D
BCC	BCC addr, [!]Rd	if ([!]Rd) goto addr
B	B addr	goto addr
READ	READ Rd	Rd <= MPU
WRITE	WRITE S	MPU <= S
HALT	HALT	Terminate

ただし，Rd,S,D,op は以下の通りである.

<table>
<tr><th colspan="2">記号</th><th>意味</th></tr>
<tr><td rowspan="4">Rd</td><td>Rn</td><td>レジスタ Rn に格納されている値</td></tr>
<tr><td>*Rn</td><td>レジスタ Rn に格納されているアドレスが参照する値</td></tr>
<tr><td>*Rn(#K)</td><td>レジスタ Rn に格納されているアドレスからオフセット#K 分ずれたアドレスが参照する値</td></tr>
<tr><td>*Rn(Rm)</td><td>レジスタ Rn に格納されているアドレスからレジスタ Rm に格納されているオフセット分ずれたアドレスが参照する値</td></tr>
<tr><td colspan="2">S, D</td><td>Rd 含む#K</td></tr>
<tr><td colspan="2">op</td><td>関係演算子</td></tr>
<tr><td colspan="2">#K</td><td>16[bits]整数定数</td></tr>
</table>

４．３　コンパイラ

本提案方式の処理系では，OMAP プロセッサの MPU である ARM プロセッサで，DSP スクリプト言語のソースコードから中間言語への翻訳を行う．このコンパイラは以下のような基本的な機能からなる．

- 字句解析
- 構文解析
- コード生成

字句解析ルーチンの作成には GNU の Flex を用い，構文解析ルーチンの作成には GNU の Bison を用いた． ソースコードの解析から 1 パスで中間コードの生成まで行い，続いてジャンプ命令の飛び先アドレスを解決する．生成される中間コードは，前述した中間言語の各命令が 8[Bytes]長のバイナリにコード化されたものである．今回実装したコンパイラのプロトタイプでは，中間コード生成に関する最適化処理は組み込んでいない。

このコンパイラは，Linux 上で動作する DSP スクリプト言語のフロントエンドとして実装されている．これには，ARM 側で翻訳し生成された中間コードを DSP 側に転送する機能と，DSP 側のインタプリタで実行される中間コードの入出力を扱う機能が含まれている．ユーザは，このフロントエンドに DSP スクリプト言語のソースコードを与える事により，それを実行する事が可能である．

４．４　インタプリタ

本提案方式の処理系では，OMAP プロセッサの DSP 側で中間言語のインタプリタが動作する．DSP 側で動作するソフトウェアは，ARM プロセッサ側から中間コードの受信し，DSP 側のメモリに配置する転送部分と，その中間コードを実際に実行するインタプリタ部分とからなる．

転送部分は，ARM プロセッサ側でコード化された中間コードを受け取り，事前に確保された DSP 側のメモリ空間にそれらを配置する．そして，インタプリタ内部のレジスタを初期化する．PC は，中間コードが配置されたメモリ空間の先頭を示すように初期化される．SP は，事前に確保されたスタック用メモリ空間を示すように初期化される．

インタプリタ部分は，PC が指す番地から 8[Bytes]を 1 つの中間コードと解釈し，それを実行する．1 つの中間コードを実行すると，PC を次の中間コードを指すように 8 バイト増やし実行を続ける．また，中間コードがジャンプ命令の場合は，該当アドレスを PC に設定し実行を続ける．インタプリタ部分は，中間コードとして HALT 命令を実行すると，その実行を終了する．

５　評価

本章では，本提案方式の処理系として構築したプロトタイプの評価について述べる．

５．１　基本動作

今回構築したプロトタイプの基本動作を確認する．まず，図 2 に示すサンプルコードを OMAP1510 プロセッサ上で動作させる．このプログラムは，入力された個数のフィボナッチ数列を出力するプログラムである．

```
#!/root/dspsh
main
{
    var n, f[2], h, l;

    n := 0;
    f[0] := 1;
    f[1] := 1;

    read l;

    while n < l do
        h := f[0] + f[1];
        write h;
        f[0] := f[1];
        f[1] := h;
        n := n + 1;
    endwhile
}
```

図 2：サンプルコード

前述のサンプルコードを，ARM で動作する Linux 上からフロントエンドを通して，DSP 側のインタプリタで実行し出力結果を得た．これにより，本プロトタイプが一連の動作，つまり，ARM 側でソースコードから中間コードの生成，それを DSP 側に送り，DSP 側のインタプリタでの実行，ARM 側のフロントエンドから DSP 側での実行結果を得られる事を確認した．前述のサンプルコードを実行したときの様子を図 3 に示す．

```
# echo 10 | dspsh fib
2
3
5
8
13
21
34
55
89
144
#
```

