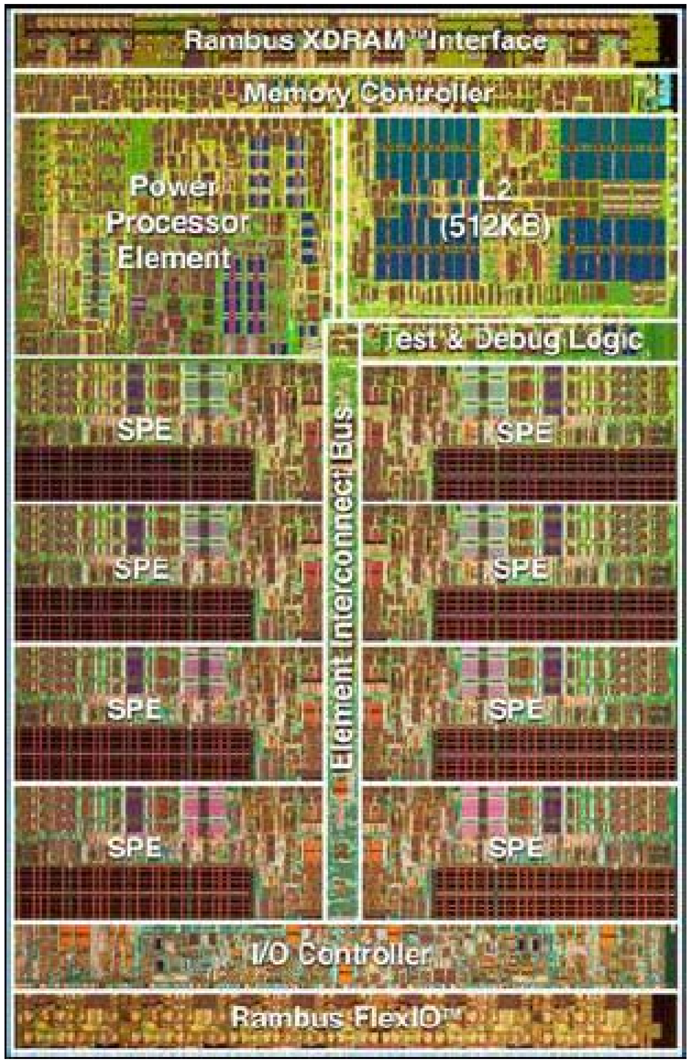
6.189 IAP 2007

Recitation 1

Getting to Know Cell

Recap

- Cell: 9 cores on single chip
  - 1 PPE
  - 8 SPEs
- PPE is a general-purpose PowerPC processor
  - Runs operating system, controls SPEs
- SPEs optimized for data processing
- Cores are connected to each other and memory through high-bandwidth Element Interconnect Bus



What Makes Cell Different (And Difficult)?

- Multiple programs in one
  - PPU and SPU programs cooperate to carry out computation
- SIMD
  - SPU has 128 128-bit registers
  - All instructions are SIMD instructions
  - Registers are treated as short vectors of 8/16/32-bit integers or single/double-precision floats
- SPE local store
  - 256 KB of low-latency storage on each SPE
  - SPU loads/stores/ifetch can *only* access local store
  - Accesses to main memory done through DMA engine
  - Allows program to control and schedule memory accesses
  - Something new to worry about, but potential to be much more efficient

What Makes Cell Different (And Difficult)?

- Multiple programs in one
  - PPU and SPU programs cooperate to carry out computation
- SIMD
  - SPU has 128 128-bit registers
  - All instructions are SIMD instructions
  - Registers are treated as short vectors of 8/16/32-bit integers or single/double-precision floats
- SPE local store
  - 256 KB of low-latency storage on each SPE
  - SPU loads/stores/ifetch can *only* access local store
  - Accesses to main memory done through DMA engine
  - Allows program to control and schedule memory accesses
  - Something new to worry about, but potential to be much more efficient

SPU Programs

- SPU programs are designed and written to work together but are compiled independently
- Separate compiler and toolchain (spuxlc/spu-gcc, etc.)
- Produces small ELF image for each program that can be embedded in PPU program
  - Contains own data, code sections
  - On startup, C runtime (CRT) initializes and provides malloc
  - printf/mmap/some other I/O functions are implemented by calling on the PPU to service the request

A Simple SPU Program

SPU program hello_spu.c

```
#include <stdio.h>

int
main(unsigned long long speid,
     unsigned long long argp,
     unsigned long long envp)
{
  printf("Hello world! (0x%x)\n", (unsigned int)speid);
  return 0;
}
```

$\downarrow$ Compile and embed hello_spu.o

PPU program

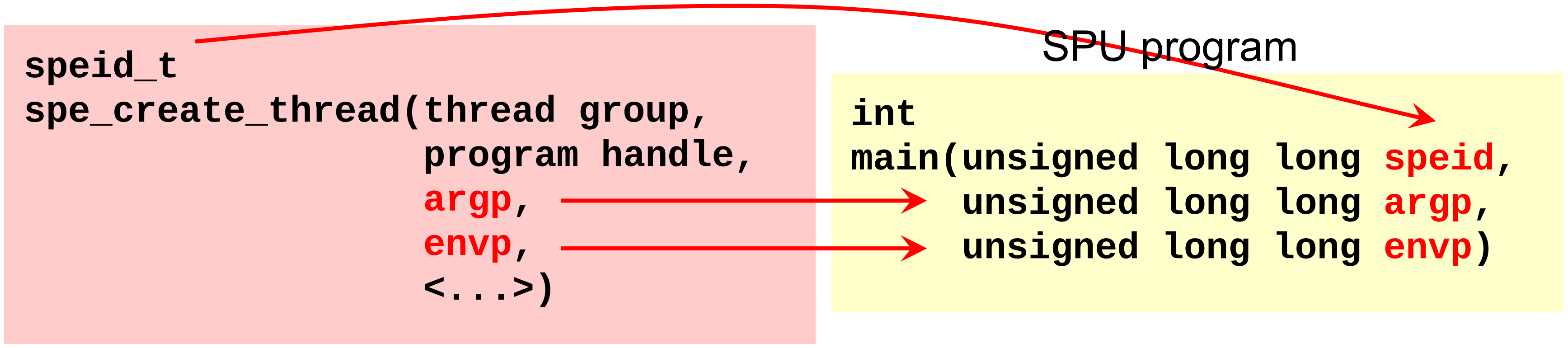
```
extern spe_program_handle_t hello_spu;
```

Running SPU Programs

- SPE runtime management library (libspe)
  - Used by PPE only
- Provides interface similar to pthreads
- Run embedded SPU program as abstracted SPE thread
  - No direct access to SPEs
  - Threads can be scheduled, swapped in/out, paused

libspe

- spe_create_thread



```
speid_t
spe_create_thread(thread group,
                  program handle,
                  argp,
                  envp,
                  <...>)
```

```
int
main(unsigned long long speid,
     unsigned long long argp,
     unsigned long long envp)
```

- spe_wait, spe_kill
- spe_read_out_mbox, spe_write_in_mbox, spe_write_signal
- spe_get_ls
  - Returns memory-mapped address of SPU’s local store
  - PPU/other SPUs can DMA using this address
- spe_get_ps_area
  - Returns memory-mapped address of SPU’s MMIO registers

examples/hello/

A Simple Cell Program

PPU (hello.c)

```
#include <stdio.h>
#include <libspe.h>

extern spe_program_handle_t hello_spu;

int main() {
  speid_t id[8];

  // Create 8 SPU threads
  for (int i = 0; i < 8; i++) {
    id[i] = spe_create_thread(0,
                              &hello_spu,
                              NULL,
                              NULL,
                              -1,
                              0);
  }

  // Wait for all threads to exit
  for (int i = 0; i < 8; i++) {
    spe_wait(id[i], NULL, 0);
  }

  return 0;
}
```

SPU (hello_spu.c)

```
#include <stdio.h>

int
main(unsigned long long speid,
     unsigned long long argp,
     unsigned long long envp)
{
  printf("Hello world! (0x%x)\n", (unsigned int)speid);
  return 0;
}
```

Exercise 1.a (8 minutes)

- Compile and run hello example
  - Fetch tarball

    See example code in recitations section.
  - Unpack tarball
    ```
    tar zxf examples.tar.gz
    ```
  - Go to hello example
    ```
    cd examples/hello
    ```
  - Compile SPU program
    ```
    cd spu
    /opt/ibmcmp/xlc/8.1/bin/spuxlc -o hello_spu hello_spu.c -g -Wl,-N
    embedspu -m32 hello_spu hello_spu hello_spu-embed.o
    ar –qcs hello_spu.a hello_spu-embed.o
    ```
  - Compile PPU program
    ```
    cd ..
    /opt/ibmcmp/xlc/8.1/bin/ppuxlc -o hello hello.c -g -Wl,-m,elf32ppc spu/hello_spu.a -lspe
    ```
  - Run
    ```
    ./hello
    ```

Exercise 1.b (2 minutes)

- Make build system makes the compilation process easier
- Compile using the make build system
  - Set environment variable $CELL_TOP

    ```
    export CELL_TOP=/opt/ibm/cell-sdk/prototype
    ```
  - Remove previously compiled code in directory

    ```
    make clean
    ```
  - Rebuild the program

    ```
    make
    ```
  - Run

    ```
    ./hello
    ```

What Makes Cell Different (And Difficult)?

