Create multistandard video engines

By Sumit Gupta
Product Marketing Manager
Tensilica Inc.
E-mail: sumitg@tensilica.com

The growing popularity of handheld multimedia devices is radically changing the requirements placed on multimedia IC providers. No longer can they design chips targeted only at one or two codecs, as today's consumers expect their mobile devices to play media from different sources, coded using different standards, and downloaded or received from a variety of sources. Video codec engines, in particular, must meet these requirements and also be area- and power-efficient.

The last generation of video ASICs was designed to decode and encode mainly MPEG-2, since this is the standard used in DVDs. Some ASICs also supported MPEG-1 to enable VCD playback. In most cases, MPEG-2 codecs were designed using RTL.

Supporting multiple video standards using a hardwired RTL architecture, however, means that each video standard requires a dedicated RTL. Using hardwired RTL blocks to implement a multistandard video engine limits the flexibility of the solution. Implementing a new video standard, upgrading an existing implemented one or fixing a bug would require a silicon respin.

Programmable processors provide more flexibility than RTL blocks to support multiple video standards. Conventional 32bit processors, on the other hand, suffer from performance bottlenecks and are designed for general-purpose code, not for video engines. Embedded DSPs are also not tailored exclusively for video, but instead have hardware functional units, instructions and interfaces for general-purpose DSP applications. To implement video codecs on conventional RISC and DSP processors, the latter have to run at megahertz speeds, use up a lot of memory and burn too much power, making them unsuitable for portable applications.

This becomes evident when examining the number of computations required in one of the video kernels. Take the Sum of Absolute Differences (SAD), a key computation kernel performed during the motion-estimation step of most video encoding algorithms. The SAD algorithm attempts to find the movement of a macroblock between two consecutive video frames by computing the summation of the absolute difference between every set of corresponding pixel values from the two macroblocks.

A simple C code implementation of the SAD kernel is shown in **Listing 1**.

The basic computation inside the SAD kernel is illustrated in **Figure 1a**. The SAD kernel performs subtraction, then computes the absolute and finally merges this with previous results.

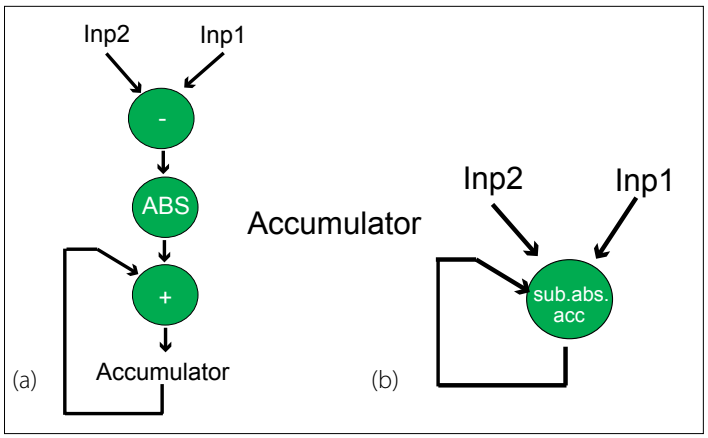

Figure 1: (a) The SAD kernel performs subtraction, then computes the absolute and finally merges this with previous results; (b) A new instruction computes substract-absolute-accumulate.

Computing the SAD of two 16 x 16 macroblocks on a RISC processor requires 256 subtractions, 256 absolute computations and 256 summations—a total of 768 arithmetic operations, which does not include the loads and stores required to move data around. Since this has to be done for all the macroblocks in each frame, it is clearly computationally very expensive and scales as the resolution of the video frame increases.

In fact, on a midrange general-purpose RISC processor that has some DSP instructions such as multiplies and multiply-accumulates, performing an H.264 Baseline decode at CIF resolution requires about 250MHz, and performing H.264 Baseline encode at CIF resolution requires more than 1GHz. That translates to almost 500mW for the processor core alone, not to mention the power being consumed by the memory and the rest of the video SoC. Clearly, this processor cannot be used in a portable device.

```
fcr (row = 0; row < numrows; row++) {
fcr (row = 0; col < numcols; col++) {
        accum += abs ( macroblk1 [row] [col] - macroblk2 [row : [col : ;
} /* column loop */
} /* row loop */
```

Listing 1: A simple C code implementation of the SAD kernel.

More efficient approach

A more efficient implementation of the SAD kernel on a processor would be to create an instruction that performs a "subtract-absolute-add". This reduces the number of arithmetic operations required for a 16 x 16 macroblock from 768 to 256 operations. Also, since a functional unit that implements such a fusion of simple arithmetic operations can usually be optimized into one cycle, this translates to 256 cycles.