図 3：サンプルコードの実行

５．２　中間コード生成

生成される中間コードを基にプロトタイプとして

実装したコンパイラの評価を行う．まず，評価対象を図2に示すサンプルコードとする． プロトタイプとして構築したコンパイラにより生成される中間コードを図 4 に示す．

```
_main:
    MOV #0, *SP(#0)
    MOV #1, *SP(#1 + #0)
    MOV #1, *SP(#1 + #1)
    READ *SP(#4)
_while_0:
    MOV *SP(#0), AR0
    MOV AR0, *SP(#5)
    MOV *SP(#4), AR0
    MOV AR0, *SP(#6)
    CMP *SP(#5) < *SP(#6), TC0
    BCC _while_end_0, !TC0
    MOV #1, AR1
    ADD #0, AR1
    MOV *SP(AR1), AR0
    MOV AR0, *SP(#7)
    MOV #1, AR1
    ADD #1, AR1
    MOV *SP(AR1), AR0
    MOV AR0, *SP(#8)
    MOV *SP(#7), AR0
    ADD *SP(#8), AR0
    MOV AR0, *SP(#9)
    MOV *SP(#9), AR0
    MOV AR0, *SP(#3)
    MOV *SP(#3), AR0
    MOV AR0, *SP(#10)
    WRITE *SP(#10)
    MOV #1, AR1
    ADD #1, AR1
    MOV *SP(AR1), AR0
    MOV AR0, *SP(#11)
    MOV *SP(#11), AR0
    MOV AR0, *SP(#1 + #0)
    MOV *SP(#3), AR0
    MOV AR0, *SP(#12)
    MOV *SP(#12), AR0
    MOV AR0, *SP(#1 + #1)
    MOV *SP(#0), AR0
    MOV AR0, *SP(#13)
    MOV *SP(#13), AR0
    ADD #1, AR0
    MOV AR0, *SP(#14)
    MOV *SP(#14), AR0
    MOV AR0, *SP(#0)
    B _while_0
_while_end_0:
    HALT
```

図 4：サンプルコードに対する中間コード

ここで，比較対象としてサンプルコードと同等な C 言語で記述したプログラムを用意する．これを図5に示す．ただし，ARM と DSP 間でデータのやり取りを行う READ，WRITE に相当する命令は C 言語レベルではサポートしていないので，それらは除外した．

```
fib(void)
{
    short n, f[2], h, l;

    n = 0;
    f[0] = 1;
    f[1] = 1;

    while (n < l) {
        h = f[0] + f[1];
        f[0] = f[1];
        f[1] = h;
        n = n + 1;
    }
}
```

図 5：C 言語によるサンプルコード

このC言語で記述されたサンプルコードを，実際のDSPソフトウェア開発環境で提供されるCコンパイラを用い，出力させたアセンブリコードを図 6 に示す．ただし，このアセンブリコードはC 言語で 1 つの関数として記述されたコードから生成されたものなので，正しく比較するために関数呼び出しに関するスタックポインタの操作などの命令は削除した．

```
    MOV #0, *SP(#0) ; ¦5¦
    MOV #1, *SP(#1) ; ¦6¦
    MOV #1, *SP(#2) ; ¦7¦
    MOV *SP(#4), AR1 ; ¦9¦
    MOV *SP(#0), AR2 ; ¦9¦
    CMP AR2 >= AR1, TC1 ; ¦9¦
    ||      NOP
    BCC L2,TC1 ; ¦9¦
L1:
    MOV *SP(#2), AR1 ; ¦10¦
    ADD *SP(#1), AR1, AR1 ; ¦10¦
    MOV AR1, *SP(#3) ; ¦10¦
    MOV *SP(#2), AR1 ; ¦11¦
    MOV AR1, *SP(#1) ; ¦11¦
    MOV *SP(#3), AR1 ; ¦12¦
    MOV AR1, *SP(#2) ; ¦12¦
    ADD #1, *SP(#0) ; ¦13¦
    MOV *SP(#4), AR2 ; ¦14¦
    MOV *SP(#0), AR1 ; ¦14¦
    CMP AR1 < AR2, TC1 ; ¦14¦
    ||      NOP
    BCC L1,TC1 ; ¦14¦
L2:
```

図 6：C 言語サンプルコードのアセンブリコード

ここでは便宜上，本提案方式の処理系から生成された中間コードをコードA，C言語で記述された同等のサンプルコードから生成されたアセンブリコードをコード B とする．まずコード A とコード B の行数の比較を表 5 に示す．

表 5：行数の比較

	コードA	コードB
コード行数	42 行	19 行

次に，コード A とコード B 内で使用されている命令の出現回数（コード内での割合%）を比較し表6に示す．

表 6：命令の出現回数

命令	コードA	コードB
MOV	34 (81[%])	13 (68[%])
ADD	5 (12[%])	2 (11[%])
CMP	1 (2[%])	2 (11[%])
BCC	1 (2[%])	2 (11[%])
B	1 (2[%])	0 (0[%])