- Multiple programs in one
  - PPU and SPU programs cooperate to carry out computation
- SIMD
  - SPU has 128 128-bit registers
  - All instructions are SIMD instructions
  - Registers are treated as short vectors of 8/16/32-bit integers or single/double-precision floats
- SPE local store
  - 256 KB of low-latency storage on each SPE
  - SPU loads/stores/ifetch can *only* access local store
  - Accesses to main memory done through DMA engine
  - Allows program to control and schedule memory accesses
  - Something new to worry about, but potential to be much more efficient

SIMD

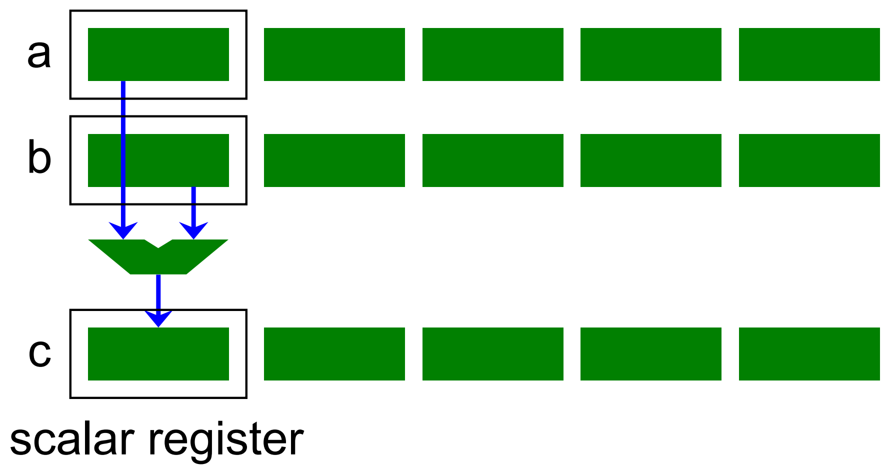
- Single Instruction, Multiple Data
- SIMD registers hold short vectors
- Instruction operates on all elements in SIMD register at once

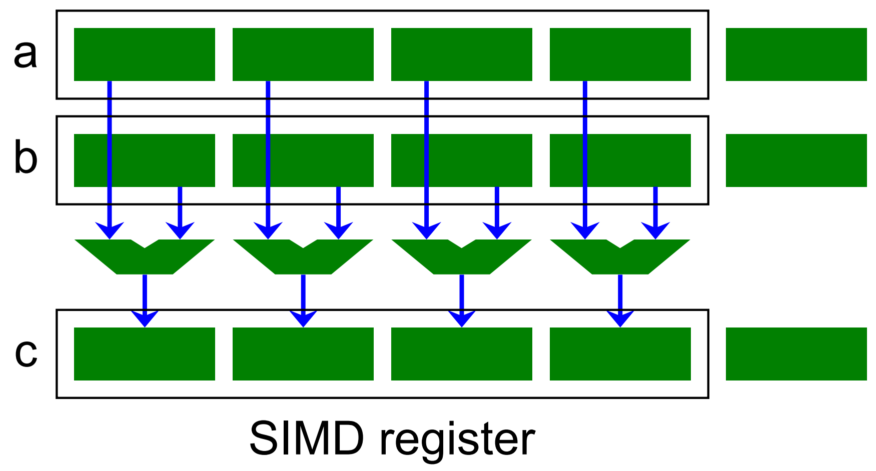
Scalar code

```
for (int i = 0; i < n; i++) {
  c[i] = a[i] + b[i]
}
```

Vector code

```
for (int i = 0; i < n; i += 4) {
  c[i:i+3] = a[i:i+3] + b[i:i+3]
}
```





SIMD

- Can offer high performance
  - Single-precision multiply-add instruction: 8 flops per cycle per SPE
- Scalar code works fine but only uses 1 element in vector
- SPU loads/stores on qword granularity only
  - Can be an issue if the SPU and other processors (via DMA) try to update different variables in the same qword
- For scalar code, compiler generates additional instructions to rotate scalar elements to the same slot and update a single element in a qword
- SIMDizing code is important
  - Auto SIMDization (compiler optimization)
  - Intrinsics (manual optimization)

SPU Intrinsics

- Vector data types
  - vector signed/unsigned char/short/int/long long
  - vector float/double
  - 16-byte vectors
- Intrinsics that wrap SPU instructions
- e.g. vector integer multiply-add instruction/intrinsic

```
int *data;

  for (int i = 0; i < cb.num_elements; i++) {
    data[i] = data[i] * MUL_FACTOR + ADD_FACTOR;
  }
```

```
vec_int4 *data;  // vec_int4 = vector with 4 ints

  for (int i = 0; i < cb.num_elements / 4; i++) {
    data[i] = spu_madd(*(vec_short8 *)&data[i],
                       (vec_short8)MUL_FACTOR,
                       (vec_int4)ADD_FACTOR);
  }
```

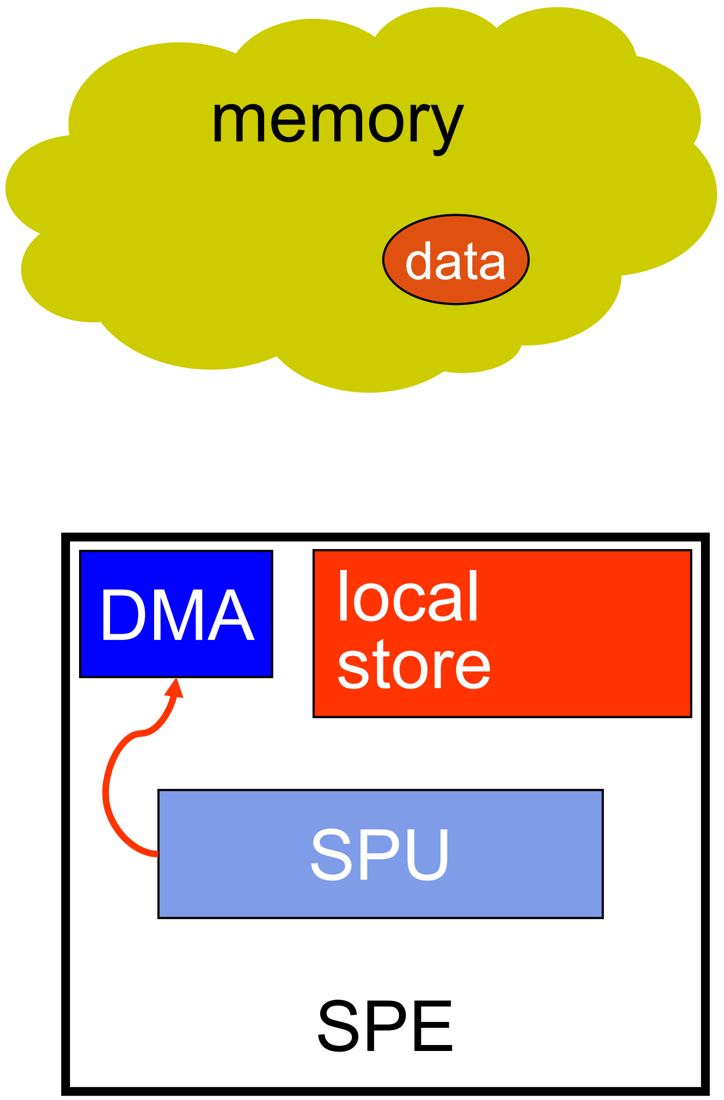
What Makes Cell Different (And Difficult)?

- Multiple programs in one
  - PPU and SPU programs cooperate to carry out computation
- SIMD
  - SPU has 128 128-bit registers
  - All instructions are SIMD instructions
  - Registers are treated as short vectors of 8/16/32-bit integers or single/double-precision floats
- SPE local store
  - 256 KB of low-latency storage on each SPE
  - SPU loads/stores/ifetch can *only* access local store
  - Accesses to main memory done through DMA engine
  - Allows program to control and schedule memory accesses
  - Something new to worry about, but potential to be much more efficient