This instruction can't be added to a standard 32bit RISC processor core, but it can be added to a configurable processor, which allows the designer to choose from a menu of configuration options and to extend the processor by adding application-specific instructions, register files and interfaces.

Configurability and extensibility are features offered by modern configurable processors, but not by traditional, fixed processors. With configurability, several options are available:

- Instructions the designer needs or doesn't need—e.g. 16x16 multiplies or multiply-accumulates, funnel shifting and floating-point instructions;
- Features such as zero-overhead loops, five or seven pipeline stages and number of local data load/store units;
- Whether to have memory protection, memory translation or a full memory management unit (MMU);
- To have or not have a system bus interface;
- The width of the system bus and local memory interfaces;
- The number and size of the local (tightly coupled) memories;
- The number of interrupts and their types and levels.

With extensibility comes the addition of designer-defined components such as:

- Registers and register files;
- Multicycle, arbitrarily complex functional units;
- SIMD functional units;
- Convert the base processor to a multiple-issue processor;
- Custom interfaces that can be read and written directly

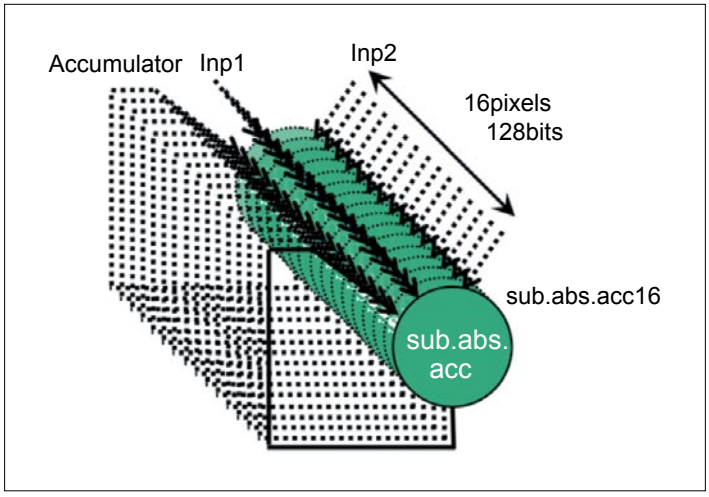

Figure 2: Shown is a simplified view of data path after inserting the sub.abs.acc video-specific functional unit.

from the data path, such as ports or pins on the processor core similar to GPIO pins and queue interfaces to external FIFOs that can be used to interface to other logic or processor cores.

Configurability lets designers build a processor core that is not too big or too small for their application, while extensibility lets them customize the processor to exactly fit the application by creating instructions, register files, functional units and interfaces designed to accelerate the application.

Configurability, extensibility

The key to configurability and extensibility is the ability to automatically generate not only pre-verified RTL for the designer-defined processor, including all extensions, but also the ability to automatically generate a matching, optimized software development tool chain, a cycle-accurate instruction set simulator and system models.

This automation means the compiler generated knows about the new instructions as well as the registers and register files specified by the designer. The compiler thus schedules the designer-defined instructions and performs register allocation. Similarly, the software developer can view the designer-defined registers and register files along with the processor's base register file in the debugger, and the instruction set simulator processes the new instructions specified by the designer. RTOS ports and system models for that particular instance are also automatically generated.

Multi-operation units

It is trivial to add a fusion operation such as SAD in a configurable processor. A new instruction called "sub.abs.acc" computes the "subtract-absolute-accumulate". This new instruction would collapse the operations in Figure 1a to the complex operation in **Figure 1b**.

After this instruction is added, the C compiler will recognize a new C intrinsic, sub.abs.acc, and schedule the corresponding instruction. The debugger will display the internal signals used in the sub.abs.acc functional unit; the assembler will treat this as a new instruction; and the ISS will simulate it in a cycle-accurate way.

Figure 2 shows a simplified view of the data path after this new video-specific functional unit has been inserted. Note that besides generating the logic for the functional unit, the hardware generation tools also automatically insert the forwarding paths, control logic and bypass logic to connect the new functional unit to the rest of the data path.

The modified C code using the C intrinsic for this new instruction for the SAD kernel is shown in **Listing 2**.

It is possible to further optimize this kernel. The inner loop traverses each of the 16 columns in the macroblocks and computes the same operation. This is an ideal situation to create a single instruction, multiple data (SIMD) functional unit and corresponding instruction, sub.abs.acc16, that computes the sub.abs.acc operation on 16 pixels simultaneously (Figure 2).

The corresponding C intrinsic is sub.abs.acc16 and is used to rewrite the C code for the SAD kernel (**Listing 3**).