この結果から，本コンパイラの生成する中間コードと C 言語のコンパイラが生成するアセンブリコードとを比較すると，コードの行数が約 2 倍になっている事が分かる．また，生成された中間コードには多くの MOV 命令が使われている事が分かる．この事から，中間コードが増加した主な原因は MOV 命令の増加によるものと考えられる．

本コンパイラが多くの MOV 命令を生成してしまう原因は，変数を扱う際の一時変数の利用方法によるものと考えられる．図4に示す中間コードを見ると，冗長な MOV 命令の使用がいくつかの個所で見受けら

れる．したがって，一時変数の利用方法を改善する事により，既存のC言語コンパイラと同程度の効率で中間コードを生成できる事が期待される．

５．３　インタプリタの実行効率

プロトタイプとして構築したDSP側で動作するインタプリタの実行時の効率について評価を行う． DSPソフトウェア開発環境を用いて，インタプリタはC言語で記述され実装されており，このインタプリタのコンパイル時にアセンブリコードを生成する．このアセンブリコードの各中間コードを処理する個所を調査する事により，1つの中間コードがDSPで処理されるのに何命令実行しているかを解析した．その結果を表7に示す．

表7：1中間コードに対するアセンブリ数

中間コード	アセンブリ命令数
MOV	26
ADD	40
SUB	41
MPY	43
DIV	43
CMP	34
BCC	25
B	12

この結果から，1中間コードに対してDSPのアセンブリ命令を約20から40命令実行している事が分かる．例えば，サンプルコードを実行した場合の総アセンブリ・ステップ数を求めると1155ステップとなる．これは生成された中間コードの27.5倍である．一般に，スクリプト言語はC言語の様なプログラミング言語に比べ，実行スピードが10から20倍になると言われている事を考慮しても，本インタプリタの実装には改善の余地があることが分かった．

５．３　開発工程

ARM側のLinux上から，サンプルコードで示したソースコードなどを，DSP側のDSP/BIOS上で動作するインタプリタで簡単に実行し，その結果を得る事が出来た．

このようにARM側から直接DSPのプログラミングが可能になると，ソフトウェアの修正が必要なとき常にDSPソフトウェアの開発環境に戻る必要がなくなるので，DSPソフトウェアの開発効率が上がることが確認できた．

さらに，DSPスクリプト言語からDSP/BIOSのシステムコール呼出しや，他のDSPモジュールとの連携が可能になると，実際のDSPソフトウェア開発において，より高いレベルのプログラミングを可能にするスクリプト言語の特徴が生き，高い生産性を実現できる事が期待される．

６　まとめ

本稿では，デュアルコア・プロセッサにおけるDSPスクリプト言語を提案し，そのプロトタイプの実装に対して評価を行った．

デュアルコア・プロセッサにおいて，本稿で提案したDSPスクリプト言語を用いる事により，DSP側のソフトウェア開発を，直接MPU側から行う事が出来る事を確認した．これにより，DSPソフトウェアの開発フェーズを2つに分けDSPソフトウェア開発の工程を効率化できる事が予想される．2つのフェーズとは，DSPソフトウェアの核となるモジュールの開発フェーズと，MPU側とのインターフェース部分や，複数のDSPソフトウェアのモジュールを連携させるグルー部分の開発フェーズである．前者には従来の様にDSP専用の開発環境を用い，後者には本稿で提案したDSPスクリプト言語を用いる．DSPソフトウェア開発工程は，まず核となるモジュールの開発を行い，続いてDSPスクリプト言語を用いるグルー部分に移る事を可能にする．したがって，従来の開発工程の様にMPUとDSPのソフトウェアのインターフェース部分の実装や，それらのインテグレショーン時に発生するような作業の煩雑さを大幅に軽減できる事が期待できる．

今後は，今回実装したプロトタイプの最適化を行う．コンパイラ部では一時変数の使用方法を効率化しコンパクトな中間コードの生成を実現し，インタプリタ部では各中間コードの実行時のオーバーヘッドを少なくする実装を行う．さらに，DSPスクリプト言語からDSP/BIOSのシステムコールや既存DSPモジュールとの連携を可能にする拡張を検討する．

参考文献

[1] A. Aho, R. Sethi, D. Ullman 著, ”コンパイラI，II”, 原田 賢一 訳, サイエンス社, 1990

[2] “OMAP5910 Technical Reference”, SPRU602B, Jan 2003, Texas Instruments

[3] “TMS320C55x DSP Mnemonic Instruction Set Reference Guide”, SPRU374G, Oct 2002, Texas Instruments

[4] Toshihiro Kobayashi, “Linux DSP Gateway Specification, Version 0.9”, Nokia Corporation, http://dspgateway.sourceforge.net/, 16 May 2003.

[5] John K. Ousterhout, “Scripting: Higher-Level Programming for the 21st Century”, IEEE Computer magazine, March 1998