Data In and Out of the SPE Local Store

- SPU needs data

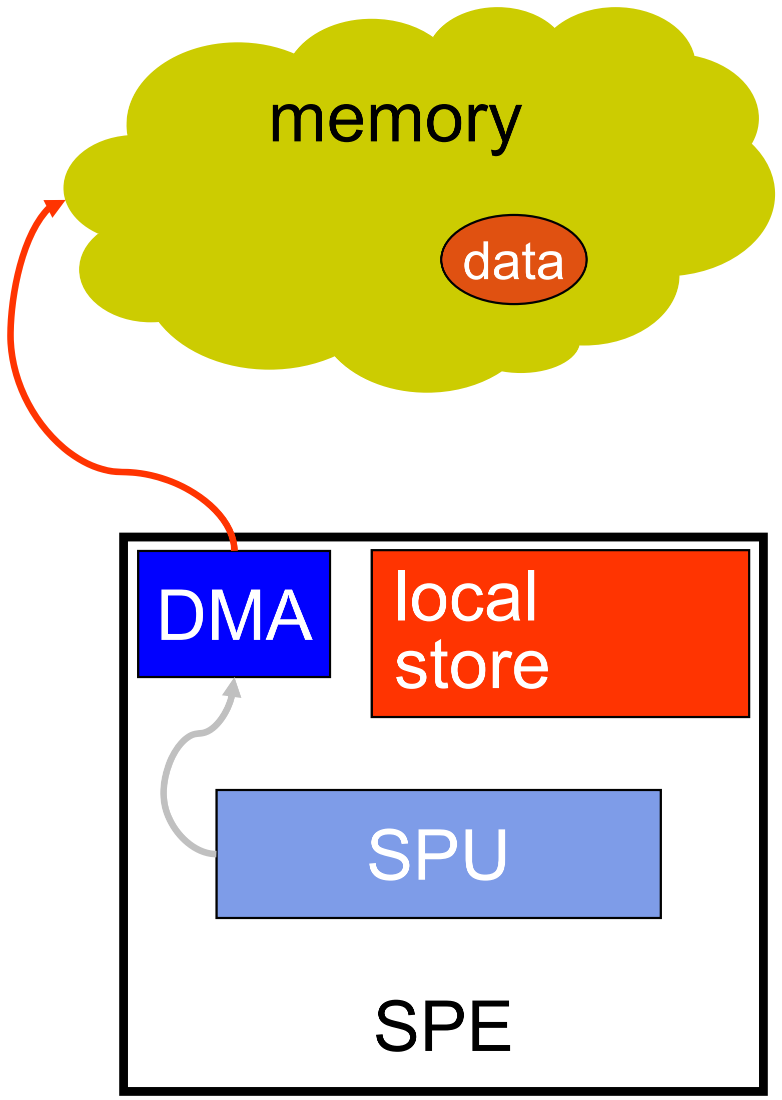
1. SPU initiates DMA request for data



Data In and Out of the SPE Local Store

- SPU needs data

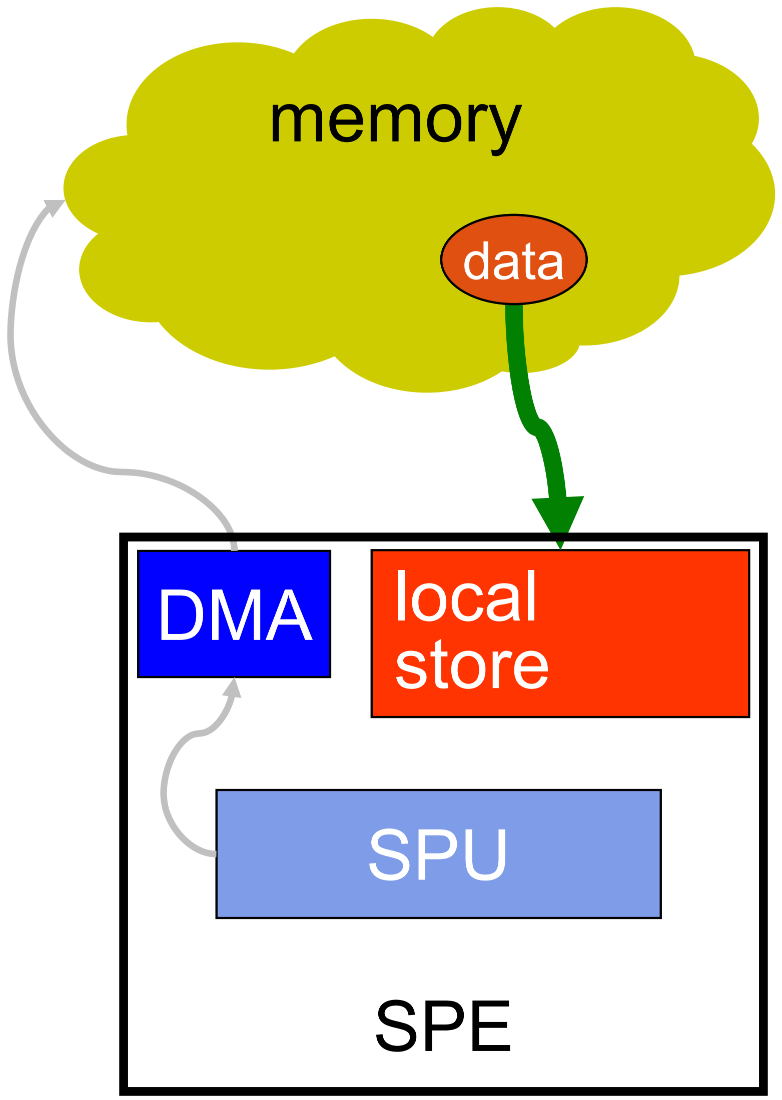
1. SPU initiates DMA request for data
2. DMA requests data from memory



Data In and Out of the SPE Local Store

- SPU needs data

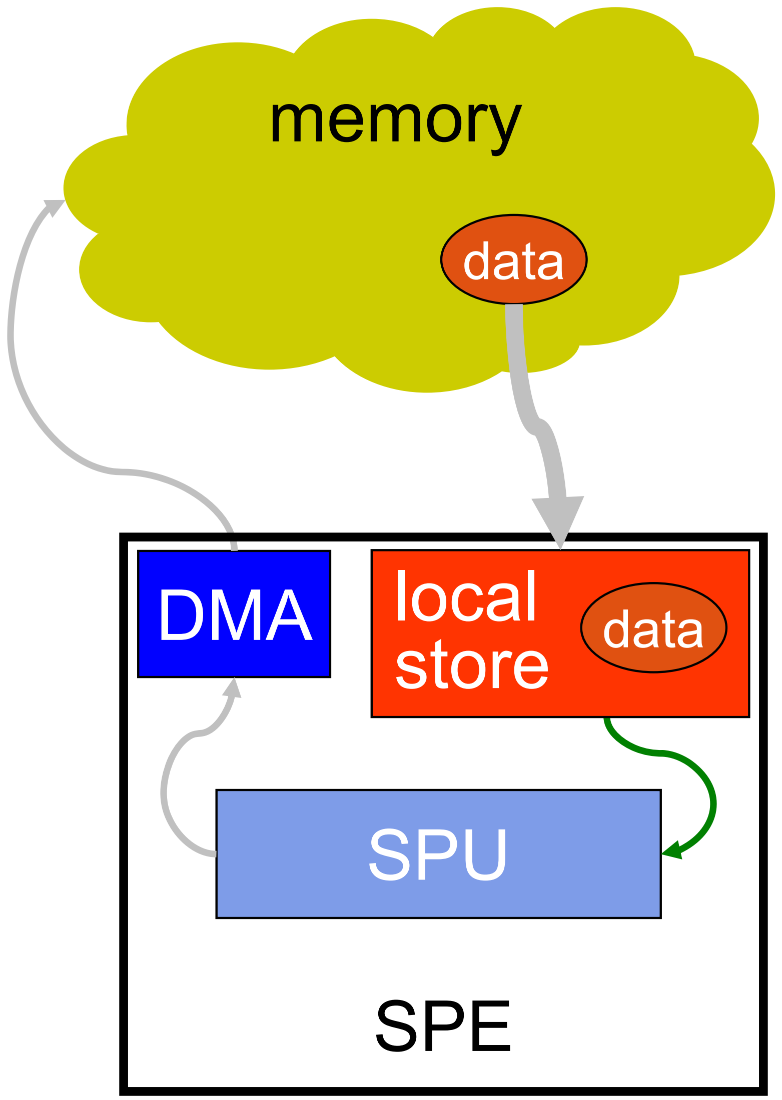
1. SPU initiates DMA request for data
2. DMA requests data from memory
3. Data is **copied** to local store



Data In and Out of the SPE Local Store

- SPU needs data

1. SPU initiates DMA request for data
2. DMA requests data from memory
3. Data is copied to local store
4. SPU can access data from local store

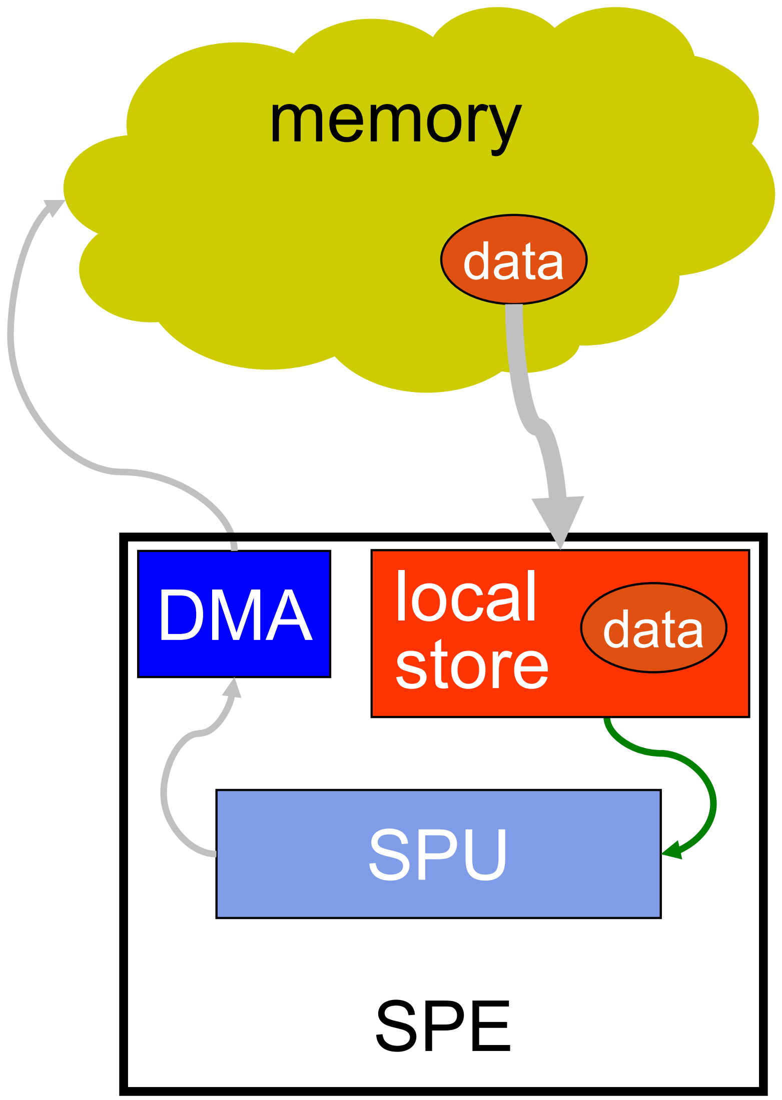


Data In and Out of the SPE Local Store

- SPU needs data

1. SPU initiates DMA request for data
2. DMA requests data from memory
3. Data is copied to local store
4. SPU can access data from local store