This reduced the SAD kernel from 768 arithmetic operations down to only 16 arithmetic operations.

The C code above, however, is not accurate. The sub.abs.acc16 instruction requires 128bit inputs from the two macroblocks. This requires support for two things: a 128bit register file and a wide load/store interface. These are also available with configurable processors.

```
for (row = 0; row < numrows; row++) {
for (row = 0; col < numcols; col++) {
        sub.abs.acc ( accum, macroblk1 [row] [col], macroblk2 [row] [col] );
} /* column loop */
} /* row loop */
```

Listing 2: Modified C code using the C intrinsic for new instruction for SAD kernel.

```
for (row = 0; rcw < numrows; row++) {
    sub.abs.acc16 ( accum, macroblk1 [row], macroblk2 [row]) ;
```

Listing 3: The corresponding C intrinsic is sub.abs.acc16 and is used to rewrite the C code for the SAD kernel.

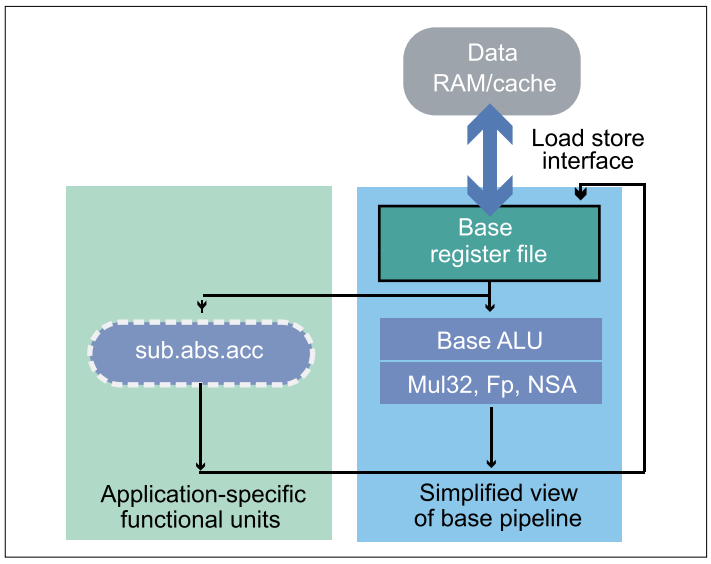

Figure 3: Shown is the processor data path after the compiler has automatically generated the load/store instructions corresponding to this wide load/store interface.

Custom register files

Using a Tensilica Xtensa configurable processor, declaring a custom register file of arbitrary width is as simple as a single-line statement. For example, a 128bit register file with four registers called "myRegFile128" creates a corresponding new C data type, myRegFile128, which can be used in the C/C++ code to declare variables. The software tools also create "move" operations that convert various C data types to this new custom data type. The correct C code for the SAD kernel using the sub.abs.acc16 intrinsic and the new register file is shown in **Listing 4**:

Now, the C/C++ compiler will generate the instructions to move data from the regular C data types to the custom C data type, myRegFile128, and do register allocation for the new register file.

Getting data in and out of such a wide custom register file (and corresponding SIMD functional units) requires the ability to do wide loads and stores. Again, in configurable processors, the designer can specify custom load and store instructions that perform wide load/stores directly to the custom register file. The compiler then automatically generates the load/store instructions corresponding to this wide load/store interface.

An updated view of the processor data path is shown in **Figure 3**. The hardware-generation tools generate the wide, custom register file, the wide load/store interface to data memory and all the associated forwarding, control and bypass logic. The tools also generate hardware logic to move data from the base register file to the user-defined register file.

Control-dominated code

The amount and complexity of the control code in multimedia applications have increased to the point where they consume almost as much computation time and effort as data-intensive portions of the code. For example, a key component in the H.264 main profile decoder is the Context-Adaptive Binary Arithmetic Coding (Cabac) algorithm, which is almost completely a control-flow decision tree with interspersed data computation and comparison.

```
fcr (row = 0; row < numrows; row++) {
        myRegFile128 mblk1, mblk2;
        mblk1 = macroblk1 [row];
        mblk2 = macroblk2 [row];
```

Listing 4: Correct C code for the SAD kernel using the sub.abs.acc16 intrinsic and the new register file.

Most traditional processor solutions offload Cabac to a dedicated RTL accelerator due to the high computational complexity. However, Cabac can be implemented efficiently as a set of instruction extensions on a configurable processor. Also, this implementation does not require data to be shuttled in and out between the processor and the RTL accelerator. This demonstrates another advantage of processor instruction extensions—much finer HW/SW partitioning can be done, since the application-specific hardware is within the processor.