- SPU operates on data then **copies** data from local store back to memory in a similar process



DMA and SPEs

- 1 Memory Flow Controller (MFC) per SPE
- High bandwidth – 16 bytes/cycle
- DMA transfers initiated using special channel instructions
- DMA transfers between virtual address space and local store
  - SPE uses PPE address translation machinery
  - Each SPE local store is mapped in virtual address space
    - Allows direct local store to local store transfers
    - Completely on chip, very fast
- Once DMA commands are issued, MFC processes them independently
  - SPU continues executing/accessing local store
  - Communication-computation concurrency/multibuffering essential for performance

DMA and SPEs

- Each MFC can service up to 24 outstanding DMA commands
  - 16 transfers initiated by SPU
  - 8 additional transfers initiated by PPU
    - PPU initiates transfers by accessing MFC through MMIO registers
- Each DMA transfer is tagged with 5-bit program-specified tag
  - Multiple DMAs can have same tag
  - SPU/PPU can wait or poll for DMA completion by tag mask
  - Can enforce ordering among DMAs with same tag

DMA Alignment

- 1/2/4/8/16-byte transfers that are naturally aligned
- Multiples of 16 bytes up to 16 KB per transfer
- DMA transfers of 16 bytes or less are atomic, no guarantee for anything else
- Memory and local store addresses must have same offset within a qword (16 bytes)
- DMA list commands
  - SPU can generate list of accesses in local store
  - Transfers between discontinuous segments in virtual address space to contiguous segment in local store
  - MFC processes list as single command

Mailboxes and Signals

- Facility for SPE to exchange small messages with PPE/other SPEs
  - e.g. memory address, “data ready” message
- From perspective of SPE
  - 1 inbound mailbox (4-entry FIFO) – send messages to this SPE
  - 1 outbound mailbox (1-entry) – send messages from this SPE
  - 1 outbound mailbox (1-entry) that interrupts PPE – send messages from this SPE to PPE
  - 2 signal notification registers – send messages to this SPE
    - Act as 1 entry or 32 independent bits
  - 32 bits
- SPU accesses its own mailboxes/signals by reading/writing to channels with special instructions
  - Read from inbound mailbox, signals
  - Write to outbound mailboxes
  - Accesses will stall if empty/full

Mailboxes and Signals

- SPE/PPE accesses another SPE mailboxes/signals through MMIO registers
  - Accesses do not stall
  - Read outbound mailboxes
  - Write inbound mailbox, signals
  - Accesses by multiple processors must be synchronized
  - If inbound mailbox overflows, last item is overwritten
  - Reading outbound mailbox when no data may return garbage

DMA

- From SPU

```
mfc_get(destination LS addr,
        source memory addr,
        # bytes,
        tag,
        <...>)
```

```
mfc_put(source LS addr,
        destination memory addr,
        # bytes,
        tag,
        <...>)
```

  - Also list commands: mfc_getl, mfc_putl
  - mfc_stat_cmd_queue
    - Queries number of free DMA command slots
    - Similar functions to query available mailbox/signal entries
- From PPU (libspe)
  - spe_mfc_get, spe_mfc_put
  - No list commands

DMA Example

- Array of integers in memory that we want to process on SPU
- Need to tell SPU program
  - Location (address) of array
  - Size of array
  - Additional parameters?
- Approach
  - Fill in control block in main memory
  - Pass address of control block to SPU
  - Have SPU DMA control block to local store

```
typedef struct {
  uintptr32_t data_addr;
  uint32_t num_elements;
  ...
} CONTROL_BLOCK;
```

Generic C code

```
for (int i = 0; i < NUM_ELEMENTS; i++) {
  data[i] = data[i] * MUL_FACTOR + ADD_FACTOR;
}
```

examples/dma/

DMA Example

Generic C code

```
for (int i = 0; i < NUM_ELEMENTS; i++) {
  data[i] = data[i] * MUL_FACTOR + ADD_FACTOR;
}
```

PPU

```
// Data array
int data[NUM_ELEMENTS] __attribute__((aligned(128)));

CONTROL_BLOCK cb __attribute__((aligned(16)));

int main() {
  ...

  // Fill in control block
  cb.data_addr = data;
  cb.num_elements = NUM_ELEMENTS;

  // Create SPU thread
  id = spe_create_thread(0, &dma_spu, &cb,
                         NULL, ...);
```

SPU

```
CONTROL_BLOCK cb __attribute__((aligned(16)));
int *data;

int main(speid, argp, envp) {
  // DMA over control block
  mfc_get(&cb, argp, sizeof(cb), 5, ...);

  // Mask out tag we're interested in
  mfc_write_tag_mask(1 << 5);
  // Wait for DMA completion
  mfc_read_tag_status_all();
  // Compare mfc_read_tag_status_any/immediate

  // Allocate 128-byte aligned buffer
  data = malloc_align(data_size, 7);

  // DMA over actual data
  mfc_get(data, cb.data_addr, data_size, 5, ...);

  // Wait for DMA completion
  mfc_read_tag_status_all();
```

examples/dma/

DMA Example

Generic C code

```
for (int i = 0; i < NUM_ELEMENTS; i++) {
  data[i] = data[i] * MUL_FACTOR + ADD_FACTOR;
}
```

PPU

```
  // Wait for mailbox message from SPU
  while (spe_stat_out_mbox(id) == 0);

  // Drain mailbox
  spe_read_out_mbox(id);

  // Done!
  ...
}
```

SPU

```
  // Process the data
  for (int i = 0; i < cb.num_elements; i++) {
    data[i] = data[i] * MUL_FACTOR + ADD_FACTOR;
  }

  // DMA back results
  mfc_put(data, cb.data_addr, data_size, 5, ...);

  // Wait for DMA completion
  mfc_read_tag_status_all();

  // Notify PPU using outbound mailbox
  spu_write_out_mbox(0);
  return 0;
}
```

- Assumed entire array fits in one DMA command (16 KB)
- Assumed array size is multiple of 16 bytes

DMA Example 2

- Add 2 arrays of integers and store result in 3rd array
- Same approach
  - Fill in control block in main memory
  - Pass address of control block to SPU
  - SPU DMAs control block to LS
  - SPU DMAs both input arrays to LS
  - SPU DMAs result back to memory

```
typedef struct {
  uintptr32_t data1_addr;
  uintptr32_t data2_addr;
  uintptr32_t result_addr;
  uint32_t num_elements;
  ...
} CONTROL_BLOCK;
```

Generic C code

```
for (int i = 0; i < NUM_ELEMENTS; i++) {
  result[i] = data1[i] + data2[i];
}
```

examples/dma_2arr/

DMA Example 2

Generic C code

```
for (int i = 0; i < NUM_ELEMENTS; i++) {
  result[i] = data1[i] + data2[i];
}
```

PPU

```
// Data and result arrays
int data1[NUM_ELEMENTS] __attribute__((aligned(128)));
int data2[NUM_ELEMENTS] __attribute__((aligned(128)));
int result[NUM_ELEMENTS] __attribute__((aligned(128)));

CONTROL_BLOCK cb __attribute__((aligned(16)));

int main() {
  ...

  // Fill in control block
  cb.data1_addr = data1;
  cb.data2_addr = data2;
  cb.result_addr = result;
  cb.num_elements = NUM_ELEMENTS;

  // Create SPU thread
  id = spe_create_thread(0, &dma_spu, &cb,
                         NULL, ...);
```

SPU

```
CONTROL_BLOCK cb __attribute__((aligned(16)));
int *data1, *data2, *result;

int main(speid, argp, envp) {
  // DMA over control block
  mfc_get(&cb, argp, sizeof(cb), 5, ...);

  // Mask out tag we’re interested in
  mfc_write_tag_mask(1 << 5);
  // Wait for DMA completion
  mfc_read_tag_status_all();

  // Allocate 128-byte aligned buffers for data
  // and results
  ...

  // Start DMA for both input arrays with same tag
  mfc_get(data1, cb.data1_addr, data_size, 5, ...);
  mfc_get(data2, cb.data2_addr, data_size, 5, ...);

  // Wait for completion of both transfers
  mfc_read_tag_status_all();
```

examples/dma_2arr/

DMA Example 2

Generic C code

```
for (int i = 0; i < NUM_ELEMENTS; i++) {
  result[i] = data1[i] + data2[i];
}
```

PPU

```
  // Wait for mailbox message from SPU
  while (spe_stat_out_mbox(id) == 0);

  // Drain mailbox
  spe_read_out_mbox(id);

  // Done!
  ...
}
```

SPU

```
  // Process the data
  for (int i = 0; i < cb.num_elements; i++) {
    result[i] = data1[i] + data2[i];
  }

  // DMA back results
  mfc_put(result, cb.result_addr, data_size, 5,
          ...);

  // Wait for DMA completion
  mfc_read_tag_status_all();

  // Notify PPU using outbound mailbox
  spu_write_out_mbox(0);
  return 0;
}
```

- Same assumptions
  - Each array fits in one DMA command (16 KB)
  - Array sizes are multiples of 16 bytes

examples/dma_2spe/

SPE-SPE DMA Example

- Streaming data from SPE to SPE
- Distribute computation so one SPE does multiplication, another does addition

```
for (int i = 0; i < cb.num_elements; i++) {
  data[i] = data[i] * MUL_FACTOR + ADD_FACTOR;
}
```

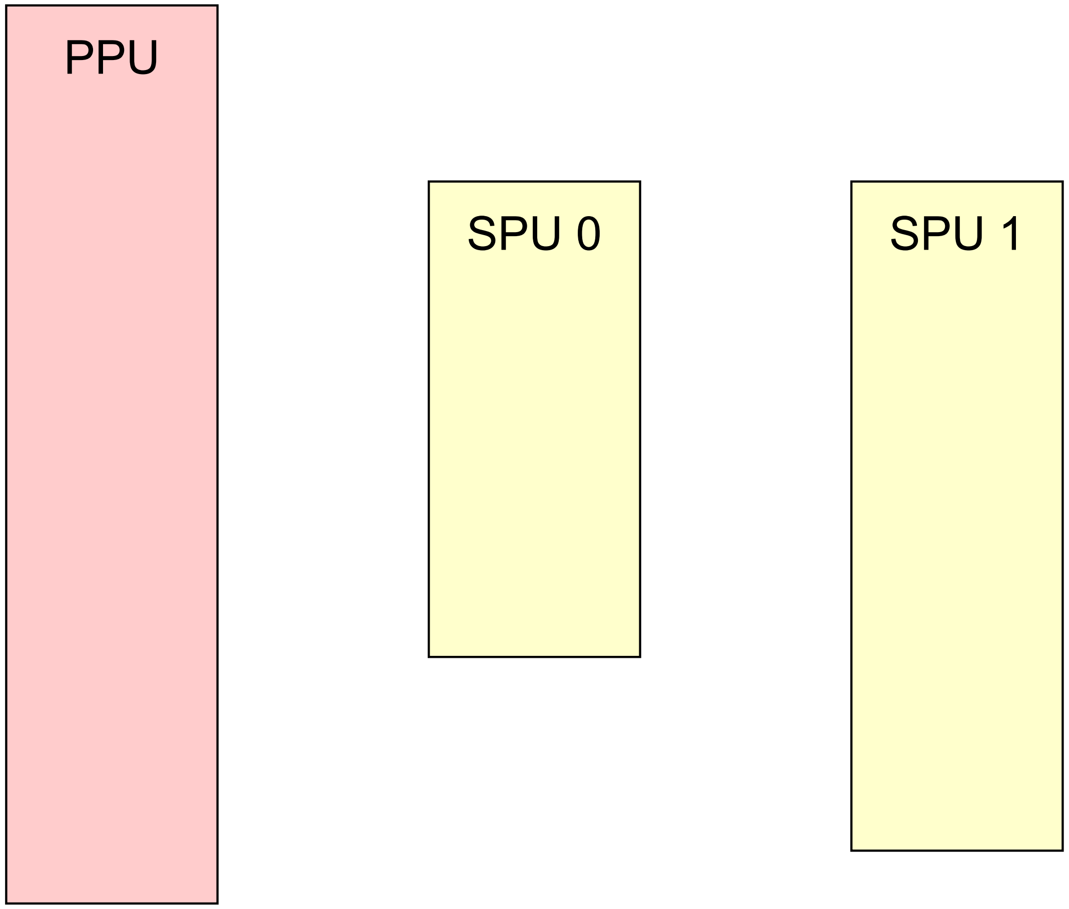
- Keep actual data transfer local store to local store
- Communication?
  - PPE orchestrates all communication
  - SPEs talk to each other via mailboxes/signals

examples/dma_2spe/

SPE-SPE DMA Example

- SPEs that communicate with each other need to know:
  - Addresses of local stores
  - Addresses of MMIO registers
- Only PPU program (via libspe) has access to this information
  - PPU creates SPE threads, gathers address information, informs SPEs

SPE-SPE DMA Example

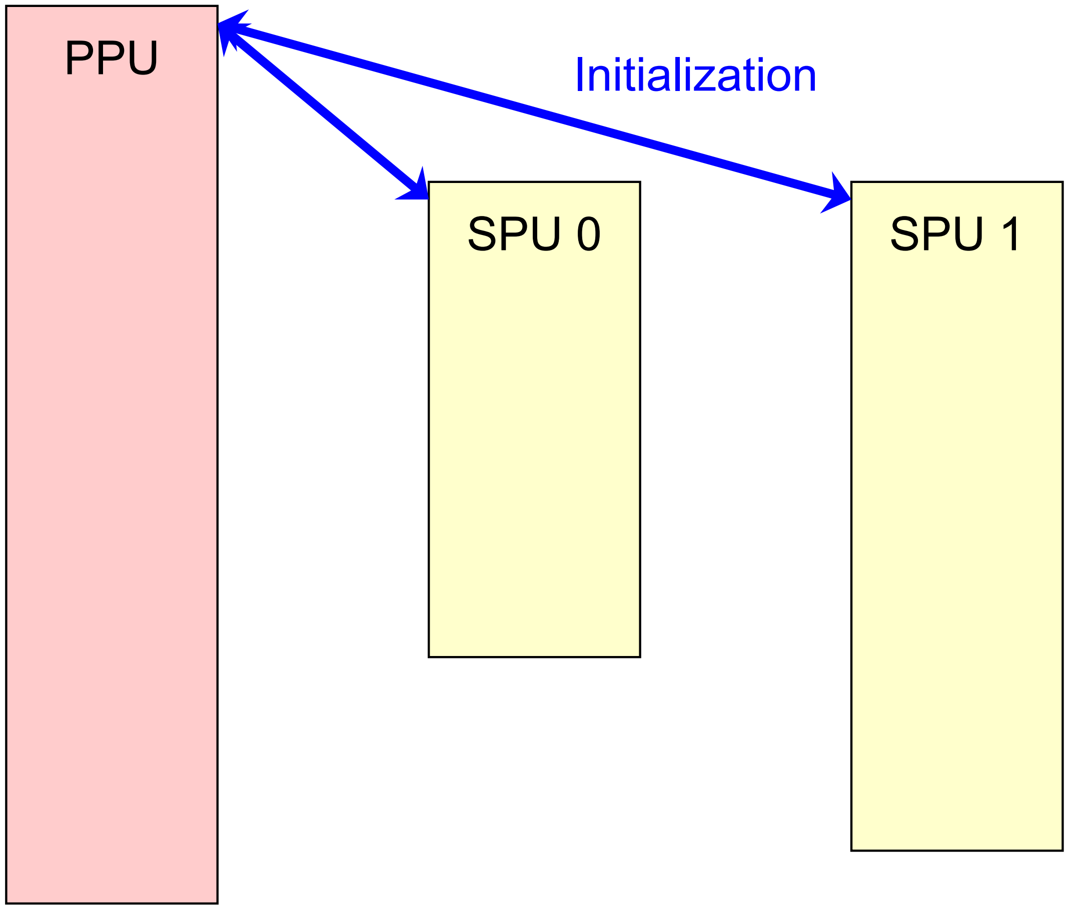


PPU

```
// Create SPE threads for multiplier and
// adder
id[0] = spe_create_thread(0, &dma_spu0, 0,
                                    NULL, ...);
id[1] = spe_create_thread(0, &dma_spu1, 1,
                                    NULL, ...);
```

examples/dma_2spe/

SPE-SPE DMA Example



PPU

```
typedef struct {
  uintptr32_t spu_ls[2];
  uintptr32_t spu_control[2];
  ...
} CONTROL_BLOCK;

// Fill in control block
for (int i = 0; i < 2; i++) {
  cb.spu_ls[i] = spe_get_ls(id[i]);
  cb.spu_control[i] =
    spe_get_ps_area(id[i], SPE_CONTROL_AREA);
}
...

// Send control block address to all SPUs
for (int i = 0; i < 2; i++) {
  spe_write_in_mbox(id[i], &cb);
}
```

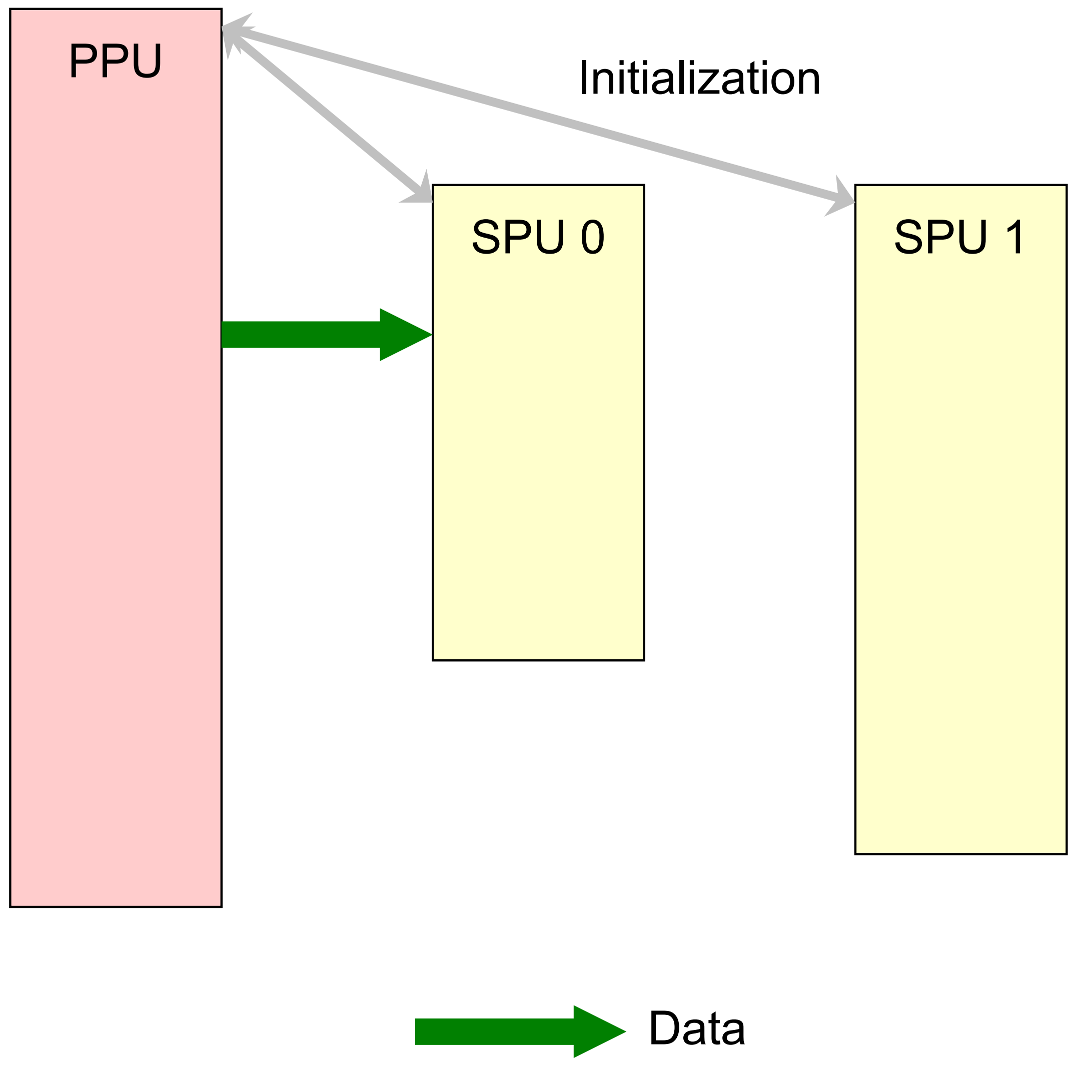
Both SPUs

```
// Wait for control block address from PPU
cb_addr = spu_read_in_mbox();

// DMA over control block and wait until done
mfc_get(&cb, cb_addr, sizeof(cb), 5, ...);
mfc_write_tag_mask(1 << 5);
mfc_read_tag_status_all();
```

examples/dma_2spe/

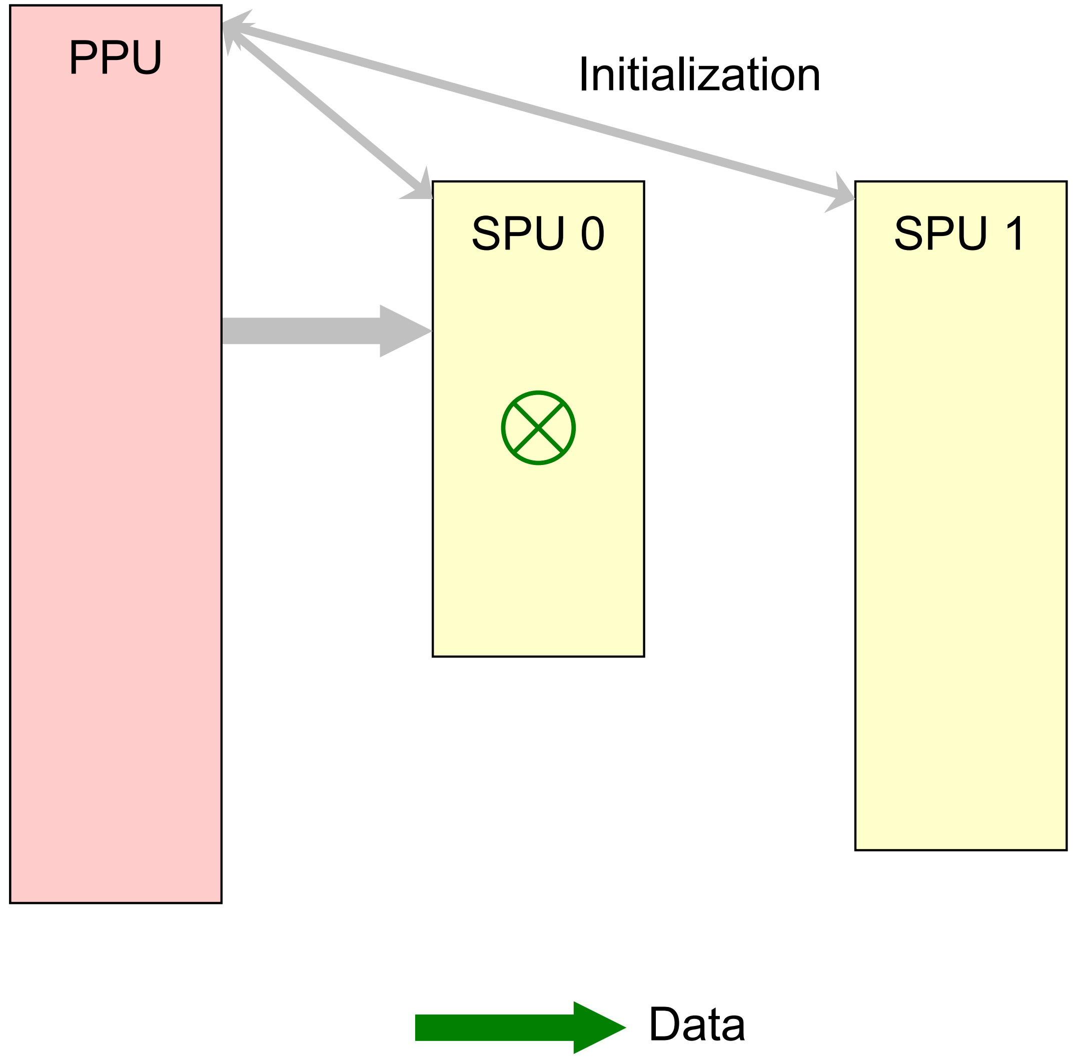
SPE-SPE DMA Example



SPU 0

```
// DMA in data from memory and wait until
// complete
mfc_get(data, cb.data_addr, data_size, ...);
mfc_read_tag_status_all();
```

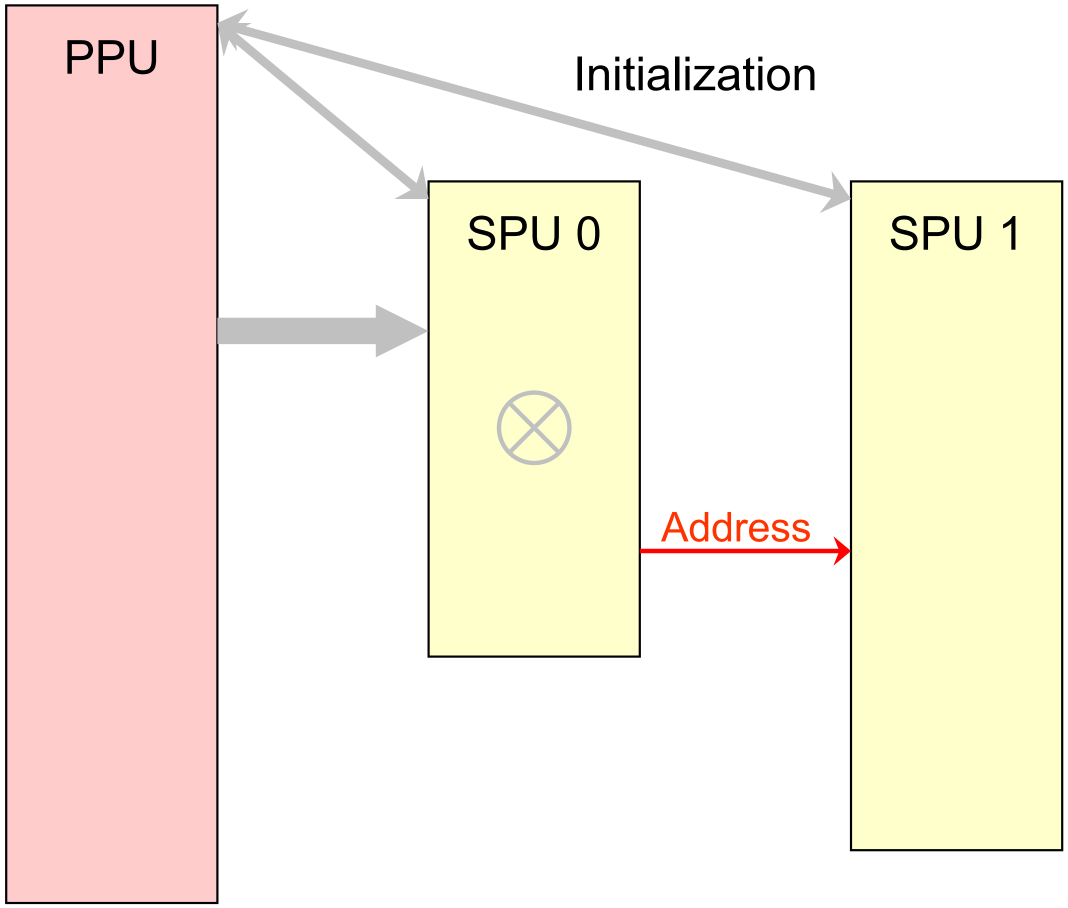
SPE-SPE DMA Example

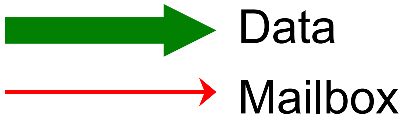


SPU 0

```
// Process data
...
```

SPE-SPE DMA Example





SPU 0

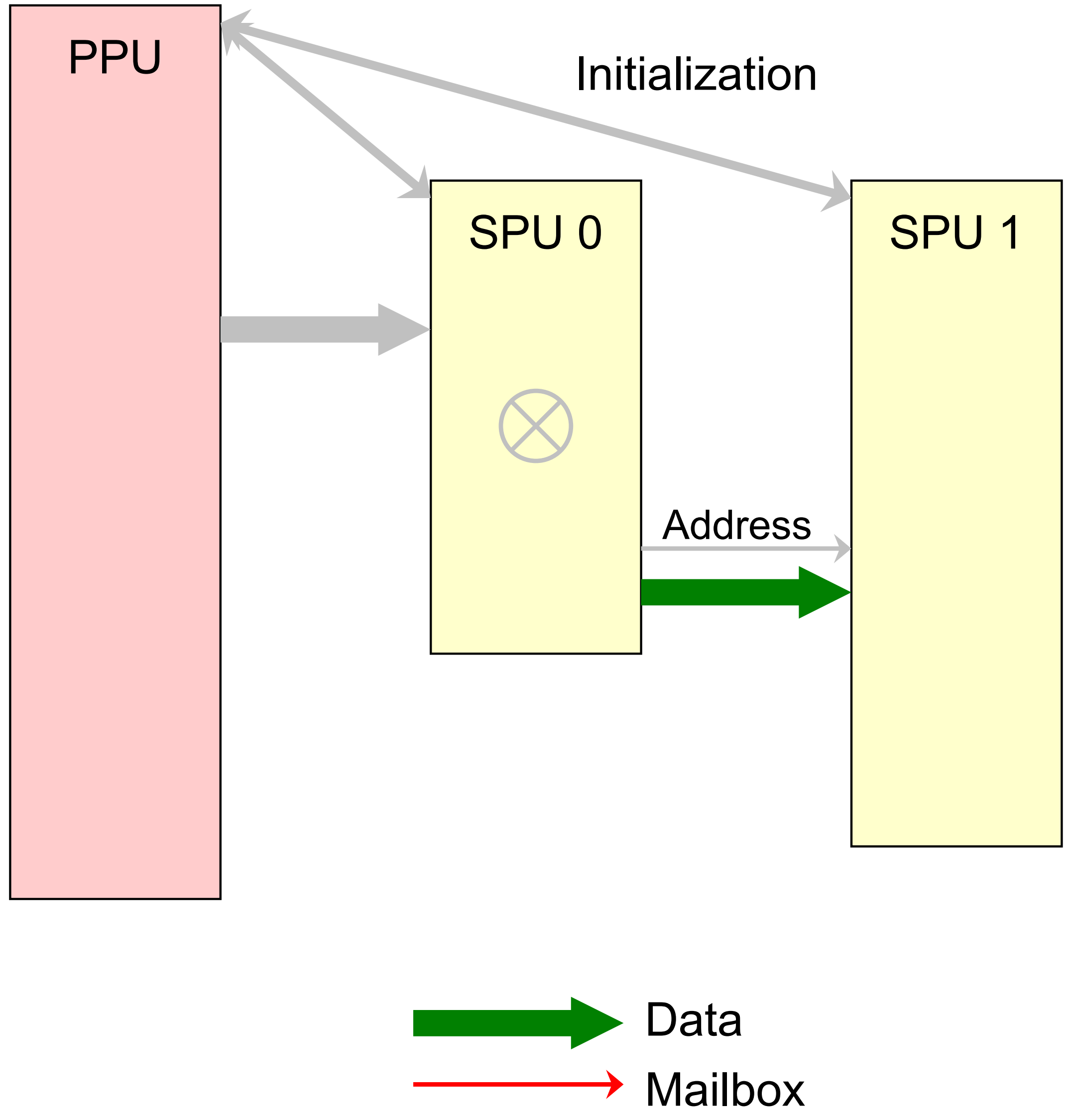
```
// Temporary area used to store values to be
// sent to mailboxes with proper alignment.
struct {
  uint32_t padding[3];
  uint32_t value;
} next_mbox __attribute__((aligned(16)));

// Notify SPU 1 that data is ready. Send over
// virtual address so SPU 1 can copy it out.
next_mbox.value = cb.spu_ls[0] + data;
mfc_put(&next_mbox.value,
        cb.spu_control[1] + 12,
        4,
        ...);
```

SPU 1

```
// Wait for mailbox message from SPU 0
// indicating data is ready.
data_addr = spu_read_in_mbox();
```

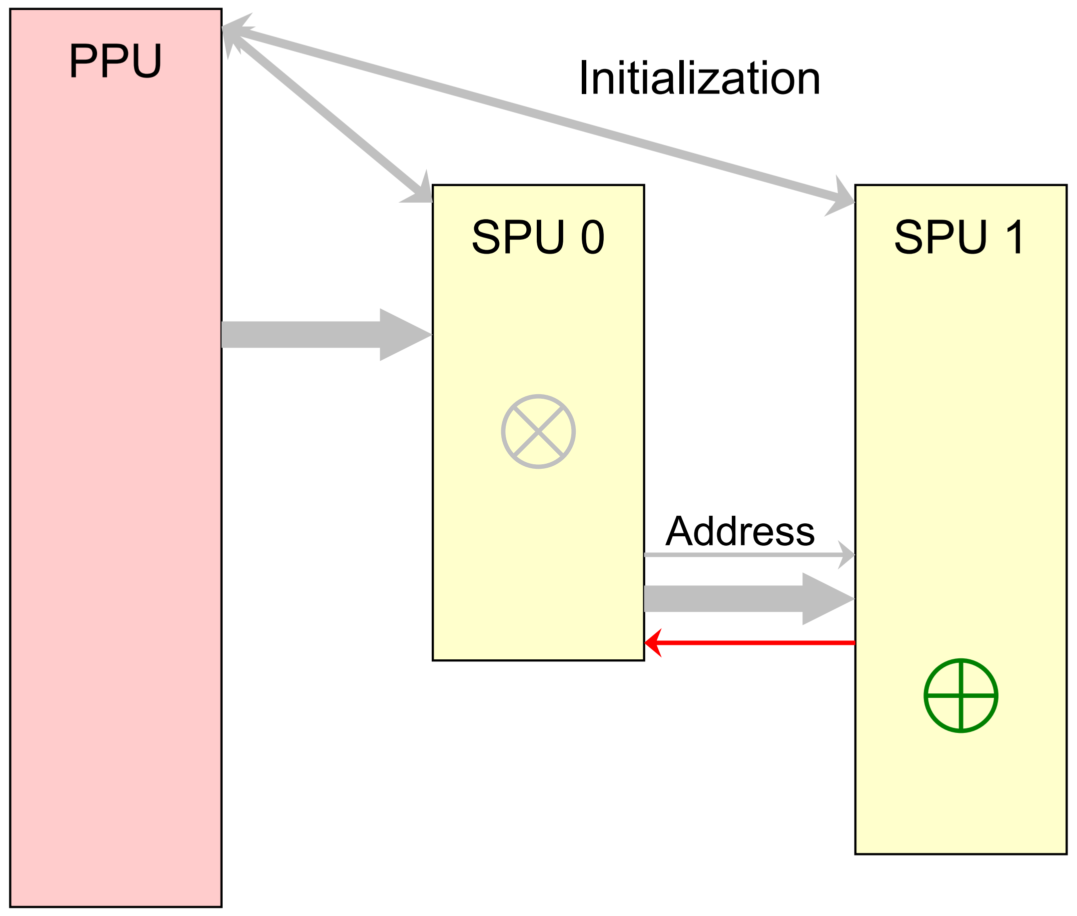
SPE-SPE DMA Example



SPU 1

```
// DMA in data from SPU 0 local store and
// wait until complete.
mfc_get(data, data_addr, data_size, ...);
mfc_read_tag_status_all();
```

SPE-SPE DMA Example



SPU 1

```
// Notify SPU 0 that data has been read.
mfc_put(<garbage>,
        cb.spu_control[0] + 12,
        4,
        ...);

// Process data
...
```

SPU 0

```
// Wait for acknowledgement from SPU 1.
spu_read_in_mbox();
return 0;
```

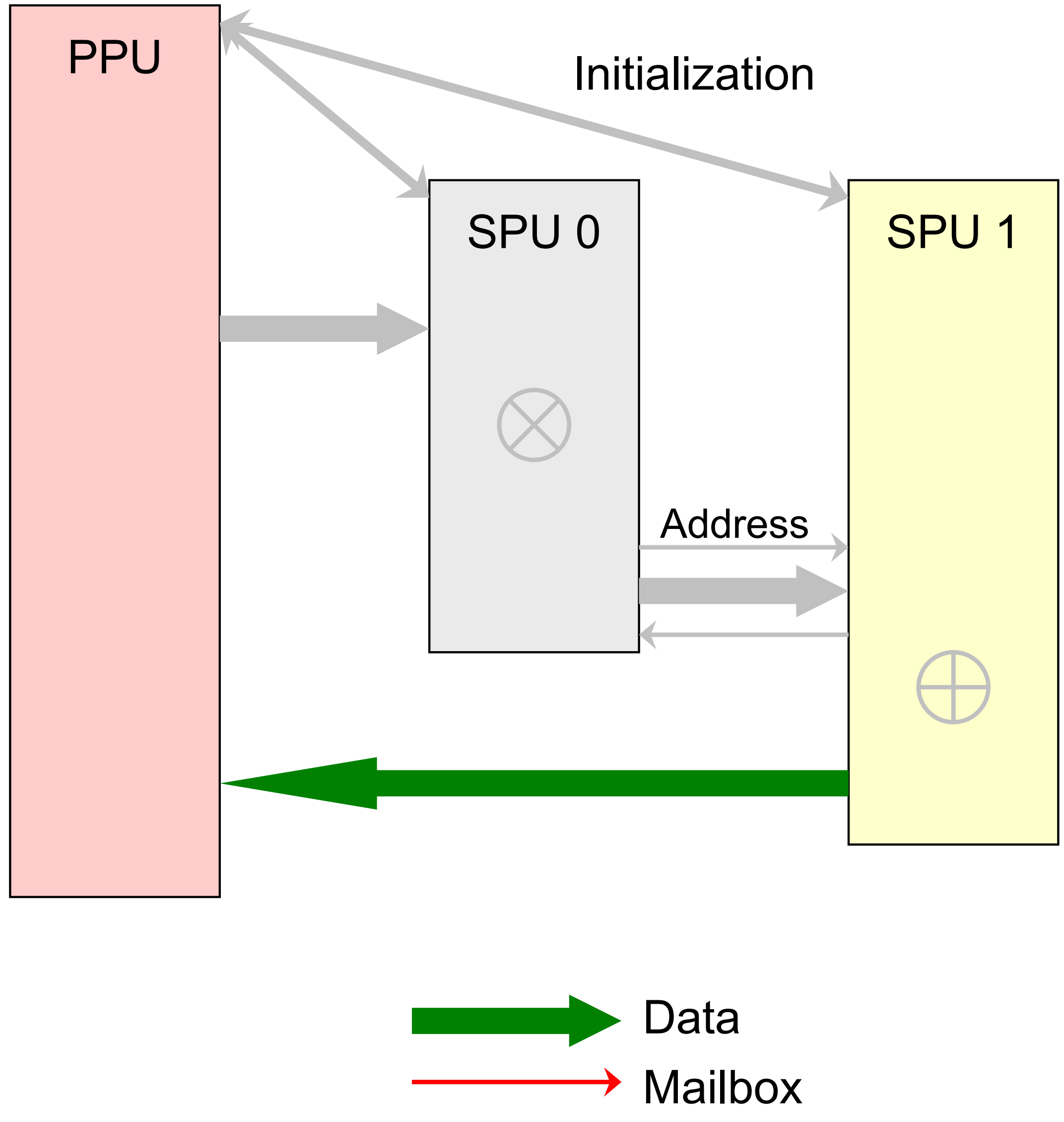
🟩 Data

→ Mailbox

SPE-SPE DMA Example

SPU 1

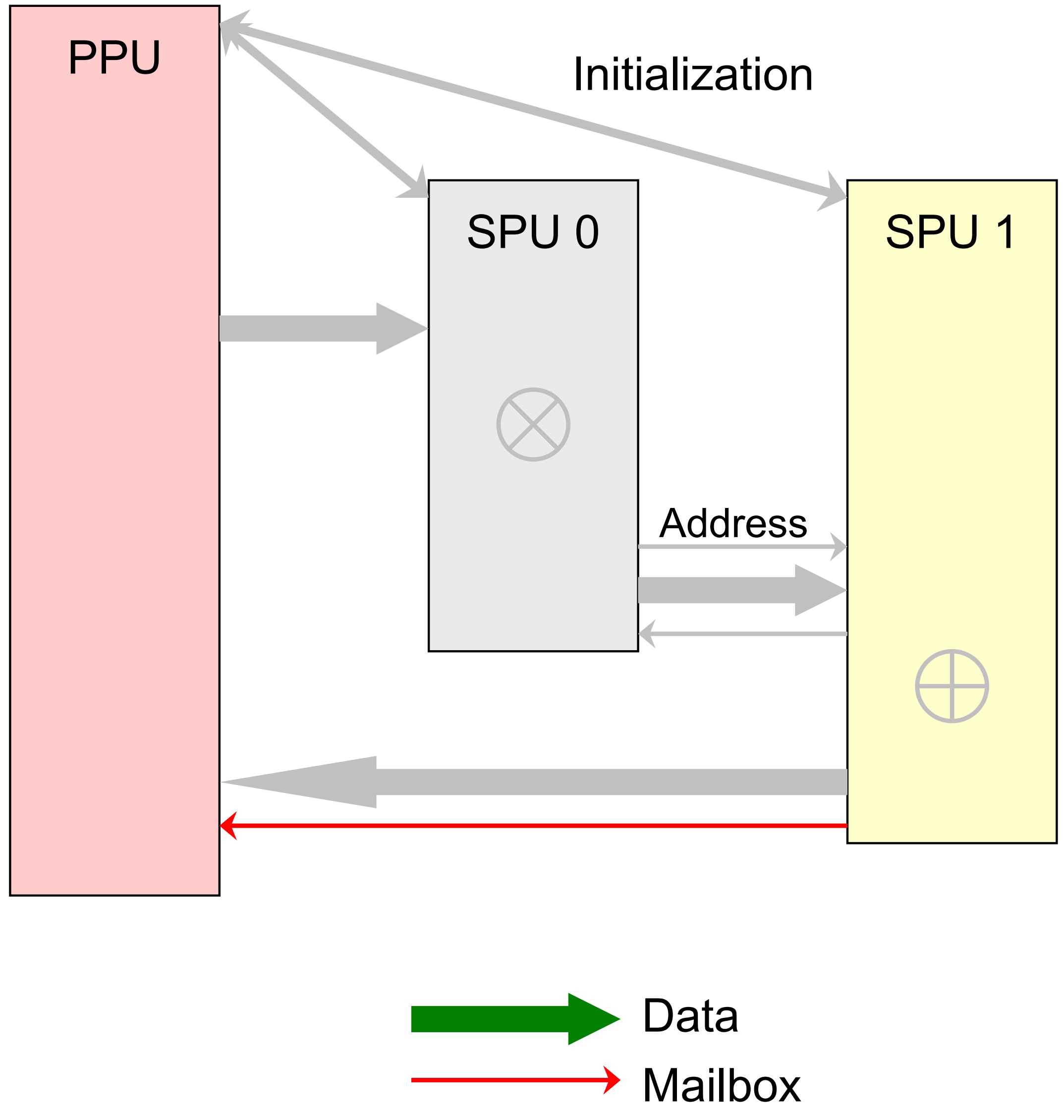
```
// DMA processed data back to memory and wait
// until complete
mfc_put(data, cb.data_addr, data_size, ...);
mfc_read_tag_status_all();
```



SPE-SPE DMA Example

SPU 1

```
// Notify PPU.
spu_write_out_mbox(0);
return 0;
```



Documentation

- Cell Broadband Engine resource center
  - http://www-128.ibm.com/developerworks/power/cell
  - Tutorial (very useful)
  - Documentation for:
    - SPE runtime management library (libspe)
    - SPU C language extension (intrinsics)
  - Programmer's Handbook (more detailed information)
  - PowerPC architecture books (SIMD on PPU)
- Samples and library source in SDK directory
  - /opt/ibm/cell-sdk/prototype/src